



Aprendiendo Perl - Primera Edición

Cooperativa Venezolana de Tecnologías Libres R.S.

1 de febrero de 2011

info@covetel.com.ve

Índice general

1. Introducción	7
1.1. Sobre Perl	7
1.2. ¿ Porque Larry creo Perl ?	7
1.3. ¿ Porque Larry simplemente no uso otro lenguaje ?	7
1.3.1. Características principales de Perl	8
1.4. ¿ Perl es fácil o difícil ?	8
1.5. ¿ Como se hizo Perl tan popular ?	9
1.6. ¿ Que esta pasando con Perl ahora ?	9
1.7. ¿ Que es CPAN ?	10
1.8. ¿ Como hacer un programa en Perl ?	10
1.9. Un programa simple	10
1.10. Sobre los ejercicios y sus respuestas	10
2. Datos Escalares	11
2.1. Números	11
2.1.1. Todos los números internamente tienen el mismo formato	11
2.1.2. Literales de punto flotante	11
3. Listas y Arreglos	13
3.1. Accediendo los Elementos de un Array	13
3.2. Indices Especiales para un Array	14
3.3. Listas Literales	15
3.4. El atajo qw	15
3.5. Asignación de listas	16
3.6. Los operadores push y pop	17
3.7. Los operadores shift y unshift	18
3.8. Interpolando Arrays en Cadenas	18
3.9. La estructura de control foreach	19
3.10. La variable mágica de Perl: \$_	19
3.11. El operador reverse	20
3.12. El operador sort	20
3.13. Contexto Escalar y Contexto de Lista	20
3.13.1. Usando Expresiones de listas en contexto escalar	21
3.13.2. Usando Expresiones escalares en contexto de lista	22
3.14. Forzar el contexto escalar	22
3.15. STDIN en contexto de lista	22

3.16. Ejercicios	23
4. Subrutinas	25
4.1. Definiendo una Subrutina	25
4.2. Llamar una Subrutina	26
4.3. Retornar Valores	26
4.4. Argumentos	27
4.5. Variables privadas en Subrutinas	28
4.6. Longitud de la lista de parámetros	29
4.6.1. Subrutina &max bien escrita	29
4.7. Notas sobre Variables Léxicas (my)	30
4.8. Usando el Pragma Strict	30
4.9. El operador return	31
4.10. Omitir el signo &	32
4.11. Retornando valores no escalares	32
4.12. Variables Privadas Persistentes	33
4.13. Ejercicios	34
5. Entrada y Salida	37
5.1. Entrada desde la Entrada Estándar	37
5.2. Invocación de Argumentos	39
5.3. Salida hacia la Salida Estándar	40
5.4. Formatear la salida con printf	41
5.5. Arrays y printf	42
5.6. Filehandles	42
5.6.1. Abrir un Filehandle	43
5.6.2. Filehandles Malos	44
5.6.3. Cerrar un Filehandle	44
5.7. Errores fatales con die	44
5.8. Mensajes de Advertencia con warn	45
5.9. Usando los Filehandle	45
5.9.1. Cambiar un Filehandle de salida por defecto	46
5.9.2. Reabrir un Filehandle por defecto	46
5.10. Salida con say	47
5.11. Ejercicios	47
6. Hashes	49
6.1. ¿ Que es un Hash ?	49
6.2. ¿ Porque usar un Hash ?	50
6.3. Acceder al elemento de un Hash	52
6.4. Hash como un todo	53
6.5. Asignación de Hash	53
6.6. La Gran Flecha	54

6.7.	Operadores de Hash	54
6.7.1.	Las funciones keys y values	54
6.7.2.	La función each	55
6.7.3.	La función exists	55
6.7.4.	La función delete	55
6.7.5.	Interpolación de un elemento en un Hash	55
6.7.6.	El hash %ENV	56
6.8.	Ejercicios	56
7.	El mundo de las Expresiones Regulares	57
7.1.	¿ Qué son las expresiones regulares ?	57
7.2.	Usando Patrones Simples	58
7.3.	Sobre los Metacaracteres	58
7.4.	Cuantificadores Simples	58
7.5.	Agrupación de Patrones	59
7.6.	Alternación	61
7.7.	Clases o Grupos de Caracteres	61
7.8.	Atajos para Clases de Caracteres	62
7.9.	Atajos Negativos	62
7.10.	Ejercicios	62

1 Introducción

1.1. Sobre Perl

Perl tiene unos 23 años hasta ahora. El lenguaje ha evolucionado desde una herramienta simple para la administración de sistemas tomando cosas de shell scripting y C (Perl 1) hasta convertirse en un poderoso lenguaje de propósito general (Perl 5), consistente, coherente, cambiando el paradigma de programación en general con la intención de mantenerse por otros 25 años. (Perl 6)

Aun así, la mayoría de los programas en el mundo, escritos en Perl 5, aprovechan muy poco las características del lenguaje. Usted **puede** escribir programas en Perl 5 como si estuviera en Perl 4 (o Perl 3 o 2 o 1), pero los programas que aprovechan todo el increíble poder del mundo de Perl 5 que la comunidad ha inventado, y mejorado, son mas cortos, rápidos, mas poderosos, y fáciles de mantener que sus alternativas en anteriores versiones.

1.2. ¿ Porque Larry creo Perl ?

Larry crea Perl a mediados de 1980, cuando intentaba producir reportes sobre la información ordenada jerárquicamente en archivos de la red de noticias USENET para un sistema de reporte de bugs, y awk no dio la talla. Entonces Larry, como buen programador perezoso que es, decide hacer una herramienta que pueda usar para resolver este problema y que pueda volver a usar para otro problema similar en otro lugar, en otro tiempo.

1.3. ¿ Porque Larry simplemente no uso otro lenguaje ?

En este momento hay gran variedad de lenguajes, pero en esa época Larry no encontró un lenguaje que se adaptara a sus necesidades.

Si uno de los lenguajes de hoy, hubiese existido para la época, pues tal vez Larry lo hubiera usado para resolver el problema.

Necesitaba algo con la rapidez de codificación disponible en **Shell** o **AWK** y con algo del poder de las herramientas como **grep**, **cut**, **sort**, y **sed**, sin tener que recurrir a un lenguaje como **C**.

Perl, intenta llenar la brecha entre los lenguajes de bajo nivel como C, C++, y Assembler, y los lenguajes de alto nivel de la época como Shell Scripting. La programación a bajo nivel, usualmente es difícil y fea de escribir, pero rápida y sin limitaciones, es difícil superar la velocidad de un programa bien escrito a bajo nivel en una máquina dada. Y no hay mucho que se pueda hacer.

1 Introducción

En el otro extremo, la programación convencional de alto nivel, tiende a ser lenta, dura, fea y limitada. Hay varias cosas que no puedes hacer con programación en shell o batch.

Perl es fácil, casi ilimitado, muy rápido y solo un poco feo.

1.3.1. Características principales de Perl

Perl es fácil

Cuando nos referimos que Perl es fácil, hablamos de que es fácil de usar, no es especialmente fácil de aprender. Si usted conduce un carro, que paso semanas o meses aprendiendo a conducir, entonces luego va a ser fácil de conducir. De igual manera cuando pase la misma o mas cantidad de horas aprendiendo Perl, entonces después va a ser muy fácil de usar.

Perl es casi ilimitado

Hay muy pocas cosas que no podrás hacer con Perl. Seguramente usted no deseará escribir un controlador de algún dispositivo de interrupción a nivel de micro-kernel (A pesar de que ya alguien hizo esto). Pero la mayoría de las cosas que la gente común y no tan común necesita, la mayoría del tiempo son buenas tareas para Perl, desde scripts muy pequeños, hasta aplicaciones de nivel industrial.

Perl es sobre todo rápido

Esto es porque no hay nadie en el desarrollo que no lo utilice, de modo que entonces todos queremos que Perl sea rápido. Si alguien quiere añadir una nueva característica que sea genial, pero que introduzca lentitud a otros aspectos, es casi seguro que Larry va a rechazar la nueva característica hasta que no se encuentre una manera de escribirla de manera eficiente.

Perl es un poco feo

Esto es cierto, el camello se ha convertido en el símbolo de Perl, desde la cubierta del legendario libro **The Camel Book**, conocido también como **Programming Perl by Larry Wall, Tom Christianesen, and Jon Orwant**. Los camellos también son un poco feos, pero trabajan duro, incluso en condiciones difíciles. Los camellos siempre están ahí siempre para hacer el trabajo a pesar de todas las dificultades, incluso cuando se ven y huelen mal le escupen. Perl es un poco así.

1.4. ¿ Perl es fácil o difícil ?

Perl es fácil de usar, pero a veces difícil de aprender. Esto por su puesto es una generalización. En el diseño de Perl, Larry tuvo que hacer muchas consideraciones. Cuando ha tenido la oportunidad de hacer algo más fácil para el programador a costa de ser más

difícil para el estudiante, ha decidido casi todo el tiempo a favor del programador. Esto es así, porque usted va a aprender Perl una sola vez, pero lo vas a usar una y otra vez. Perl posee cualquier cantidad de comodidades que le permiten al programador ahorrar tiempo. Por ejemplo, la mayoría de las funciones tienen un valor predeterminado, con frecuencia, el valor por defecto es justamente la manera en la que usted desea usar la función. De esta manera, podrás encontrar líneas de código en Perl como las siguientes:

```
1   while (<>) {  
2       chomp;  
3       print join("\t", (split /\:/)[0, 2, 1, 5] ), "\n";  
4   }
```

En su totalidad, sin usar los atajos y los valores por defecto de Perl, este trozo de código sería aproximadamente 10 o 12 veces mas largo, entonces tomaría mas tiempo para escribirlo y leerlo. Va a ser mas duro de mantener y depurar cuando tenga mas variables.

1.5. ¿ Como se hizo Perl tan popular ?

Después de jugar un rato con Perl, agregando cosas aquí y allá, Larry se lanzó a la comunidad de lectores de la Usenet, comúnmente conocida como **La Red**. Los usuarios de esta flota fugitiva que trabajaban en sistemas en todo el mundo (decenas de miles de ellos), le dieron retroalimentación, pidiendo a Larry, maneras de hacer esto, aquello o lo otro, muchos de los cuales Larry nunca había imaginado en su hasta el momento poco manejo de Perl.

Pero como resultado, Perl creció, y creció y creció. Creció en características. Creció en portabilidad. Lo que antes era un lenguaje poco disponible en solo un par de sistema tipo Unix, ahora tiene miles de páginas de documentación en linea gratis, docenas de libros, y miles de listas de correo en cientos de idiomas con un número incontable de lectores, e implementaciones en varios sistemas de uso cotidiano hoy en día, y adicionalmente un curso como este.

1.6. ¿ Que esta pasando con Perl ahora ?

Durante estos días, Larry Wall no escribe código, pero el sigue guiando el desarrollo y toma las decisiones importantes. Perl es mayormente mantenido por un grupo de resistentes y valientes personas llamadas **The Perl 5 Porters**. Puede suscribirse a la lista de correo en **perl5-porters@perl.org** para seguir el trabajo de esta gente y sus debates. Al momento de escribir esta documentación, muchas cosas están pasando con Perl. Durante los últimos años, muchas personas han estado trabajando en la próxima versión de Perl: Perl 6.

No tire a un lado Perl 5, este sigue siendo la versión actual y estable. No se espera una version estable de Perl 6 durante un rato largo. Perl 5 hace todo lo de siempre, y siempre sera así. Perl 5 no va a desaparecer cuando Perl 6 salga. Y la gente puede seguir usando Perl 5 durante muchos años mas.

1.7. ¿ Que es CPAN ?

CPAN es **Comprehensive Perl Archive Network**, su primera parada para Perl. Posee el código fuente de Perl, listo para instalar y los **ports** para todos los sistemas no Unix, ejemplos, documentación, y extensiones del lenguaje.

CPAN es replicado en cientos de máquinas al rededor del mundo, comenzando en <http://search.cpan.org> o <http://kobesearch.cpan.org> para navegar o buscar un paquete. Si no tienes acceso a la red, todo CPAN puede caber en 8 o 10 GB. Es importante mencionar que debido a los cambios diarios en CPAN, un mirror con dos años sin actualizares es definitivamente una antigüedad.

1.8. ¿ Como hacer un programa en Perl ?

Ya es tiempo de hacer esta pregunta (aunque usted no la halla hecho aun), Los programas de Perl son archivos de texto plano, usted puede crearlos y editarlos en su editor de textos preferido, yo personalmente prefiero **VIM**. (Usted no necesita entornos de desarrollo especiales, aunque hay disponibles editores comerciales de varios proveedores. Nunca hemos utilizado uno de estos lo suficiente como para recomendarlo). Por lo general debería usar un editor de texto pensado para programadores, en lugar de un editor de textos convencional. ¿ Cual es la diferencia ? Bueno, un editor pensado para programadores le va a permitir hacer cosas que necesitan los programadores, como sangrado automático o subrayado de bloques de código, o indicar automáticamente la llave de cierre de un bloque de código. En sistemas operativos tipo Unix, los editores de texto mas populares son **Emacs** y **Vim**. BBEdit y TextMate son buenos para Mac OS X, mucha gente ha hablado muy bien sobre UltraEdit y PFE en Windows. La página del manual perlfaq2 tiene una lista de otros editores que también pueden usarse.

1.9. Un programa simple

Según la regla mas antigua de los libros de computación, cualquier libro sobre algún lenguaje de programación que tenga raíces Unix, va a comenzar con **Hola, Mundo**, por ejemplo en Perl seria así:

```
1      #!/usr/bin/perl
2      print "Hello, world!\n";
```

1.10. Sobre los ejercicios y sus respuestas

Los ejercicios se encuentran al final de cada capítulo, una vez que se desarrolle cada capítulo, se tomaran 45 a 50 minutos para resolver los ejercicios correspondientes a cada capítulo.

2 Datos Escalares

2.1. Números

Aunque los números y las cadenas con frecuencia se pueden tratar como escalares, es útil observarlos inicialmente por separado. Vamos a ver primero los números y luego pasamos a las cadenas.

2.1.1. Todos los números internamente tienen el mismo formato

Como vera en los próximos párrafos, se puede especificar tanto números enteros (como 255 o 2001) y números de coma flotante (números reales con puntos decimales, como 3.1416 o 1.35×10^{25}). Pero internamente, Perl calcula valores punto flotantes de doble precisión. Esto significa que no hay valores enteros internamente en Perl. Una constante entera en Perl es tratada como su valor equivalente en coma flotante. Probablemente no se dará cuenta de la conversión, deje de buscar las distintas operaciones con enteros (en oposición a las operaciones de punto flotante), porque simplemente no existen.¹

2.1.2. Literales de punto flotante

Un literal es la forma en que un valor se representa en el código fuente en Perl. Un literal no es el resultado de una operación de calculo o de una operación de I/O. Es datos escritos directamente en el código fuente.

Un literal de coma flotante, ya debe serle familiar. Números con y sin punto flotante también son permitidos (incluyendo el prefijo opcional mas y menos), y de remate el indicador de notación exponencial E.

```
1      1.25
2      255.000
3      255.0
4      7.25e45
5
```

¹Existe un pragma llamado Integer, que permite realizar operaciones con enteros en lugar de punto flotante, pero es otra cosa, y no es de lo que estamos hablando en este punto

3 Listas y Arreglos

Ya hablamos en el capítulo anterior de la singularidad de Perl, en este capítulo vamos a hablar sobre la pluralidad. La pluralidad en Perl esta representada por las listas y los arreglos.

Una lista es una colección ordenada de escalares. Un array es una variable que contiene una lista. En Perl, estos dos términos se utilizan a menudo como si fueran intercambiables. Pero, para ser precisos, una lista es datos, y un array es una variable. En este sentido, puedes tener una lista de valores que no estén contenidos en un array, pero cada variable de tipo array contiene una lista (incluso si la lista esta vacía). Por ejemplo:

```
1      36
2      12.4
3      42
4      "hola"
5      1.72e30
6      "bye\n"
```

Cada elemento en un array o una lista, es una variable escalar independiente con un valor independiente. Estos valores están ordenados, es decir, tienen un secuencia particular desde el primer hasta el último elemento. Los elementos de un array o una lista están **indexados** por pequeños enteros comenzando por el cero ¹ (0) y contando de uno en uno, entonces el primer elemento de cualquier array o lista es siempre el elemento cero (0).

Puesto que cada elemento es un valor escalar independiente, una lista o una matriz puede contener números, cadenas, valores undef, o cualquier mezcla de diferentes valores escalares. Sin embargo, es más común tener todos los elementos del mismo tipo, tales como una lista de títulos de libros (todas son cadenas) o una lista de cosenos (todos son números).

Las matrices o listas pueden tener cualquier número de elementos. La lista mas pequeña es aquella que no tiene elementos, mientras que la mas grande puede llenar toda la memoria disponible. Una vez mas esto es parte de la filosofía de Perl “No hay limites innecesarios”.

3.1. Accediendo los Elementos de un Array

Si ha usado arrays en otro lenguaje, no se debe sorprender al encontrar que perl provee un mecanismo de subíndice del array para referirse a un elemento por un índice numérico.

Los elementos de un array son enumerados usando una secuencia de enteros, comenzando en cero (0) e incrementando de uno en uno para cada elemento. Por ejemplo:

¹En Perl, es posible cambiar el número de inicio de un array o una lista indexada. Larry luego considero esto como una mala característica y su (ab)uso esta fuertemente desaconsejado

```
1     $prueba[0] = "yabba";
2     $prueba[1] = "dabba";
3     $prueba[2] = "doo";
```

El nombre del array en este caso “prueba”, es de un espacio de nombres totalmente separado de los escalares, puedes tener una variable escalar llamada \$prueba en el mismo programa, y perl va a tratarlas como cosas diferentes y no va a confundirse.

Puede usar un elemento de un array como por ejemplo \$prueba[0] en cualquier lugar como si se tratara de un escalar como \$prueba. Por ejemplo, puede obtener el valor de un elemento del array o cambiar su valor con algunas expresiones cortas y prácticas que vimos en el capítulo anterior:

```
1     print $prueba[0];
2     $prueba[2] = "diddley";
3     $prueba[1] .= "whatsis";
```

Por su puesto, el subíndice puede ser cualquier expresión que devuelva un valor numérico. Si no es un entero, automáticamente va a ser truncado al próximo entero menor. Por ejemplo:

```
1     $number = 2.71828;
2     print $prueba[$number - 1]; # Es lo mismo que imprimir $prueba[1]
```

Si se indica un elemento en el subíndice que este mas allá del final del array, el valor correspondiente va a ser `undef`. Al igual que con las variables escalares, si nunca ha guardado un valor en una variable escalar, pues entonces el valor de esta variable va a ser `undef`. Por ejemplo:

```
1     $blank = $prueba[ 142_857 ]; # obtengo undef
2     $blanc = $mel;               # escalar no usado $mel obtengo undef
```

3.2. Índices Especiales para un Array

Si guardas en un elemento del array que esta mas allá del final del array, este automáticamente se extiende hasta donde sea necesario, aquí no hay límite de longitud, el límite básicamente es la memoria que Perl tenga disponible para usar. Si Perl necesita crear elementos intermedios, estos son creados con valores `undef`. Por ejemplo:

```
1     $rocks[0] = 'bedrock';
2     $rocks[1] = 'slate';
3     $rocks[2] = 'lava';
4     $rocks[3] = 'crushed rock';
5     $rocks[99] = 'schist';      # ahora hay 95 elementos undef
```

Algunas veces necesitamos encontrar el ultimo elemento indexado en el array. Para el array `rocks` que hemos definido en el ejemplo anterior, el ultimo índice del array es `$#rocks`. Este valor no es igual a la cantidad de elementos contenidos en el array, esto es porque hay un índice 0.

```
1     $end = $#rocks;           # 99
2     $numero_de_elementos = $end + 1;
3     $rocks[$#rocks] = 'hard rock';    # El último elemento.
```

El uso del valor `$$name` como un índice, como vimos en el ejemplo anterior, ocurre con tanta frecuencia que Larry proporciono un acceso directo: índices negativos para el array contando desde el final del array. No valla a pensar con esto que puede ir mas allá del inicio del array. Si tienes tres elementos en el array, entonces los índices negativos válidos son: -1 (el último elemento), -2 (el elemento del medio), y -3 (el primer elemento). En la práctica parece que no se suele usar esto, con la excepción del índice -1.

```
1 $rocks[ -1 ] = 'hard rock'; # una forma fácil de hacer el ultimo ejemplo
2 $dead_rock = $rocks[-100]; # se obtiene 'bedrock'
3 $rocks[ -200 ] = 'crystal' # Error fatal !
```

3.3. Listas Literales

Un array (la forma de representar un valor de lista en el programa) es una lista de valores separados por coma encerrados entre paréntesis. Estos valores constituyen los elementos de la lista. Por ejemplo:

```
1 (1, 2, 3) # Una lista de tres valores 1, 2 y 3
2 (1, 2, 3,) # La misma lista de tres valores (la ultima coma se ignora)
3 ("prueba", 4.5) # dos valores, "prueba" y 4.5
4 ( ) # una lista vacía, cero elementos
5 (1..100) # Una lista de 100 enteros.
```

El ultimo ejemplo, aparece el operador de rango `...`. Este operador crea una lista de valores contando desde el escalar que se encuentra a la izquierda hasta el escalar de la derecha, uno a uno. Por ejemplo:

```
1 (1..5) # lo mismo que (1, 2, 3, 4, 5)
2 (1.7..5.7) # la misma cosa, pero se truncan los valores.
3 (5..1) # Una lista vacía, solo se cuenta hacia arriba.
4 (0, 2..6, 10, 12) # Esto es (0, 2, 3, 4, 5, 6, 10, 12)
5 ($m..$n) # rango determinado por los valores de $m y $n
6 (0..$#rocks) # Usando el indice del array rocks.
```

Como puede ver en los últimos dos ejemplos, los elementos de una lista literal no necesariamente son constantes, pueden ser expresiones que van a ser evaluadas cada vez que el literal sea usado. Por ejemplo:

```
1 ($m, 17) # Dos valores, el valor correspondiente de $m y 7
2 ($m+$om, $p+$q) # Dos valores
```

Por supuesto, la lista puede contener cualquier valor escalar, como esta lista de cadenas:

```
1 ("walter", "lilibeth", "juan", "juan", "jose", "carlos")
```

3.4. El atajo qw

Resulta que las listas de palabras simples son necesarias con frecuencia en programas en Perl. El atajo `qw` hace que sea mas fácil generarlas sin tener que escribir un montón de comillas adicionales:

```
1 qw( walter lilibeth juan juan jose carlos );
```

`qw` significa “palabras citadas” del ingles “quoted words”. Perl trata esto como una cadena entre comillas simples en el contexto escalar, de manera que no podrá usar `\n` o `$prueba` en una lista `qw` como si estuviera entre comillas dobles.

Los espacios en blanco (caracteres como espacios, tabs, y nuevas lineas) son descartados, y todo lo que queda se convierte en una lista de elementos. Tomando en cuenta que el espacio en blanco se descarta, podemos escribir la lista de la siguiente forma:

```
1      qw( walter
2          lilibeth
3          juan
4          juan
5          jose
6          carlos
7      );
```

El ejemplo anterior usa paréntesis como delimitador, pero Perl actualmente permite que uses cualquier carácter de puntuación como delimitador. Algunos ejemplos mas comunes son:

```
1      qw( walter lilibeth juan juan jose carlos );
2      qw! walter lilibeth juan juan jose carlos !;
3      qw/ walter lilibeth juan juan jose carlos /;
4      qw# walter lilibeth juan juan jose carlos #;
5      qw{ walter lilibeth juan juan jose carlos };
6      qw[ walter lilibeth juan juan jose carlos ];
7      qw< walter lilibeth juan juan jose carlos >;
```

Si necesita incluir el delimitador usado como parte de una cadena en uno de los elementos, probablemente escogió el delimitador equivocado, pero incluso podría hacerlo, escapando el delimitador con un `\`. Ejemplo:

```
1      qw! yahoo\! google ask msn ! # include yahoo! as an element
```

Aunque el lema de perl es “Siempre Hay Mas De Una Forma De Hacer Las Cosas”, se preguntara, para que necesito todas estas formas de citar. Bueno, mas adelante que hay otras formas de citar en Perl que son útiles en casos especiales. Por ejemplo, ahora mismo podría necesitar hacer una lista de nombres de archivos Unix, entonces podría hacerlo así:

```
1      qw{
2          /usr/dict/words
3          /home/elsanto/.vimrc
4      }
```

3.5. Asignación de listas

De la misma forma que asignamos escalares a variables, podemos asignar valores de listas a variables. Por ejemplo:

```
1      ($walter, $lilibeth, $juan) = (25, 26, 30);
```

Las tres variables en la lista de la izquierda, reciben nuevos valores, como si los asignara individualmente. Dado que la lista se constituye antes del inicio de la asignación. Esto hace que sea fácil intercambiar los valores de dos variables en Perl:


```

1      ($walter, $lilibeth) = ($lilibeth, $walter);
2      ($betty[0], $betty[1]) = ($betty[1], $betty[0]);

```

Pero que ocurre si el número de variables (a la izquierda del signo de =) no es igual al número de valores (a la derecha del signo =) en una asignación de lista. Los valores extra son simplemente ignorados. De igual manera si ocurre lo contrario, las variables adicionales obtienen el valor `undef`.

```

1      ($fred, $barney) = qw< flintstone rubble slate granite >;
2      ($wilma, $dino) = qw[flintstone];          # $dino obtiene undef.

```

Ahora que puedes asignar listas, puedes crear un array de cadenas con una línea de código como esta:

```

1      ($rocks[0], $rocks[1], $rocks[2]) = qw/talc mica feldspar/;

```

Podemos referirnos a toda una matriz a través de una notación simple que posee Perl. Solo tiene que usar una arroba (@) antes del nombre del array (y sin corchetes de índices) para referirse a toda la matriz. Este símil funciona a ambos lados del operador de asignación.

```

1      @rocks = qw/ bedrock slate lava /;
2      @tiny  = ( );                      # una lista vacía
3      @giant = 1..1e5;                   # una lista de 100.000 elementos
4      @stuff = (@giant, undef, @giant);  # una lista de 200.001 elementos

```

`@tiny` es una lista conformada por cero elementos. (Esto en particular no pone `undef` dentro de la lista, pero lo podemos hacer de manera explícita como se puede ver con la lista `@stuff`).

El valor de una lista nueva a la que no se le ha asignado nada es `()` de lista vacía. Así como los valores escalares nacen con el valor `undef` una lista nace con el valor `()` (lista vacía).

Larry escogió el signo @ porque el lee `$scalar` (scalar) y `@array` (array), es una buena regla nemotécnica para recordar.

3.6. Los operadores push y pop

Usted puede agregar nuevos elementos al final de un array simplemente guardando estos elementos con un índice nuevo más grande. Pero los verdaderos programadores de Perl no usan índices ². En las siguientes secciones explicaremos como trabajar con arrays sin usar índices.

Un uso común para un array es guardar información, en donde los nuevos valores se añaden y se eliminan por la parte derecha de la lista (Este es el final de la lista que contiene el último elemento, con el índice más alto). Estas operaciones ocurren con la suficiente frecuencia como para tener sus propios operadores especiales.

El operador `pop` saca el último elemento de la lista y lo retorna. Ejemplo:

```

1      @array = 5..9;
2      $fred  = pop(@array); # $fred = 9, @array = (5, 6, 7, 8)
3      $barney = pop @array; # $barney = 8, @array = (5, 6, 7)
4      pop @array;          # @array = (5, 6) (El 7 fue descartado)

```

²Esto por su puesto es una broma. Los índices en Perl no son una fortaleza del lenguaje. Si usas `pop`, `push` y operadores similares en lugar de usar índices, tu código va a ser generalmente más rápido

El último ejemplo usamos `pop` en un “contexto vacío”, que es una manera elegante de decir que el valor de retorno no va a ninguna parte. No hay nada malo con el uso de `pop` de esta manera.

Si la matriz esta vacía, `pop` no hace nada y devuelve `undef`.

Seguro noto que `pop` puede usarse sin los paréntesis. Esta es una regla general en Perl, siempre y cuando no cambie el sentido de la expresión, se pueden remover los paréntesis.³

El operador contrario es `push` que agrega elementos o una lista de elementos al final de un array. Por ejemplo:

```
1      push(@array, 0);           # @array ahora tiene (5, 6, 0)
2      push @array, 8;           # @array ahora tiene (5, 6, 0, 8)
3      push @array, 1..10;       # @array ahora tiene 10 elementos mas.
4      @others = qw/ 9 0 2 1 0/;
5      push @array, @others;     # @array ahora tiene 5 elementos nuevos (19)
```

Note que el primer argumento de `push` o el único argumento que `pop` requiere debe ser una variable array. Hacer `push` y `pop` de listas literales no tiene ningún sentido.

3.7. Los operadores `shift` y `unshift`

Los operadores `push` y `pop` hacen cosas al final del array (o al lado derecho del array). De manera similar, `unshift` y `shift` realizan las operaciones correspondientes pero al principio del array (o al lado izquierdo del array). Aquí hay algunos ejemplos:

```
1      @array = qw# walter lilibeth juan #;
2      $m = shift(@array);       # $m = "dino", @array = ("fred", "barney")
3      $n = shift @array;        # $n = "fred", @array = ("barney")
4      shift @array;             # @array = (), vacío
5      $o = shift @array;        # $o = undef, @array esta vacío
6      unshift(@array, 5);       # @array ahora tiene un elemento en la lista
7      unshift @array, 4;        # @array = (4, 5)
8      @others = 1..3;
9      unshift @array, @others;  # @array = (1, 2, 3, 4, 5)
```

3.8. Interpolando Arrays en Cadenas

Así como los escalares, los valores array pueden ser interpolados en una cadena de dobles comillas. Los elementos de un array son automáticamente separados por espacios (Espacio en blanco es el separador por defecto contenido como valor en la variable especial `$'`).

```
1      @rocks = qw{ flintstone slate rubble };
2      print "quartz @rocks limestone\n"; # imprime 5 rocas separadas por blanco
```

No se agregan espacios en blanco antes o después de la interpolación, si usted quiere esto, tendrá que hacerlo a mano.

³Un estudiante avanzado, va a reconocer que esto es una tautología

```

1     print "Three rocks are: @rocks.\n";
2     print "There's nothing in the parens (@empty) here.\n";

```

Si de casualidad olvida que los array se interpolan entre dobles comillas, se va a llevar una sorpresa cuando intente colocar un correo electrónico en una cadena.

```

1     $email = "walter@covetel.com.ve;    # MAL! Intentara interpolar @covetel

```

Para poder hacer esto, deberá usar comillas simples o escapar el carácter @.

```

1     $email = "walter\@covetel.com.ve"; # Correcto
2     $email = 'walter@covetel.com.ve'   # otra forma de hacerlo.

```

3.9. La estructura de control foreach

Regularmente es necesario procesar una lista completa, por lo que Perl proporciona una estructura de control que hace justamente esto. El ciclo `foreach` pasa por cada valor de la lista, realizando una iteración por cada valor. Por ejemplo:

```

1     foreach $rock (qw/ bedrock slate lava /) {
2         print "One rock is $rock.\n"; # Prints names of three rocks
3     }

```

La variable de control (`$rock` en este ejemplo) toma un nuevo valor de la lista por cada iteración. En la primera corrida es “bedrock; y en la tercera corrida es “lava”.

La variable de control no es una copia del elemento de la lista. Es el elemento en si. Lo que significa que si modificas la variable de control dentro del ciclo, vas a modificar el elemento en si. Por ejemplo:

```

1     @rocks = qw/ bedrock slate lava /;
2     foreach $rock (@rocks) {
3         $rock = "\t$rock";
4         $rock .= "\n";
5     }
6     print "The rocks are:\n", @rocks;
7

```

¿Cual es el valor de la variable de control cuando el ciclo ha terminado?. Es el mismo que tenia para cuando el ciclo comenzó. Perl automáticamente guarda y restaura el valor de la variable de control de un ciclo `foreach`. Esto significa que no hay que preocuparse por usar una variable que este en uso en otro lado del programa.

3.10. La variable mágica de Perl: `$_`

Si omite la variable de control del ciclo, Perl usa la variable mágica, `$_`. Esta es (usualmente) una variable escalar mas, solo que con un nombre no usual. Por ejemplo:

```

1     foreach (1..10){ # Usa por defecto $_
2         print "La cuenta va por $_\n";
3     }

```

Aunque esta no es la única cosa mágica que hace Perl, es el hechizo mas común.

Existen otros casos en los que Perl va a usar la variable mágica `$_`, uno de los casos en los que esto ocurre es con `print`, va a imprimir `$_` si no le damos otro argumento.

```
1     $_ = "Hola";
2     print;           #Imprime por defecto $_
```

3.11. El operador reverse

El operador `reverse` toma una lista de valores (que pueden venir de un array) y devuelve la lista en el orden opuesto. Entonces si no le gusta esto de que el operador de rango `..` solo cuenta hacia arriba, esta es una forma de arreglar esto:

```
1     @lista = 6..10;
2     @lista2 = reverse(@lista);
3     @lista3 = reverse 6..10;
4     @lista = reverse @lista;
```

La última línea del ejemplo es importante, porque usa `@lista` dos veces. Esto es debido a que los operadores de lista tienen una fuerte precedencia a la derecha y una poca precedencia a la izquierda, y en la tabla de precedencias, los operadores de lista están de primeros junto a los paréntesis.

Recuerde que `reverse` devuelve la lista en el orden inverso, no afecta a la lista argumento, si el valor devuelto no se asigna a nada, es una operación inútil.

```
1     reverse @lista;           # MAL! no hace nada.
2     @lista = reverse @lista;  # Bien!
```

3.12. El operador sort

El operador `sort` toma una lista de valores (que pueden ser un array) y los ordena usando la ordenación interna de caracteres. Para cadenas ASCII, va a usar un orden ASCIIbetico. Como ya debe saber, ASCII es un lugar extraño donde todas las letras mayúsculas van delante de todas las letras minúsculas, donde los números vienen antes de las letras, y los signos de puntuación, esos signos que están aquí allá y en todas partes. Pero el orden ASCII es el orden por defecto, en otro capítulo vamos a ver como ordenar de otras maneras.

```
1     @rocks      = qw / bedrock slate rubble granite /;
2     @sorted     = sort(@rocks);           # bedrock, granite, rubble, slate
3     @back       = reverse sort @rocks;    # va de slate a bedrock
4     @rocks      = sort @rocks;            # ordena y asigna
5     @numbers    = sort 97..102;          # 100, 101, 102, 97, 98, 99
```

3.13. Contexto Escalar y Contexto de Lista

Esta es la sección mas importante de este capítulo. En efecto, esta es la sección mas importante del curso. No es una exageración afirmar que el desempeño de su carrera en el uso de Perl va a depender de comprender de manera correcta esta sección.

No significa que esto es difícil de entender. Es una idea simple: Una expresión dada puede significar cosas distintas dependiendo de en que lugar aparece. Esto no debería ser

nada nuevo para usted, pasa todo el tiempo en lenguajes naturales. Por ejemplo en el contexto gocho hay expresiones que son contrarias al contexto caraqueño.

El contexto se refiere al lugar en donde se encuentra la expresión. Es la manera en que Perl interpreta las expresiones. Siempre se espera un valor de lista o un valor escalar. Ejemplo:

```
1      42 + algo # algo debe ser un escalar
2      sort algo # algo debe ser una lista.
```

Incluso si algo es exactamente la misma secuencia de caracteres, en un caso puede ser un simple valor escalar, mientras que en otro caso, puede ser una lista. Las expresiones en Perl siempre devuelven el valor apropiado para el contexto. Por ejemplo, tome un array, en un contexto de lista, este devuelve una lista de elementos. Pero en un contexto escalar, devuelve el numero de elementos que están contenidos en el array. Por ejemplo:

```
1      @people = qw ( walter lilibeth juan );
2      @sorted = sort @people;      # Contexto de lista: walter, lilibeth, juan
3      $number = 42 + @people;      # Contexto escalar: 42 + 3
```

Incluso en una asignación simple, ocurren diferentes contextos:

```
1      @list = @people;      # una lista de 3 personas
2      $n = @people;          # el número 3.
```

3.13.1. Usando Expresiones de listas en contexto escalar

Hay varias expresiones que deberían usarse para producir una lista. Si usa una de estas expresiones en un contexto escalar, ¿Qué cree que obtendrá?. Para saber esto debe ver que es lo que piensa el autor de la expresión. Usualmente esta persona es Larry, y usualmente en la documentación obtendrá la información completa. En efecto, parte del aprendizaje de Perl es aprender como es que Larry piensa.

Hay algunas expresiones que en contexto escalar no devuelven nada. Por ejemplo, **sort** en contexto escalar devuelve **undef**.

Otro ejemplo es **reverse**. En contexto de lista, devuelve la lista inversa. En contexto escalar, devuelve la cadena reversa (o el resultado reverso concatenando todas las cadenas). Ejemplo:

```
1      @backwards = reverse qw/ yabba dabba doo /; # doo, dabba, yabba
2      $backwards = reverse qw/ yabba dabba doo /; # oodabbadabbay
```

Algunos contextos mas comunes son:

```
1      $prueba = algo # contexto escalar
2      @gente = algo # contexto de lista
3      ($nombre, $apellido) = algo # contexto de lista
4      ($algo) = algo # contexto de lista aún.
```

Aquí tenemos algunas otras expresiones que resultan en contexto escalar:

```
1      $fred = something;
2      $fred[3] = something;
3      123 + something something + 654
4      if (something) { ... }
5      while (something) { ... }
6      $fred[something] = something;
```

Y aquí tenemos algunas expresiones que proveen un contexto de lista:

```
1    @fred = something;
2    ($fred, $barney) = something;
3    ($fred) = something;
4    push @fred, something;
5    foreach $fred (something) { ... }
6    sort something
7    reverse something
8    print something
```

3.13.2. Usando Expresiones escalares en contexto de lista

En este sentido es mas claro, si una expresión no tiene normalmente un valor de lista, el valor escalar es automáticamente promovido a ser un elemento de la lista. Por ejemplo:

```
1    @fred = 6 * 7; # Un elemento (42)
2    @barney = "hello" . ' ' . "world";
```

Hay algunos inconvenientes posibles:

```
1    @wilma = undef; # OOPS!
2    @betty = ( ); # la forma correcta de declarar una lista vacía.
```

`undef` es un valor escalar, asignar `undef` a un array no limpia el array. La manera correcta de limpiar un array es asignándole una lista vacía.

3.14. Forzar el contexto escalar

En ocasiones es necesario forzar el contexto escalar, en donde Perl espera una lista. En este caso, se puede usar la función falsa `scalar`. Esta no es una función verdadera, ya que lo único que hace es decirle a Perl que provea un contexto escalar.

```
1    @rocks = qw( talc quartz jade obsidian );
2    print "How many rocks do you have?\n";
3    print "I have ", @rocks, " rocks!\n";          #MAL!
4    print "I have ", scalar @rocks, " rocks!\n";    #BIEN!
```

De manera contraria, no hay una función que force el contexto de lista, confíe en nosotros, no la va a necesitar.

3.15. STDIN en contexto de lista

En un capítulo anterior vimos que `STDIN` en un contexto escalar devuelve la ultima línea leída de la entrada estándar. En contexto de lista, este operador devuelve todas las líneas desde el principio hasta el final del archivo. Cada línea es retornada como un elemento separado de la lista. Por ejemplo:

```
1    @lines = <STDIN>; # lee la entrada estándar en el contexto de lista
```

Cuando la entrada estándar es un archivo, esto va a leer el archivo completo hasta el final. Pero cuando se lee de la entrada estándar en Unix, debe usar `Control + D`.

3.16. Ejercicios

1. Escriba un programa que lea una lista de caracteres separados por lineas e imprima esta lista en modo inverso, si la entrada proviene del teclado, se necesita una señal al para la finalización de la entrada, en Unix es CTRL-D.
2. Escriba un programa que lea una lista de números separados por lineas y por cada número imprima el nombre correspondiente a la lista mostrada a continuación. fred betty barney dino wilma pebbles bamm-bamm. Por ejemplo, si la entrada es 1, 2, 4, la salida debe mostrar lo siguiente, fred, betty, dino.
3. Escriba un programa que lea una lista de caracteres separados por lineas e imprima esta lista en orden alfabético, por ejemplo si la entrada fue fred, barney, wilma, betty, la salida debe ser, barney, betty, fred, wilma

4 Subrutinas

Hasta el momento hemos visto y usado algunas funciones internas de Perl, como `chomp`, `reverse`, `print`, y otras mas. Al igual que otros lenguajes, Perl tiene la habilidad de crear **subrutinas**, que son funciones definidas por el usuario.¹ Pero una subrutina es siempre definida por un usuario, mientras que una función puede o no serlo. Esto es, la palabra *función* puede ser usada como sinónimo para *subrutina*, pero es una función que el programador puede definir, no las funciones internas de Perl.

El nombre de una subrutina es otro identificador de Perl (letras, dígitos y pisos bajos, pero este no puede iniciar con un dígito.) con un signo `&` (ampersand) a veces opcional adelante. Existe una regla acerca de cuando podemos omitir el signo `&` y cuando no, al final del capítulo vamos a ver esta regla. Por ahora vamos a usar el signo `&` adelante siempre, esto no puede olvidarse.

El nombre de una subrutina viene de un espacio de nombres separado, por lo que Perl no va a confundirse si usted tiene una subrutina llamada `&fred` y una variable escalar llamada `$fred` en el mismo programa, aunque no hay una razón para hacer esto en circunstancias normales.

4.1. Definiendo una Subrutina

Para definir su propia subrutina, use la palabra reservada `sub`, el nombre de la subrutina (sin el signo `&`), luego el bloque de código indentado (entre llaves)², va a conformar el cuerpo de la subrutina, quedaría algo como esto:

```
1  sub marino {
2      $n += 1;    # Variable global $n
3      print "Hola, marinera número $n!\n";
4  }
```

Las subrutinas se pueden definir en cualquier lugar del programa, pero los programadores que vienen de lenguajes como C o Pascal, les gusta colocar las subrutinas al inicio del archivo. Otros prefieren poner las subrutinas al final del archivo, entonces la parte principal del programa queda al principio. Esto depende de usted. En cualquier caso, normalmente no va a necesitar ningún tipo de declaración previa³

¹En Perl, no se hace la distinción que se hacía en Pascal, entre funciones que retornan un valor y procedimientos, que no retornan un valor.

²Bueno, los puristas admitimos que las llaves son parte del bloque. Y Perl no requiere la indentación del bloque, pero los nuevos programadores son estilizados

³A menos que tu subrutina valla a ser particularmente compleja y declare un prototipo, que indica como el compilador va a parsear e interpretar la invocación de los argumentos. Esto no es común, vea la página del manual `perlsub` para mas información.

La definición de la subrutina es global; sin hechicería no hay subrutina privadas. Si tiene dos subrutinas definidas con el mismo nombre, la última subrutina sobre escribe la primera. Esto es generalmente considerado como una mala práctica, o una señal de que el programador de mantenimiento esta confundido.

Como pudo notarlo en el ejemplo previo, puedes usar variables globales dentro de el cuerpo de una subrutina. De hecho, todas las variables que hemos visto hasta ahora son globales; esto significa, que podemos acceder a ellas desde cualquier parte del programa. Esto horroriza a los puristas del lenguaje, pero no se preocupe, el equipo de desarrollo de Perl formo una turba iracunda con antorchas y corrió fuera de la ciudad hace años. Mas adelante veremos como hacer variables privadas en “Variables Privadas en Subrutinas”.

4.2. Llamar una Subrutina

Puede invocar o llamar una subrutina desde cualquier expresión usando el nombre de la subrutina (con el signo `&`) ⁴.

```
1      &marino;      # dice: Hola, marinera numero 1!
2      &marino;      # dice: Hola, marinera numero 2!
3      &marino;      # dice: Hola, marinera numero 3!
4      &marino;      # dice: Hola, marinera numero 4!
```

4.3. Retornar Valores

Las subrutinas siempre son invocadas como parte de una expresión, incluso si el resultado de esa expresión no se usa para nada. Cuando llamamos anterior mente a la subrutina `&marino`, calculamos un valor en la expresión contenido en la llamada, pero el resultado simplemente se envió a la pantalla.

Muchas veces que se llama a una subrutina, se quiere hacer algo con el resultado. Entonces debe prestar atención al *valor de retorno* de una subrutina. Todas las subrutinas en Perl tienen un valor de retorno. No hay distinción entre las que retornan valores y las que no. Sin embargo no todas las subrutinas en Perl tiene un valor de retorno **útil**.

Puesto que en Perl, las subrutinas siempre deben devolver un valor, sería un poco inútil tener que declarar una sintaxis especial de tipo `return` para la mayoría de los casos. Entonces Larry hizo algo simple. Cualquier calculo que se encuentre de último en una subrutina es automáticamente el valor de retorno.

Por ejemplo, vamos a definir esta subrutina:

```
1      sub sum_of_fred_and_barney {
2          print "Hey, has llamado a la subrutina sum_of_fred_and_barney !\n";
3          $fred + $barney; # Este es el valor de retorno.
4      }
```

La última expresión evaluada en el cuerpo de la subrutina, es la suma de `$fred` y `$barney`, entonces la suma de `$fred` y `$barney` va a ser el valor de retorno. Veamos esto en acción:

⁴Y frecuentemente un par de paréntesis, incluso si están vacíos

```

1    $fred = 3;
2    $barney = 4;
3    $wilma = &sum_of_fred_and_barney; # $wilma obtiene 7
4    print "\$wilma es $wilma.\n";
5    $betty = 3 * &sum_of_fred_and_barney; # $betty obtiene 21
6    print "\$betty es $betty.\n";

```

Ahora supongamos que agrega una linea mas a la subrutina, por ejemplo:

```

1    sub sum_of_fred_and_barney {
2        print "Hey, has llamado a la subrutina sum_of_fred_and_barney !\n";
3        $fred + $barney; # Este es el valor de retorno.
4        print "Hey, estoy retornando un valor ahora!\n";
5    }

```

En este ejemplo, la última expresión evaluada no es una suma; es la sentencia print. Esta retorna normalmente 1, que significa “printing was successful”, pero no es el valor de retorno que esperaba. Entonces, debe ser cuidadoso cuando agrega código adicional a una subrutina, puesto que la ultima expresión evaluada va a ser el valor de retorno.

Tome en cuenta que no es la última linea de la subrutina, es la última expresión evaluada. Por ejemplo, esta subrutina devuelve el valor mas grande, entre \$fred y \$barney:

```

1    sub larger_of_fred_or_barney {
2        if ($fred > $barney) {
3            $fred;
4        } else {
5            $barney;
6        }
7    }

```

La última expresión evaluada esta entre \$fred o \$barney, entonces el valor de una de estas variables va a ser el valor de retorno. No podemos saber cual va a ser la variable de retorno hasta que no veamos que guarda cada variable en tiempo de ejecución.

Estos son ejemplos triviales, se pone mejor cuando podemos pasar valores a la subrutina en lugar de depender de variables globales.

4.4. Argumentos

La subrutina llamada `larger_of_fred_or_barney` puede ser mas útil si no estuviera forzada a usar variables globales.

Perl tiene argumentos para subrutinas. Para pasar una lista de argumentos a una subrutina, simplemente se coloca la expresión de lista en paréntesis, después de la llamada a la subrutina, por ejemplo:

```

1    $n = &max(10, 15); # Esta llamada tiene dos paréntesis.

```

Una vez que lista es enviada a la subrutina, va a estar disponible en la subrutina para hacer lo que sea necesario. Por supuesto, usted tiene que guardar esta lista en algún lugar, Perl automáticamente la lista de parámetros (este es otro nombre para la lista de argumentos) en una variable de tipo array especial llamada `@_` durante el tiempo de

existencia de la subrutina. La subrutina puede acceder a esta variable para determinar o el número de argumentos y el valor de los argumentos.

Esto significa que el primer parámetro de la subrutina es guardado en `$_[0]`, el segundo es guardado en `$_[1]` y así sucesivamente. Pero, he aquí una nota importante, estas variables no tienen nada que ver con la variable `$_`, es solo que la lista de parámetros debe ser almacenada en una variable array para que la subrutina pueda utilizarlos, y Perl llama a esta variable `@_`.

Ahora usted puede escribir la subrutina `&max` para que se parezca un poco a la subrutina `&larger_of_fred_or_barney`, pero en lugar de usar `$fred` usted puede usar el primer parámetro de la subrutina (`$_[0]`), y en lugar de usar `$barney`, puede usar el segundo parámetro de la subrutina (`$_[1]`). Y así puede terminar un código que se vea como esto:

```

1      sub max {
2          # Compare esto con &larger_of_fred_or_barney
3          if ($_[0] > $_[1]) {
4              $_[0];
5          } else {
6              $_[1];
7          }
8      }
```

Bien, como ya dijimos, puede hacer esto. Pero es un poco feo con todos esos subíndices, y es poco fácil de escribir, revisar, y depurar. En un momento veremos una mejor forma de hacerlo.

Hay otro problema con esta subrutina, los parámetros adicionales son ignorados, puesto que la subrutina nunca mira en `$_[2]`, a Perl no le importa si allí hay algo o no, y la ausencia de un parámetro también es ignorada, simplemente obtendrá `undef` si mira mas allá del final del array `@_`. En este mismo capítulo veremos como hacer esta subrutina mas eficiente.

La variable `@_` es privada para cada subrutina; <A menos que se coloque un signo `&` delante del nombre de la subrutina en la variable y no se indique ningún argumento o paréntesis, en cuyo caso `@_` es heredada desde el contexto de la llamada a la subrutina. Generalmente es una mala idea, pero en ocasiones es útil> si hay una variable global `@_`, su valor es conservado antes de la invocación de la subrutina, y su valor es restaurado cuando la llamada termina. Esto significa que una subrutina puede pasar valores a otra subrutina sin miedo a perder los valores contenidos en `@_`. Si una subrutina se llama recursivamente, cada invocación obtiene un nuevo `@_`, entonces `@_` es siempre la lista de parámetros para la subrutina.

4.5. Variables privadas en Subrutinas

Por defecto, todas las variables en Perl son variables globales; lo que significa que son accesibles desde cualquier parte del programa. Pero usted puede crear variables privadas llamadas variables léxicas, en cualquier momento usando el operador `my`. Por ejemplo:

```

1      sub max {
2          my ($m, $n);      # Nuevas variables privadas para este bloque
```

```

3      ($m, $n) = @_; # obtener los parámetros
4      if ($m > $n) { $m } else { $n }
5  }

```

Estas variables son privadas (o de ámbito local) al bloque que las contiene, ningún otro código puede acceder o modificar estas variables privadas, por accidente o por diseño ⁵.

También es importante señalar que, en el bloque `if`, no fue necesario un punto y coma después de la expresión del valor de retorno. Aunque Perl permite omitir el último punto y coma de un bloque, en la práctica tu puedes omitirlo solo cuando el código es tan simple que puedas escribirlo en un bloque de una sola línea.

La subrutina en el ejemplo previo puede ser mas simple aún. Debe haber notado que la lista `($m, $n)` fue escrita dos veces, el operador `my` puede ser aplicado a una lista de variables encerradas en paréntesis, entonces se pueden combinar las dos primeras sentencias de la subrutina de la siguiente manera:

```

1      my($m, $n) = @_; # nombre de los parámetros de la subrutina

```

Una sola sentencia crea las variables privadas y define sus valores. Casi todas las subrutinas van a comenzar con una línea muy parecida a esta, nombres entre paréntesis. Cuando vea esta línea, va a saber que la subrutina espera dos parámetros escalares, que vas a llamar `$m` y `$n` dentro de la subrutina.

4.6. Longitud de la lista de parámetros

En el mundo real, las subrutinas no poseen un valor arbitrario para la longitud de la lista de parámetros. Esto es porque en Perl no hay límites innecesarios. Es agradable que Perl sea tan flexible, pero puede haber problemas cuando se llama a la subrutina con una cantidad diferente de parámetros que espera el autor.

Por su puesto, la subrutina puede fácilmente probar si el número de argumentos es correcto examinando el array `@_`. Por ejemplo, podemos escribir la función `&max` de cierta forma para que revise la lista de argumentos:

```

1      sub max {
2          if (@_ != 2){
3              print "WARNING! &max debería tener exactamente dos argumentos\n";
4          }
5          # ...
6      }

```

Perl en el mundo real, esto casi no se usa, es mejor hacer que la subrutina se adapte a los parámetros.

4.6.1. Subrutina `&max` bien escrita

```

1      $maximo = &max(4, 5, 10, 4, 6);
2
3      sub max {

```

⁵Programadores mas avanzados puede hacer una variable de ámbito local accesible por referencia desde fuera de su ámbito, pero nunca por el nombre de la variable

```

4      my ($max_so_far) = shift @_;    # tomo el primer valor como el mayor
5      foreach (@_) {
6          if ($_ > $max_so_far) {
7              $max_so_far = $_;
8          }
9      }
10     $max_so_far;
11 }

```

4.7. Notas sobre Variables Léxicas (my)

Estas variables léxicas pueden usarse en cualquier bloque, no necesariamente en una subrutina. Por ejemplo, pueden usarse en un bloque de un `if`, `while`, o `foreach`:

```

1      foreach (1..10){
2          my ($cuadrado) = $_ * $_; # variable privada en este ciclo.
3          print "El cuadrado de $_ es $square.\n";
4      }

```

La variable `$cuadrado` es privada al bloque de código; en este caso, al bloque de código del ciclo `foreach`. Si no se encuentra encerrada en un bloque, la variable es privada para el archivo completo. El concepto importante es que el ámbito de una variable léxica es limitado al pequeño bloque de código que encierre la variable. Esto es una gran virtud para la mantenibilidad del código. Si hay un valor erróneo en `$cuadrado`, la culpabilidad va a estar limitada a una pequeña porción del código.

Debe haber notado, que el operador `my` tampoco altera el contexto de la asignación:

```

1      my ($num)    = @_; # contexto de lista.
2      my $num      = @_; # contexto escalar.

```

En la primera línea, `$num` obtiene el primer parámetro, en la segunda línea `$num` obtiene el número de parámetros en contexto escalar.

Vale la pena recordar, que el uso de `my` sin paréntesis solo va a declarar una variable léxica simple:

```

1      my $fred, $barney;    # MAL! Falla al declarar $barney
2      my ($fred, $barney);  # declara ambas variables.

```

También puede usar `my` para crear un nuevo y privado array:

```

1      my @phone_number;

```

4.8. Usando el Pragma Strict

Perl tiende a ser un lenguaje muy permisivo. Pero puede ser que quieras que Perl imponga un poco de disciplina; esto se puede hacer con el pragma `strict`.

Un pragma es una sugerencia al compilador, que le dice algo sobre el código. En este caso, el uso del pragma `strict` le dice al compilador interno de Perl que debe forzar el uso algunas buenas reglas de programación para el resto de este bloque o archivo.

¿ Porqué esto es importante ?, Bueno imagine que esta creando un programa y usted escribe una línea como esta:

```
1      $bamm_bamm = 3; # Perl crea esta variable automáticamente
```

Ahora, luego de un rato, escribe un ciclo while. Después de que la línea anterior no sea visible en la pantalla, usted tipea una línea para incrementar la variable.

```
1      $bammbamm += 1;
```

Puesto que perl ve un nuevo nombre de variable (el guión bajo es significativo) crea una nueva variable, e incrementa su valor en uno. Si eres afortunado e inteligente, activaste las advertencias (warnings), y Perl te va a decir que has usado una o varias variables globales una sola vez en tu programa. Pero si eres simplemente inteligente, vas a usar cada variable mas de una vez, y Perl no va a advertirte nada.

Para decirle a Perl que quieres ser mas restrictivo, coloque `use strict` al principio de su programa.

```
1      use strict;      # Forzar algunas buenas reglas de programación.
```

Ahora entre otras restricciones, Perl va a insistir que declares cada variable nueva, usualmente con `my`.

```
1      my $bamm_$bamm = 3;      # Nueva variable léxica.
```

Ahora, Perl va a notar que no hay una variable `$bammbamm` declarada, entonces tu error va ser automáticamente atrapado en tiempo de compilación.

```
1      $bammbamm += 1;      # No such variable: Compile time fatal error
```

Para aprender mas sobre las restricciones que aplica el pragma `strict`, le recomendamos ver la documentación oficial. La documentación de los pragmas se encuentra en un archivo bajo el nombre del pragma, entonces podemos usar `perldoc strict`.

La mayoría de la gente cree que si un programa es mas largo que la pantalla, generalmente necesita `use strict`. En esto estamos de acuerdo.

4.9. El operador return

El operador `return` inmediatamente retorna un valor de desde una subrutina:

```
1      my @names = qw/ fred barney betty dino wilma pebbles bamm-bamm /;
2      my $result = &which_element_is("dino", @names);
3
4      sub which_element_is {
5          my($what, @array) = @_;
6          foreach (0..$#array) {
7              if ($what eq $array[$_]) {
8                  return $_;
9              }
10         }
11         -1;
12     }
```

Esta subrutina es usada para encontrar el índice de “dino” en el array `@names`.

A algunos programadores les gusta usar `return` siempre, como una forma de documentar que este es el valor de retorno. Por ejemplo, se puede usar `return` cuando el valor de retorno no es la ultima expresión evaluada de la subrutina.

4.10. Omitir el signo &

Como lo prometimos, vamos a darle la regla para cuando una llamada a subrutina puede omitir el signo &. Si el compilador ve una definición de la subrutina antes de la llamada, o si Perl puede decidir que la sintaxis es una llamada a subrutina, la subrutina puede ser llamada sin un signo &. Como si fuera una función integrada de Perl. Pero hay una trampa escondida en esta norma, como veremos a continuación.

Esto significa que si Perl puede ver que es una llamada a una subrutina sin el signo &, por lo general va bien. Entonces si tienes una lista de parámetros entre paréntesis, entonces esto es una llamada a una función:

```
1 my $cards = shuffle(@deck_of_cards); # & no es necesario.
```

Si el compilador interno de perl ya ha visto la definición de la subrutina, entonces generalmente funciona bien. En este caso particular, puedes incluso omitir los paréntesis que abrazan la lista de argumentos.

```
1 sub division {
2     $_[0] / $_[1];
3 }
4
5 my $cociente = division 355, 113; # Usa &division
```

Esto funciona por la regla de que los paréntesis pueden ser omitidos.

Pero no ponga la declaración de la subrutina después de la llamada, o el compilador no sabrá de que se trata la llamada de `division`. Sin embargo esta no es la trampa, el problema en si es el siguiente: Si la subrutina tiene el mismo nombre que una función propia de Perl, usted *debe* usar el ampersand (&) para hacer la llamada a la subrutina. Al usar el ampersand, nos aseguramos de que estamos llamando a una subrutina; sin el; usted puede hacer la llamada solo si la subrutina no tiene el mismo nombre que una función propia de Perl. Por ejemplo:

```
1 sub chomp {
2     print "Mucho, mucho!\n";
3 }
4
5 &chomp; # El ampersand aquí no es opcional!
```

Sin el ampersand, estamos haciendo una llamada a la función `chomp` interna de Perl.

La regla general aquí es: Hasta que usted conozca los nombres de todas las funciones internas de Perl, siempre use ampersand en las llamadas a funciones.

En nuestro caso, podemos usar funciones en nuestra lengua materna (Español) para evitar caer en la trampa.

4.11. Retornando valores no escalares

Un escalar no es la única cosa que puede retornar una subrutina. Si llamas a tu subrutina en un contexto de lista ⁶, esta puede retornar una lista de valores.

⁶Puedes detectar en que contexto esta siendo llamada su subrutina, usando la función `wantarray`

Supongamos que quieres obtener un rango de números (como viene del operador de rango, ..), pero quieres habilitar el conteo hacia abajo. El operador de rango solo cuenta hacia arriba, pero es algo fácil de arreglar:

```

1      sub list_from_fred_to_barney {
2          if ($fred < $barney){
3              $fred..$barney;
4          } else {
5              reverse $barney..$fred;
6          }
7      }
8
9      $fred = 11;
10     $barney = 6;
11
12     @c = &list_from_fred_to_barney;
```

Finalmente, puedes retornar nada. Cuando usamos **return** sin argumentos vamos a retornar **undef** en contexto escalar o una lista vacía en contexto de lista. Esto puede ser útil para retornar errores de una subrutina.

4.12. Variables Privadas Persistentes

Con **my** podemos hacer variables privadas en una subrutina, sin embargo cada vez que llamamos a la subrutina va a re-definir los valores una vez mas. Con **state**, podemos tener variables privadas del ámbito de una subrutina pero Perl va a mantener sus valores entre las llamadas.

En el primer ejemplo de este capítulo, vimos una subrutina llamada *marino*, que incrementaba una variable:

```

1      sub marino {
2          $n += 1;    # Variable global $n
3          print "Hola, marinera numero $n!\n";
4      }
```

Ahora que debemos usar **strict**, el uso de la variable global **\$n** no esta permitido. No podemos hacer de **\$n** una variable léxica con **my** porque entonces no va a mantener el valor.

Declarando nuestra variable con **state** le decimos a Perl que retenga el valor de esta variable entre las llamadas a la subrutina y hace que la variable sea una variable privada de la subrutina.

```

1      use 5.010;
2
3      sub marino {
4          state $n = 0;    # privada, variable persistente $n
5          $n += 1;
6          print "Hola, marinera numero $n!\n";
7      }
```

Ahora, podemos obtener la misma salida mientras usamos `strict` sin usar variables globales. La primera vez que llamamos a la subrutina, Perl declara e inicializa `$n`, para las siguientes llamadas de la subrutina, Perl ignora la sentencia.

Podemos conservar el estado de cualquier variable, no es solo para los datos escalares. Aquí tenemos una subrutina que recuerda sus argumentos y provee una suma usando `state` en un array.

```

1      use 5.010;
2
3      running_sum(5, 6);
4      running_sum(1..3);
5      running_sum( 4 );
6
7      sub running_sum {
8          state $sum = 0;
9          state @numbers;
10
11         foreach my $number (@_){
12             push @numbers, $number;
13             $sum += $number;
14         }
15         say "La suma de (@numbers) es $sum";
16     }

```

La salida de este programa es:

Terminal

```

La suma de (5 6) es 11
La suma de (5 6 1 2 3) es 17
La suma de (5 6 1 2 3 4) es 21

```

Sin embargo, hay una ligera restricción en arrays y hashes como variables de estado. No podemos inicializarlas en contexto de lista. Por ejemplo:

```

1      state @array = qw(a b c);    # Error !

```

Esto nos dará un error que sugiere que podríamos usar esto en una versión futura de Perl:

Terminal

```

Initialization of state variables in list context currently forbidden

```

4.13. Ejercicios

1. Escriba una función llamada “total” que retorne el total de una lista de números. (Nota: la función no debe realizar algún tipo de I/O, esta debe simplemente procesar sus parámetros y retornar un valor). Complete el siguiente programa de ejemplo con la función “total”, el resultado de la sumatoria debe dar 25 para el primer grupo de números.

```

1      my @fred = qw{ 1 3 5 7 9 };
2      my $fred_total = total(@fred);
3      print "El total de \@fred es $fred_total.\n";
4      print "Ingrese algunos números separados por lineas: ";

```

```

5         my $user_total = total(<STDIN>);
6         print "El total para los números ingresados es: $user_total.\n";

```

2. Usando la función escrita en el ejercicio anterior, realice un programa que calcule la sumatoria de todos los números del 1 al 1000.

3. Escriba una función llamada `&above_average` que tome una lista de números y retorne solo aquellos números que estén por encima del promedio (Nota: escriba otra función que calcule el promedio de varios números dividiendo el total de la sumatoria de los números por el numero de items). Use su función para probar el siguiente programa:

```

1         my @fred = above_average(1..10);
2         print "\@fred es \@fred\n";
3         print "(Debe ser 6 7 8 9 10)\n";
4         my @barney = above_average(100, 1..10);
5         print "\@barney es \@barney\n";
6         print "(Debe ser solo 100)\n";

```

4. Escriba una función llamada “saludo”, que de la bienvenida a una persona por su nombre y que ademas diga el nombre de la ultima persona saludada, ejemplo:

```

1         saludo("Fred");
2         saludo("Barney");

```

Esto debe escribir lo siguiente:

```

1         Hola Fred! Eres el primero aquí!
2         Hola Barney! Fred también esta aquí!

```

5. Modifique el programa anterior para que imprima los nombres de todas las personas que han sido previamente saludadas.

```

1         saludo("Fred");
2         saludo("Barney");
3         saludo("Wilma");
4         saludo("Betty");

```

La salida debe ser como lo siguiente:

```

1         Hola Fred! Eres el primero aquí!
2         Hola Barney! He visto a: Fred
3         Hola Wilma! He visto a: Fred Barney
4         Hola Betty! He visto a: Fred Barney Wilma

```


5 Entrada y Salida

Ya hemos visto antes como hacer alguna entrada/salida, con el fin de hacer algunos de los ejercicios. Pero ahora vamos a aprender más acerca de las operaciones que cubren el 80 % de la E/S que se necesita para la mayoría de los programas. Por ahora, sólo piense en “entrada estándar” como “el teclado”, y “salida estándar” como “la pantalla”.

5.1. Entrada desde la Entrada Estándar

Leer el flujo de datos que viene desde la entrada estándar es fácil. Ya hemos estado haciendo esto antes con el operador line-input STD. Si evaluamos este operador en contexto escalar, obtendremos la próxima línea de la entrada. Por ejemplo:

```
1 $line = <STDIN>;          # lectura de la siguiente línea
2 chomp($line);             # y chomp
3 chomp($line = <STDIN>);    # lo mismo más idiomáticamente
4
```

Dado que del operador de línea de entrada devolverá `undef` cuando llegue al final del archivo, podemos usar esto para salir de un ciclo:

```
1 while (defined($line = <STDIN>)) {
2     print "I saw $line";
3 }
```

Muchas cosas ocurren esa primera línea: estamos leyendo la entrada en una variable, se comprueba que este definida, y si lo está (lo que significa que no hemos llegado al final de la entrada) entramos al bucle `while`. Luego dentro del bucle, vamos a tener cada línea una tras otra en la variable `$_`. Es importante resaltar que no estamos haciendo `chomp` de la entrada. En este tipo de bucles, no puedes poner `chomp` en la expresión condicional, entonces lo que se hace con frecuencia, si es necesario, es poner `chomp` en la primera línea del cuerpo del bucle. Como es algo que vas a querer hacer con bastante frecuencia, naturalmente como ya hemos visto antes, Perl tiene un atajo para hacer esto. El atajo se ve así:

```
1 while (<STDIN>){
2     print "Say: $_";
3 }
```

De esta manera, las líneas leídas de la entrada estándar, automáticamente van a estar contenidas en la variable mágica `$_` por defecto. Pero tenga cuidado, esto funciona solo cuando no hay mas nada que `<STDIN>` en la expresión condicional de un bucle `while`. Si colocas algo más en la expresión condicional, este atajo no va a funcionar.

Es importante destacar que cuando evaluamos el operador `line-input` en la expresión condicional de un ciclo `while`, lo estamos evaluando en un contexto escalar.

De otra manera, evaluando el operador `line-input` en contexto de lista, obtendremos todas las líneas de la entrada como una lista, cada elemento de la lista es una línea:

```
1   foreach (<STDIN>){
2       print "Say: $_";
3   }
```

No existe una conexión entre la variable mágica `$_` y el operador `line-input` (`STDIN`). Para este caso, la variable de control por defecto de un bucle `foreach` es `$_`.

Posiblemente no vea la diferencia aun entre como relaciona Perl al operador `line-input` y la variable mágica `$_` en el bucle `while` y el bucle `foreach`. En el bucle `while`, Perl lee la línea entrante y la asigna a la variable `$_`, y luego entra al bucle. Luego regresa, lee la siguiente línea y la asigna de igual manera a la variable mágica `$_` y entra nuevamente al bucle. Pero en un bucle `foreach`, el operador `line-input` es utilizado en un contexto de lista (porque el `foreach` necesita una lista que iterar). Entonces lee todas las líneas antes de iniciar el bucle. La diferencia la podemos ver en el momento en que tengamos que procesar por ejemplo 400 MB de archivos de registro (logs), en este caso generalmente es mejor usar `while`, pues va a procesar una línea a la vez.

El Operador Diamante (La Cometa)

Otra forma de leer la entrada de datos es con el operador diamante (o cometa): `<>`. Es muy útil para usar en programas que corren en sistemas operativos tipo Unix, con respecto a la invocación de argumentos. Si desea que sus programas en Perl puedan usarse como utilidades de Unix (cat, sed, sort, grep, lpr, y otras mas), el operador diamante va a ser su amigo.

La invocación de argumentos de un programa normalmente son un numero de “palabras” en la línea de comandos que vienen después del nombre del programa. En este caso, vamos a darle nombres de archivos y su programa va a procesarlos en secuencia:

<pre>Terminal \$./my_program fred barney betty</pre>

El comando anterior, corre el *comando* `my_program` (que debe estar en el directorio actual), y el programa va a procesar los archivos `fred`, `barney`, y `betty`

Si no invocamos argumentos, el programa va a procesar la entrada estándar. O, como caso especial, si le da un solo guion (-) como argumento esto también significa leer desde la entrada estándar. Entonces, si la lista de argumentos es: `fred - betty`, va a leer el archivo `fred`, luego va a leer la entrada estándar y luego va a leer el archivo `betty`.

La ventaja de hacer que sus programas funcionen de esta manera es que tu puedes escoger el tipo de entrada de datos en tiempo de ejecución, por ejemplo no necesitas reescribir un programa para usar una tubería (|). Larry puso esta característica en Perl, porque quería que fuera fácil para usted, escribir programas que trabajaran como utilidades estándar de Unix.

El operador diamante, es actualmente un operador especial de `line-input`. Pero en lugar de obtener la entrada desde el teclado, este toma los datos del tipo de entrada que el usuario escoja:

```

1   while (defined($line = <>)) {
2       chomp($line);
3       print "It was $line that I saw!\n";
4   }

```

Entonces, con este programa, si le damos los archivos fred, barney, y betty, va a procesar las líneas del archivo fred en la variable \$line, luego que termine con el archivo fred va por el archivo barney y luego por el archivo betty. Note que el ciclo no se rompe entre un archivo y otro, cuando usamos el operador diamante, todos los archivos se juntan en un solo archivo grande.

El operador diamante va a retornar **undef** cuando llegue al final de todo el conjunto de archivos, con lo que se va a terminar el ciclo.

De igual manera el atajo de la variable mágica \$_ se puede usar con este operador. Por ejemplo:

```

1   while(<>){
2       chomp;
3       print "It was $_ that I saw!\n";
4   }

```

Esto funciona exactamente igual al código anterior, solo que escribimos menos. Y debe haber notado que estamos usando el comportamiento por defecto de **chomp**; si no le damos argumentos a **chomp**, va a trabajar con la variable mágica \$_.

Cuando el operador de diamante no puede abrir un archivo, muestra un mensaje de diagnostico:

Terminal Can't open maria: No such file or directory
--

5.2. Invocación de Argumentos

Técnicamente, usar el operador de diamante no hace invocación de argumentos. El funciona usando el array @ARGV. Este array es un array especial que esta predefinido en el interprete de Perl como la lista de invocación de argumentos. En otras palabras, este es otro array con un divertido nombre en mayúsculas sostenidas, cuando tu programa inicia, @ARGV ya esta lleno con la lista de argumentos.

Puedes usar el array @ARGV como cualquier otro array, puedes sacar cosas de el con **shift** o usar un **foreach** para iterarlo.

El operador de diamante busca en @ARGV para determinar que nombres de archivos debe usar. Si encuentra una lista vacía, usa lo que viene de la entrada estándar. Esto significa que después que el programa inicia y antes de usar el operador de diamante, tienes chance de jugar con @ARGV. Por ejemplo, si sabemos que archivos queremos procesar, podemos poner los nombres en @ARGV sin necesidad de leerlos de la entrada estándar.

```

1   @ARGV = qw# larry moe curly #; # Fuerza a leer estos archivos
2   while (<>){
3       chomp;
4       print "$_ Esto fue lo que vi en los archivos títeres";
5   }
6 =end programlisting

```

5.3. Salida hacia la Salida Estándar

El operador `print`, toma una lista de valores y envía cada elemento (como una cadena por supuesto) hacia la salida estándar, uno después del otro. No agrega nada delante, atrás, o entre los elementos. Si quiere algo adicional, como un salto de línea o un espacio, tiene que colocarlo a mano usted mismo.

Por su puesto, esto significa que hay una diferencia entre imprimir un array e interpolar un array:

```
1      print @array;    # Imprime la lista de elementos
2      print "@array"; # Imprime una cadena (el array interpolado)
```

La primera instrucción va a imprimir la lista de elementos, uno tras el otro, sin colocar espacios en el medio ni nada. La segunda instrucción va a imprimir cada elemento con un espacio entre cada elemento. Ejemplo.

```
DB<36> @a = qw/maria pedro juan/
```

```
DB<38> print @a
```

```
DB<39> print "@a";
      maria pedro juan
```

mariap

Dado que `print` espera una lista de cadenas y las imprime, sus argumentos son evaluados en contexto de lista. Dado que el operador diamante devuelve una lista de líneas en contexto de lista. Podemos reescribir las dos clásicas herramientas de Unix (**cat** y **sort**).

```
1      print <>;        # cat en Perl
2
3      print sort <>;   # sort en Perl
```

Claro, para ser honestos, los comandos `cat` y `sort` tienen una serie de funcionalidades adicionales que estos pequeños programas no tienen.

Seguro a notado que `print` al ser una función, puede omitir los paréntesis, recuerde que puede hacer esto siempre y cuando quitar los paréntesis no cambie el significado de la expresión, supongamos que tenemos el siguiente ejemplo:

```
1      print (2+3);
```

Parece una llamada a función y es una llamada a función. Esto imprime 5, y retorna un valor como cualquier otra función. El valor de retorno de `print` es verdadero o falso, indicando que efectivamente realizó la impresión, a menos que ocurra algún error del tipo I/O, el resultado normal de `print` va a ser **1**.

```
1      $result = print("Hola mundo\n");
```

Pero supongamos que tenemos el siguiente código:

```
1      print (2+3)*4;
```

¿ Que cree que va a ocurrir con esta instrucción ?, definitivamente no va a salir 20 por la pantalla. Perl va a imprimir 5, y va a multiplicar 1 por 4, y el resultado lo va a tirar a la basura.

Este es un problema típico de los paréntesis opcionales, a veces los humanos olvidamos que los paréntesis realmente importan. Cuando usamos `print` sin los paréntesis, `print` es

un operador de lista, imprime todos los elementos de esa lista. Pero cuando la primera cosa después de print es un paréntesis, print es una llamada de función, y va a imprimir solo lo que encuentre entre los paréntesis.

Actualmente, esta regla “Si parece una llamada a una función, es una llamada a función, aplica para todas las funciones de lista.

5.4. Formatear la salida con printf

Puede ser que quiera tener mas control con la salida de lo que ofrece print. En efecto, puede estar acostumbrada a formatear la salida con la función `printf` de Perl. No se preocupe, Perl provee una operación compatible con el mismo nombre.

El operador `printf` toma una cadena con formato seguida de una lista de cosas a imprimir. Ejemplo:

```
1 printf "Hola, %s; su contraseña expira en %d días!\n",
2 $user, $dias_para_morir;
```

La cadena con formato posee un numero de conversiones, estas conversiones comienzan con un signo de porcentaje (%) y termina con una letra. Debe haber la misma cantidad de elementos en la cadena con formato que en la lista que se pasa como argumento.

Para imprimir números generalmente es bueno usar la conversión `%g`, que automáticamente escoje entre punto flotante, entero, o notación exponencial, cual sea necesaria.

```
1 printf "%g %g %g\n", 5/2, 51/17, 51 ** 17; # 2.5 3 1.0683e+29
```

El formato `%d` indica decimal entero truncado.

```
1 printf "in %d days!\n", 17.85; # in 17 days!
```

Note que este proceso de truncado no redondea, en un momento vamos a ver como redondear números.

En Perl, `printf` es usado con frecuencia para datos en columnas, la mayoría de los formatos permiten indicar el ancho del campo. Si los datos no llenan el campo, de igual manera el campo se expande dejando del espacio requerido.

```
1 printf "%6d\n", 42; # se vería asi:  '42 (' representa espacio blanco)
2 printf "%2d\n", 2e3 + 1.95; # 2001
```

La conversión `%s` se usa para las cadenas, efectivamente interpola el valor obtenido como una cadena, y también se le puede indicar el tamaño del campo:

```
1 printf "%10s\n", "wilma"; # se vería  'wilma
```

Un valor negativo en el tamaño del campo, indica justificado a la izquierda:

```
1 printf "%-15s\n", "flinstone"; # se ve flinstone
```

La conversión `%f` redondea, y permite indicar cuantos números quiere después del punto decimal.

```
1 printf "%12f\n", 6 * 7 + 2/3; # se ve 42.666667
2 printf "%12.3f\n", 6 * 7 + 2/3; # se ve 42.667
3 printf "%12.0f\n", 6 * 7 + 2/3; # se ve 43
```

Para imprimir un signo de porcentaje, usamos `%%`:

```
1 printf "Interés Mensual : %.2f%%\n", 5.25/12; # El valor seria "0.44%
```

5.5. Arrays y printf

Generalmente, usted no va a usar un array como argumento para `printf`. Esto es porque un array almacena cualquier numero de elementos, y la cadena de formato viene dada para trabajar con un numero finito de elementos. Si hay tres conversiones de formato, deben haber exactamente tres elementos.

Pero no hay razón para no hacer la cadena de formato al vuelo, puesto que puede ser cualquier expresión. Este puede ser un truco para hacer esto:

```

1   my @items = qw( wilma dino pebbles );
2   my $format = "The items are:\n" . ("%10s\n" x @items);
3   ## print "the format is >>$format<<\n"; # for debugging
4   printf $format, @items;
```

Aquí usamos el operador `x` (Que aprendimos en el capítulo 2) para replicar la cadena dada el número de veces dado por `@items` (en contexto escalar).

Pero podemos hacer esto mas corto y mas mágico:

```

1   printf "The items are:\n".("%10s\n" x @items), @items;
```

Note que `@items` esta siendo usado en contexto escalar, para obtener su longitud, y en contexto de lista para obtener sus valores. El **Contexto** es importante.

5.6. Filehandles

Un Filehandle es el nombre en un programa en Perl para una conexión de E/S entre el proceso Perl y el mundo exterior. Es decir, es el nombre de una conexión, no necesariamente el nombre de un archivo.

Los Filehandle se llaman igual que otros identificadores de Perl (con letras, dígitos y pisos bajos) pero no pueden comenzar con un dígito.

Actualmente hay seis nombres de Filehandle especiales en Perl que se usan para propósitos específicos: `STDIN`, `STDOUT`, `STDERR`, `DATA`, `ARGV`, y `ARGVOUT`. Aunque puede elegir cualquier nombre para un Filehandle, no debería usar ninguno de estos seis nombres, a menos que quiera hacer algo muy especial.¹

Cuando su programa inicia, el Filehandle llamado `STDIN` es la conexión entre el proceso Perl y lo que sea que el programa tenga como entrada, conocido como entrada estándar o *standard input stream*. Esto generalmente es el teclado del usuario a menos que el usuario use otra cosa como fuente de datos para la entrada, como un archivo o la salida de otro programa a través de una *tubería*. También esta la salida estándar o *standard output stream*, que es el Filehandle `STDOUT`. Por defecto, esta conectada con la pantalla del usuario, pero el usuario puede enviar la salida a un archivo o a otro programa a través de una *tubería*. Esto del flujo estándar o *standard stream* viene de la librería de Unix *standard I/O*.

La idea general es que el programa lea desde `STDIN` y escriba en `STDOUT`, confiando en el usuario (o generalmente un programa que inicie su programa). De esta forma, el usuario puede escribir una línea de comandos en el prompt del shell como esta:

¹En algunos casos puede reusar estos nombres sin ningún problema. Pero un programador menos experimentado que mantenga el código va a confundirse

Terminal

```
$ ./su_programa <dino >willma
```

Esto le dice a la terminal, que la entrada de su programa viene de *dino* y la salida va a *willma*. Esto funciona siempre y cuando la entrada de su programa sea STDIN y la salida valla a STDOUT.

Sin ninguna modificación el programa va a trabajar sin problemas con *tuberías*. Este es otro concepto de Unix, que le permite escribir líneas de comando como la siguiente:

Terminal

```
$ cat fred barney | sort | ./su_programa | grep algo | lpr
```

5.6.1. Abrir un Filehandle

Como ya sabe, Perl provee tres filehandles - STDIN, STDOUT, STDERR, que automáticamente abren archivos o dispositivos establecidos por el programa padre del proceso (probablemente la terminal). Cuando necesita otros filehandles, debe usar el operador `open` para decirle a Perl, que le pida al sistema operativo que abra una conexión entre su programa y el mundo exterior. Aquí tenemos algunos ejemplos:

```
1 open CONFIG, "dino";
2 open CONFIG, "<dino";
3 open BEDROCK, ">fred";
4 open LOG, ">>logfile";
```

El primero, abre un filehandles llamado `CONFIG` a un archivo llamado `dino`. Esto es, el archivo *dino* va a ser abierto y lo que sea que tenga en su contenido, va a estar disponible en el programa a través del filehandle llamado `CONFIG`. El segundo hace exactamente lo mismo que el primero, pero con un signo *menor que* indicando “Use este nombre de archivo como entrada”, a pesar de que eso es lo que hace por defecto.

Para abrir el filehandle `BEDROCK` para enviar la salida allí, usamos el signo *mayor que*, esto envía la salida al archivo `fred`. Si el archivo ya existe y tiene contenido, automáticamente reemplaza el contenido del archivo con la salida que enviamos al filehandle.

El cuarto ejemplo, indica como podemos usar dos signos *mayor que* para abrir un archivo sin reemplazar su contenido y anexar nuestra salida al final del archivo. Si no existe el archivo, lo crea y agrega el contenido, como si se tratara de un solo signo *mayor que*. Esta es una forma de manejar los archivos de registro (logfile). El programa puede escribir un grupo de líneas al archivo de registro cada vez que se ejecuta.

Puedes usar una expresión en lugar de un nombre de filehandle específico. Luego solo queda indicar la dirección explícita:

```
1 my $selected_output = "my_output";
2 open LOG, ">" $selected_output;
```

Note que el espacio después del signo *mayor que*. Perl ignora esto.

En desde la version 5.6 de Perl, puedes usar un tercer argumento extra para `open`:

```
1 open CONFIG, "<", "dino";
2 open BEDROCK, ">", $file_name;
3 open LOG, ">>", &logfile_name();
```

5.6.2. Filehandles Malos

Perl actualmente no abre el archivo por el mismo, Perl le pide al sistema operativo que abra el archivo por el. Por su puesto, el sistema operativo puede negarse a hacerlo debido a una configuración de permisos, un nombre de archivo incorrecto u otras razones.

Si intentas leer de un filehandle malo, vas a ver inmediatamente un **end-of-file** o fin de archivo. **end-of-file** indica **undef** en contexto escalar o una lista vacía en contexto de lista.

Si intentas escribir en un filehandle malo, los datos simplemente se perderán.

Afortunadamente, estas consecuencias son fáciles de evitar. Primero que todo, si estamos usando el pragma **warnings** o el switch **-w**, Perl va a advertirnos cuando vea que estamos usando un filehandle malo. Pero incluso antes de esto, la llamada **open** siempre nos indica si puedo hacer el trabajo o fallo, retornando verdadero o falso. Entonces, puedes escribir tu código de la siguiente manera:

```
1      my $success = open LOG, ">>logfile"; # capture the return value
2      if ( ! $success) {
3          # Fallo Open
4          ...
5      }
```

5.6.3. Cerrar un Filehandle

Cuando termines de trabajar con el Filehandle, puedes cerrarlo con el operador de cierre, de la siguiente forma:

```
1      close BEDROCK;
```

Cerrar un Filehandle le dice al sistema operativo, que hemos terminado la conexión con el archivo, entonces los últimos datos enviados por esa conexión son escritos al disco, si eso es lo que se espera.

Perl, automáticamente cierra un filehandle si lo abres de nuevo (esto ocurre si reusas el nombre del filehandle en otra llamada a **open**) o si sales del programa.

Debido a esto, varios programas simples de Perl no cierran los filehandles. Pero si quieres ser ordenado, debe haber una llamada a **close** por cada llamada a **open**. En general, es mejor cerrar cada filehandle pronto después de haber terminado con el. Aun que este cerca del fin del programa.

5.7. Errores fatales con die

Miremos hacia un lado un momento. Necesitamos saber algunas cosas mas que aunque no están relacionadas con los procesos de I/O nos va a permitir salir de un programa antes de lo esperado.

Cuando ocurre un error fatal en Perl (por ejemplo, si haces una division por cero, uso de una expresión regular inválida, o llamar una subrutina que no esta definida), sus

programas se rompen con un mensaje de error indicando la razón del porqué ². Esta funcionalidad esta disponible para nosotros a través de la función `die`, entonces podemos hacer nuestros propios errores fatales.

La función `die` imprime el mensaje que le pasas como argumento, y se asegura que el programa termine con una salida diferente de cero.

Probablemente no lo sepa, pero cada programa que se corre en Unix tiene un estado de salida, indicando cuando termina satisfactoriamente y cuando no. Programas que corren otros programas (como la utilidad `make`) mira el estado de salida para saber si todo va bien. El estado de salida es un simple byte, tradicionalmente 0 es *termine satisfactoriamente* y diferente de 0 es lo contrario.

Entonces, podemos escribir el ejemplo anterior de la siguiente manera:

```
1   if ( ! open LOG, ">>logfile" ) {
2       die "No se pudo crear el archivo de logs: $!";
3   }
```

Si `open` falla, `die` va a terminar el programa diciendo: “No se pudo crear el archivo de logs. Pero, ¿ Que es esa variable \$! en el mensaje ?, Esto es, una queja del sistema que un ser humano puede leer. En general, cuando el sistema se niega a hacer algo que le estamos pidiendo, en \$! esta la razón del porqué.

Hay otra cosa interesante que `die` puede hacer por nosotros, automáticamente va a indicar el nombre del programa y el número de linea en donde ocurrió el error.

Suponiendo que en el programa anterior no podemos abrir el filehandle por problemas con los permisos, vamos a obtener el siguiente mensaje:

<pre>Terminal No se pudo crear el archivo de logs: permission denied at your_program line 1234.</pre>

5.8. Mensajes de Advertencia con warn

Así como `die` indica un mensaje de error fatal, puedes usar `warn` para indicar un advertencia como si fuera una advertencia interna de Perl.

La función `warn` trabaja igual que la función `die`, excepto que esta función no termina el programa. Pero si agrega el nombre del programa y la linea e imprime el mensaje hacia la salida estándar de errores (STDERR).

5.9. Usando los Filehandle

Cuando un filehandle se abre para leer, puedes leer las lineas justo como leemos la entrada estándar con STDIN. Entonces, por ejemplo para leer las lineas de un archivo `passwd` de Unix:

```
1   if ( ! open PASSWD, "/etc/passwd" ) {
2       die "How did you get logged in? ($!)";
3   }
```

²Bueno, esto pasa por defecto, pero podemos atrapar los errores con la función `eval`, que vamos a ver mas adelante en otro capítulo

```

4
5     while (<PASSWD>) {
6         chomp;
7     }

```

Un filehandle abierto para escritura puede usarse con `print` o `printf`, colocándolo inmediatamente después de la palabra reservada pero antes de la lista de argumentos:

```

1     print LOG "Captain's log, stardate 3.14159\n"; # output goes to LOG
2     printf STDERR "%d percent complete.\n", $done/$total * 100;

```

5.9.1. Cambiar un Filehandle de salida por defecto

Por defecto, si no le indicas un filehandle a `print` o a `printf`, la salida va a ir a `STDOUT`. Pero esto se puede cambiar con el operador `select`. Por ejemplo, vamos a enviar un par de líneas al filehandle `BEDROCK`:

```

1     select BEDROCK;
2
3     print "Hola esto es una prueba de piedra, o una piedra de prueba";
4     print "Willma!\n";

```

Hay que tener cuidado con esto, una vez que se termine de usar, se debe volver al valor por defecto `STDOUT`. De igual manera, por defecto, la salida hacia un filehandle se guarda en el buffer. Configurando la variable especial `$|` a 1, va a hacer que el filehandle seleccionado siempre haga flush del buffer después de cada operación de salida. Entonces si se quiere asegurar que los archivos de log tengan los registros a tiempo, en el caso que quieras estar leyendo el log para monitorear el progreso del programa, puedes hacer algo como esto:

```

1     select LOG;
2     $| = 1;          # No mantenga las entradas de LOG en el buffer.
3     select STDOUT;
4     # El tiempo pasa, los niños aprenden a caminar, Chavez no se va ...
5     print LOG "Esto es escrito en el log de una vez\n";

```

5.9.2. Reabrir un Filehandle por defecto

Ya mencionamos antes que si abres un archivo el anterior se cierra automáticamente. Y también dijimos que no debería reusar uno de los seis nombres de filehandles por defecto, al menos que quiera hacer algo especial. Adicionalmente dijimos que los mensajes de error en Perl van a `STDERR`. Si usa estas tres piezas de información juntas, ahora puedes tener una idea de como enviar los mensajes de error a un archivo.

```

1     # Envía los errores a mi log privado de errores
2     if ( ! open STDERR, ">>/home/elsanto/.error_log"){
3         die "No pude abrir el log de errores: $!";
4     }

```

¿ Que pasa si no es posible abrir el archivo de LOGS ?, la operación de reapertura de uno de los tres filehandles (`STDIN`, `STDOUT`, `STDERR`) falla, Perl amablemente restaura al valor original.

5.10. Salida con say

Perl 5.10 toma la función `say` del desarrollo de Perl 6. Es lo mismo que `print`, con la diferencia de que agrega al final un carácter de nueva línea (`\n`). Las siguientes sentencias producen la misma salida:

```
1 use 5.010;
2
3 print "Hello!\n";
4 print "Hello!", "\n";
5 say "Hello!";
```

Para interpolar una variable escalar o una lista, debe colocarse la variable entre comillas dobles, ejemplo:

```
1 use 5.010;
2
3 my @array = qw( a b c d );
4
5 say @array;      # "abcd\n"
6
7 say "@array";    # "a b c d\n"
```

Al igual que con `print`, puedes especificar un filehandle con `say`:

```
1 use 5.010;
2
3 say BEDROCK "Hola !";
```

Esta es una nueva característica de Perl 5.10, y nosotros la usamos solo cuando estamos usando alguna otra función de Perl 5.10. La vieja y confiable `print` sigue siendo tan buena como siempre lo ha sido, pero sospechamos que hay algunos programadores de Perl que van a querer ahorrarse el tener que escribir cuatro caracteres extra. (dos del nombre y `\n`).

5.11. Ejercicios

1. Escriba un programa que actúe como `cat`, pero que invierta el orden de las líneas en la salida. (Algunos programas tienen `tac`). Si corres tu programa como `./tac prueba prueba2 prueba3`, la salida debe ser todo el archivo `prueba3` desde la última línea hasta la primera, luego todo el archivo `prueba2` desde la última línea hasta la primera y luego el archivo `prueba` desde la última línea hasta la primera.
2. Escriba un programa que pregunte al usuario una lista de cadenas en líneas separadas, imprima cada cadena justificada a la derecha, en una columna de 20 caracteres. Para verificar que la columna tiene esa cantidad de caracteres, imprima una línea guía con los números de las columnas. Asegúrese que las cadenas no tengan más de 20 caracteres. Ejemplo:

<pre>123456789012345678901234567890123456789012345678901234567890 hello good-bye</pre>
--

5 Entrada y Salida

3. Modifique el programa anterior, para que le permita al usuario indicar el numero del tamaño de la columna en la primera linea, y luego en las otras lineas separadas, las cadenas. Recuerde que ahora la linea guía debe ajustarse a este número.

6 Hashes

En este capítulo, usted va a ver una de las características que hacen de Perl uno de los mejores lenguajes del mundo de programación imperativa: Hashes ¹

Aunque los Hashes son una útil y poderosa característica, es probable que hallas usado otros lenguajes poderosos durante años sin haber escuchado sobre los Hashes. Pero usted va a usar hashes en casi todos los programas en Perl que escriba de ahora en adelante.

6.1. ¿ Que es un Hash ?

Un hash es una estructura de datos, no es muy diferente a un array, que puede almacenar cualquier número de valores y obtenerlos cuando quiera. Pero en lugar de indexar los valores por un numero, como lo hacemos con los arrays, vamos a indexar los valores por un nombre. Esto significa, que los indices (a los que vamos a llamar llaves) no son números, en lugar de esto vamos a usar cadenas arbitrarias únicas.

Primero que todo, las llaves son cadenas de caracteres, entonces, en lugar de usar el numero 3 para obtener un elemento de un array, vamos a acceder al elemento del hash llamado `wilma`.

Estas llaves, son cadenas arbitrarias - usted puede usar cualquier expresión de cadena para la llave de un hash. Y estas son cadenas únicas - de igual manera que en un array tenemos un único elemento numérico 3, aquí vamos a tener un único elemento del hash llamado `wilma`.

Otra forma de pensar en un hash, es compararlo con un barril de datos, donde cada pieza de datos tiene una etiqueta. Tu puedes llegar al barril y sacar una etiqueta y mirar que pieza de datos contiene. Pero no va a ser el “primer” elemento del barril. En un array, comenzamos desde el elemento 0, seguido del elemento 1, el elemento 2 y así. Pero en un hash, no hay un orden fijo, no hay un primer elemento. Es solo una colección de pares llave-valor.

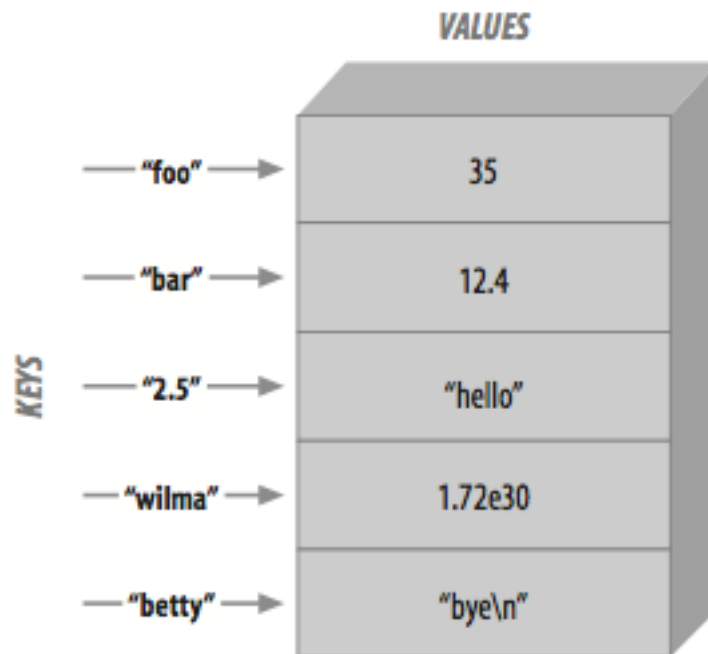
Las llaves y los valores son un conjunto de valores escalares ordinarios, pero las llaves son siempre convertidas a cadenas. Entonces, si usas la expresión numérica 50/20 como una llave, esto se va a convertir en una cadena de tres caracteres “2.5”.

Como es usual en Perl, “no existen los límites innecesarios”, esto también aplica a un hash, un hash puede ser de cualquier tamaño, desde un hash vacío con cero pares llave-valor, hasta cualquier cantidad de valores con los que pueda llena su memoria.

Algunas implementaciones de hashes (como originalmente era en el lenguaje awk) cuando los hashes son de gran tamaño tienden a ser lentos. Este no es el caso en Perl, en

¹En los viejos tiempos, los llamábamos “arrays asociativos”. Pero la comunidad de Perl decidió por 1995 que eran muchas letras al escribirlas y muchas sílabas al pronunciarlo, entonces cambiaron el nombre a “Hashes”

Figura 6.1: Hashes



Perl es un algoritmo eficiente, bueno y escalable. Entonces, si un hash tiene un solo árbol de pares llave-valor, es muy rápido “buscar en el barril” y tomar uno de esos valores. Si un hash tiene tres millones de pares clave-valor, debe ser igual de rápido tomar un valor. No se asuste, los hashes grandes no muerden.

Vale la pena volver a mencionar de nuevo que las llaves de un hash son siempre únicas, aunque los valores pueden estar duplicados. Los valores de un hash pueden ser todos números, cadenas, valores `undef`, o una mezcla ². Pero las llaves son siempre arbitrariamente cadenas únicas.

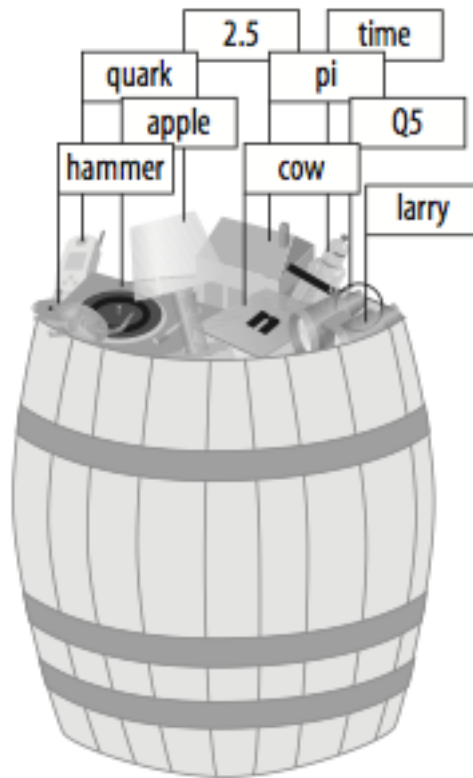
6.2. ¿ Porqué usar un Hash ?

Cuando usted escucha por primera vez hablar de hashes, especialmente si usted ha vivido gran parte de su vida productiva como programador en otros lenguajes que no poseen hashes, podría preguntarse, quien querría una de estas criaturas extrañas. Pues bien, la idea general es que usted tendrá un grupo de datos relacionado con otro grupo de datos. Por ejemplo, aquí tenemos algunos de los hashes que podrá encontrar en aplicaciones típicas de Perl:

Nombres y Apellidos

²En efecto, cualquier valor escalar, incluyendo otros tipos de datos escalares.

Figura 6.2: Hashes como Barril



En este caso, el primer nombre es la llave, y el apellido es el valor. Este tipo de hash requiere, por supuesto, que los nombres sean únicos. Si tiene dos personas con el mismo nombre, este ejemplo no va a funcionar para usted. Con este hash, puedes buscar a cualquiera por el nombre y obtener su apellido. Si usted busca por el nombre **Walter**, va a obtener el valor **Vargas**.

Nombre de Host y dirección IP

Usted debe saber que cada computadora en Internet tiene un nombre de host y una dirección IP. Esto es porque a las máquinas les gusta trabajar con números, pero a los humanos se les hace más fácil recordar nombres. Los nombres de hosts son cadenas únicas, entonces pueden usarse para hacer este hash. Con este hash puedes buscar por el nombre de host y obtener la dirección IP correspondiente.

Palabras, contar el número de veces que una palabra aparece.

La idea aquí es, que si usted quiere saber cuantas veces aparece una palabra en un documento dado. Puede hacer un hash que contenga como llaves del hash a las palabras y como valor el número de repeticiones de la palabra.

Cédula y Nombre

Este es un ejemplo común, ya sabemos que los números de cédulas son cadenas únicas, entonces podemos tener un hash que contenga como llave el número de cédula y como valor el nombre de la persona.

6.3. Acceder al elemento de un Hash

Para acceder al elemento de un hash usamos la siguiente sintaxis:

```
1 $hash{$alguna_llave_del_hash}
```

Es bastante similar a la sintaxis que usamos para acceder a un array, pero en este caso vamos a usar llaves en lugar de corchetes alrededor del subíndice (llave).³

Podemos asignar los valores de la siguiente manera:

```
1 $apellido{"fred"}      = "flintstone";
2 $apellido{"barney"}    = "rubble";
3 $apellido{"hugo"}      = "Chávez";
```

Con el hash anterior, podríamos usar el siguiente código para accederlo:

```
1 foreach $persona (qw< fred barney hugo >){
2     print "He oído algo sobre $persona $apellido{$persona}.\n";
3 }
```

El nombre del hash es de igual manera otro identificador de Perl (letras, dígitos y pisos bajos, pero no puede iniciar con un número). Y son de un espacio de nombres separado, lo que significa que no hay conexión entre la variable `$apellido{"walter"}`, la subrutina `&apellido` o la variable `$apellido`.

Por su puesto, la llave de un hash puede ser una expresión, no solo cadenas literales o variables escalares simples. Por ejemplo:

```
1 $foo = "bar";
2 print $apellido{ $foo . "ney"};    # Imprime "rubble"
```

Cuando guardamos algo en un elemento que ya existía en el hash, este sobre escribe al valor anterior:

```
1 $apellido{"fred"} = "astaire";
```

Es igual a lo que pasa con los arrays y los escalares.

Si accede a un elemento que no se encuentra en el hash va a obtener `undef`:

```
1 $granito = $apellido{"larry"};    # No hay larry: undef
```

³Aquí daremos un vistazo dentro de la mente de Larry Wall: Larry decía, vamos a usar llaves en lugar de corchetes porque estamos haciendo algo mas sofisticado que acceder a un array, entonces debemos usar un signo de puntuación mas sofisticado.

6.4. Hash como un todo

Para referirse a un hash completo, usamos el signo de porcentaje (%) como sígil. Entonces, el hash que hemos estado usando en las últimas páginas actualmente se llama `%apellido`.

Por conveniencia, un hash puede convertirse en una lista, y luego volver a convertirlo en un hash. Asignando un hash en contexto de lista, donde la lista esta conformada por un conjunto de pares llave-valor.

```
1  %some_hash = (
2      "foo", 35,      "bar", 12.4, 2.5, "hello",
3      "wilma", 1.72e30, "betty", "bye\n"
4  )
```

El valor de un hash (en contexto de lista) es una lista simple de pares llave-valor:

```
1  @any_array = %some_hash;
```

A esto le llamamos *desenrollar* el hash, básicamente convertimos el hash en una lista de pares llave-valor. Como era de esperarse, los pares no están necesariamente en el mismo orden de la lista original.

```
1  print "@any_array\n";
2      # vamos a obtener algo como esto:
3      # betty bye (una nueva linea) wilma 1.72e+30 foo 35 2.5 hello bar 12.4
```

El orden es algo confuso porque Perl mantiene los pares llave-valor en el orden que sea mas conveniente para Perl, de manera que pueda buscarse cualquier item rápidamente. Se usa un hash cuando no nos importa el orden de los elementos, o tenemos una forma fácil de ordenarlos como queramos.

6.5. Asignación de Hash

Esto no es algo común, pero usted puede copiar un hash a otro con la siguiente sintaxis obvia:

```
1  %new_hash = %old_hash;
```

Esto es mas complejo para Perl de lo que parece, mientras en otros lenguajes como C o Pascal, es tan simple como copiar un bloque de memoria, las estructuras de datos en Perl sin mas complejas. Entonces, esta linea de código le dice a Perl que desenrolle a `%old_hash` en una lista de pares llaves-valor, y que arme el hash `%new_hash` desde la lista de pares llave-valor.

Es mas común hacer otro tipo de transformaciones con un hash. Por ejemplo, podemos obtener el inverso de un hash así:

```
1  %inverse_hash = reverse %any_hash;
```

6.6. La Gran Flecha

Cuando asignamos una lista a un hash, algunas veces no es obvio que elementos son llaves y que elementos son valores. Por ejemplo, en esta asignación, los humanos debemos ir contando por la lista diciendo: “llave, valor, llave, valor, llave, valor ...”, para poder determinar si 2.5 es una llave o un valor:

```
1  %some_hash = (
2      "foo", 35, "bar", 12.4, 2.5, "hello",
3      "wilma", 1.72e30, "betty", "bye\n"
4  );
```

¿ No sería agradable que Perl nos diera una manera emparejar las llaves y los valores en contexto de lista ?, Larry también pensaba esto y por eso invento a la flecha mayor (\Rightarrow). En Perl, es simplemente un hechizo para representar una coma, por lo que algunas veces se le llama “la coma gorda”. En la gramática de Perl, cada vez que usted necesite una coma (,), puedes usar la flecha grande, esto va a ser lo mismo para Perl. Por ejemplo:

```
1  my %last_name = ( # Un hash también puede ser una variable léxica.
2      "fred" => "flintstone",
3      "dino"  => undef,
4      "barney" => "rubble",
5      "betty" => "rubble",
6  );
```

Como puede ver, es fácil determinar en el ejemplo anterior, que elementos son valores y que elementos son llaves.

6.7. Operadores de Hash

Naturalmente, hay algunas funciones útiles para poder trabajar con hashes.

6.7.1. Las funciones keys y values

La función `keys` obtiene una lista de todas las llaves de un hash, mientras que la función `values` obtiene los valores correspondientes. Si el hash no tiene elementos, ambas funciones retornan una lista vacía.

```
1  my %hash = ("a" => 1, "b" => 2, "c" => 3);
2  my @k = keys %hash;
3  my @v = values %hash;
```

En el ejemplo anterior `@k` va a contener “a”, “b”, y “c”, y `@v` va a contener 1, 2, y 3 en el mismo orden. Recuerde que Perl no mantiene el orden de un hash. Pero cual sea el orden que tenga la lista de llaves va a ser el orden correspondiente de la lista de valores.

En contexto escalar estas funciones retornan la cantidad de elementos en el hash. Ejemplo:

```
1  my $count = keys %hash;
```

En un ciclo `while`, lo podemos usar como expresión en el contexto booleano, por ejemplo:

```
1  if (%hash){
2      print "Esto es un valor verdadero\n";
3  }
```

6.7.2. La función `each`

Si usted quiere iterar el hash completo, una de las maneras usuales es usar la función `each`, que retorna un par llave-valor como una lista de dos elementos. En cada evaluación de esta función sobre el mismo hash, va a retornar el siguiente par llave-valor, hasta que todos los elementos sean accedidos. Cuando no hay mas pares, `each` retorna una lista vacía.

En la práctica, la única manera de usar `each` es en un ciclo `while`, por ejemplo:

```
1 while ( ($key, $value) = each %hash ) {
2     print "$key => $value\n";
3 }
```

Por supuesto, `each` retorna los pares llave-valor desordenadamente. Si necesita recorrer el hash en orden, simplemente ordena las llaves, por ejemplo:

```
1 foreach $key ( sort keys %hash ) {
2     $value = $hash{$key};
3     print "$key => $value \n";
4 }
```

6.7.3. La función `exists`

Para verificar que una llave exista en un hash, puede usar la función `exists`, que retorna un valor verdadero si la llave existe en el hash, de lo contrario retorna falso:

```
1 if (exists $books{"dino"}) {
2     print "Hey, there's a library card for dino!\n";
3 }
```

6.7.4. La función `delete`

La función `delete` remueve una llave de el hash. Si no existe la llave, la función termina, y en ese caso causa una advertencia o un error.

```
1 my $person = "larry";
2 delete $books{$person};
```

Note que esto no es lo mismo que guardar un elemento como `undef` en el hash. En efecto, si chequea el hash con `exists($books{"betty"})` va a obtener verdadero, luego de usar `delete`, la llave no puede existir en el hash, pero después de guardar `undef` la llave va a existir en el hash.

6.7.5. Interpolación de un elemento en un Hash

Puedes interpolar un elemento simple de un hash colocándolo entre dobles comillas.

```
1 foreach $person (sort keys %books) {
2     if ($books{$person}) {
3         print "$person has $books{$person} items\n";
4     }
5 }
```

Pero no es posible interpolar el hash completo, si tratamos de usar “`%books`” simplemente obtendremos seis caracteres de (literalmente) `%books`.

6.7.6. El hash %ENV

Su programa, como cualquier otro programa, corre en un ambiente específico, de modo que el programa puede mirar en el ambiente para obtener información de sus alrededores. Perl guarda esa información en el hash %ENV. Por ejemplo, podemos ver el PATH de la siguiente forma:

```
1 print "PATH is $ENV{PATH}\n";
```

Dependiendo de su sistema operativo particular, va a obtener algo parecido a esto:

Terminal
PATH is /Users/elsanto/.vim/bin/:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin

6.8. Ejercicios

1. Escriba un programa que pregunte al usuario por un nombre e imprima correctamente su apellido. Use los nombres de las personas que usted conozca, o use la tabla siguiente:

Terminal

fred	flinstone
barney	rubble
wilma	flinstone

2. Escriba un programa que lea una serie de palabras (una palabra por línea) hasta el final de la entrada (End-Of-Input), e imprima un resumen de cuantas veces fue vista cada palabra.
3. Escriba un programa que liste todas las llaves de los valores en el hash %ENV. Imprima los resultados en dos columnas en orden ASCIIbetico. Una vez que tenga el programa funcionando, intente definir nuevas variables de entorno y asegúrese que estas variables están en la salida de su programa.

7 El mundo de las Expresiones Regulares

Perl posee muchas características que lo distinguen de otros lenguajes. De todas estas características una de las mas importantes es el soporte solido de expresiones regulares. Este permite un manejo de cadenas rápido, flexible y confiable.

Este poder tiene un precio. Las expresiones regulares actualmente son pequeños programas con su propio lenguaje especial, metidos dentro de Perl. (Si, esta pensando correctamente, va a aprender otro lenguaje de programación.)

En este capítulo, va a visitar el mundo de las expresiones regulares, en donde probablemente podrás olvidar algunas cosas del mundo de Perl. Luego en un capítulo siguiente, vamos a mostrarle donde es que este mundo encaja en el mundo de Perl.

Las expresiones regulares no son exclusivas de Perl, podemos encontrarlas en `sed` y `awk`, `procmail`, `grep`, editores de texto como `vim` e `emacs`.

7.1. ¿ Qué son las expresiones regulares ?

Una expresión regular, también llamada *patrón* en Perl, es una plantilla en donde una cadena determinada encaja o no encaja. Esto es, hay un número infinito de cadenas de texto posibles; con un patrón de terminado que las divide en dos grupos: las cadenas que encajan o coinciden, y aquellas que no.

Un patrón puede aplicar a una sola cadena, o solo dos o tres, o a una docena, o a cientos, o a un número infinito. O puede aplicar a todas excepto a una, o excepto a un grupo, o excepto a un número infinito.

Nos referimos a las expresiones regulares como pequeños programas con su propio lenguaje de programación simple. Es un lenguaje simple porque los programas tienen una sola tarea: mirar una cadena y decir “Cumple, esta cadena encaja en el patron” o “No cumple, esta cadena no encaja en el patrón”. Y esto es todo lo que hace.

Uno de los lugares donde probablemente halla visto las expresiones regulares es en el comando de Unix `grep`. Por ejemplo:

```
Terminal
$ grep 'flint.*stone' chapter*.pod
chapter_02.pod:      ($fred, $barney) = qw< flintstone rubble slate granite >;
chapter_02.pod:      ($wilma, $dino) = qw[flintstone];          # $dino obtiene undef.
chapter_02.pod:      @rocks = qw{ flintstone slate rubble };
chapter_05.pod:      $apellido{"fred"}      = "flintstone";
chapter_05.pod:      "fred"                  => "flintstone",
```

No debe confundir las expresiones regulares con los patrones de archivo de la shell llamados globs. Un glob típico es el que usas cuando escribes `*.pm` en la consola de Unix para coincidir con todos los archivos que terminen en `.pm`.

7.2. Usando Patrones Simples

Para verificar una cadena contra un patrón o expresión regular contenida en la variable `$_`, simplemente colocamos la cadena entre un par de barras, por ejemplo:

```

1     $_ = "yabba dabba doo";
2     if (/abba/){
3         print "Matched ! \n";
4     }

```

La expresión `/abba/` busca estas cuatro letras en la cadena contenida en `$_`, si la encuentra retorna un valor verdadero. En este caso, fue encontrada en mas de una oportunidad, pero eso no causa ninguna diferencia. Si la encuentra, coincide o hace match; si no la encuentra en toda la expresión, falla.

Como evaluar una expresión regular devuelve verdadero o falso, esta evaluación normalmente se encuentra en expresiones condicionales `if` o `while`.

Todas las secuencias de escape usuales que pueda colocar entre dobles comillas están disponibles en los patrones de expresiones regulares, entonces podría usar el patron `/coke\tsprite/` para probar la coincidencia de 11 caracteres de coke, un tab, y sprite.

7.3. Sobre los Metacaracteres

Si los patrones solo verificaran cadenas de caracteres simples, nos serían muy útiles en verdad. Esto es porque hay una número de caracteres especiales, llamados *metacaracteres* que tienen un significado especial en las expresiones regulares.

Por ejemplo, el punto (.) es un carácter comodín, hace match de cualquier carácter excepto de una nueva linea. Entonces el patrón `/bet.y/` va a hacer match para la cadena `betty`. O va a hacer match de `betsy`, o de `bet=y` o de cualquier otra cadena que tenga `bet` seguida por cualquier carácter (excepto una nueva linea) seguida por una `y`. El punto siempre hace match de un solo carácter.

Si quisiera usar el punto en un patrón, para evaluar la existencia de el punto en la cadena, debe anteponer un backslash o barra invertida. Esta regla aplica para todos los metacaracteres de expresiones regulares en Perl.

7.4. Cuantificadores Simples

Con frecuencia, tendrá algo que se repite en un patrón. El asterisco (*) indica hacer match de el item que lo precede cero o mas veces. Entonces, `/fred\t*barney/` hace match para cualquier número de caracteres tabs entre `fred` y `barney`. Entonces, va a hacer match para `"fred\tbarney"` con un tab, `"fred\t\tbarney"` con dos tabs, o `"fred\t\t\tbarney"` con tres tabs, o de `"fredbarney"`.

Para hacer referencia a cualquier grupo de caracteres, usamos el punto y el asterisco. Entonces, `.*` va a hacer match de cualquier carácter, para cualquier número de veces.

Al asterisco (*) se le llama formalmente *cuantificador*. Pero no es el único cuantificador; el signo (+) es otro. El signo + indica que va a hacer match de el carácter anterior una

o mas veces: `/fred +barney/` hace match si fred y barney esta separados por uno o mas espacios y solo espacios. (El espacio, no es un metacaracter). Esto no hace match de `fredbarney`, debido a que el signo (+) indica que debe haber uno o mas espacios entre los dos nombres, entonces el ultimo espacio es requerido. Puede ayudarle a recordar que el signo (+) indica que el carácter que antepone al signo mas (+) puede estar opcionalmente repetido.

Existe un tercer cuantificador como el asterisco (*) y el signo mas (+), pero un poco mas limitado. Es el signo de interrogación cerrando (?), el cual indica que el elemento que se encuentra antes de él es opcional. Esto es, el item que le precede, puede ocurrir una o ninguna vez. Entonces, si tenemos `/bamm-?bamm/` hace match para `bamm-bamm` o para `bambamm`. Esto es fácil de recordar, diciendo: “La última cosa, ¿ puede estar ?, ¿ o no puede estar ?”.

7.5. Agrupación de Patrones

Como en matemáticas, los paréntesis (()) los usábamos para agrupar. Entonces, los paréntesis son metacaracteres. Ejemplo, el patron `/fred+/` hace match de cadenas como `freddddddddd`, pero, este tipo de cadenas no son del mundo real. Ahora, el patrón `/(fred)+/` hace match de cadenas como `fredfredfred`, que es algo mas parecido a lo que tal vez tu quisieras. ¿ Y que pasaría con el patrón `/(fred)*/` ?, esto va a hacer match para cadenas como “Hola, Mundo”.

Los paréntesis también nos dan la posibilidad de reusar parte de la cadena que hizo match directamente. Podemos usar puntos de referencias para acceder a una cadena que hizo match en los paréntesis. Un punto de referencia se escribe como una barra invertida seguida por un número, como `\1`, `\2`, `\n`. El número indica el grupo creado con los paréntesis.

Cuando usamos paréntesis al rededor de un punto, vamos ha hacer match de cualquier carácter que no sea una nueva linea. Podemos hacer match de nuevo al carácter que hizo match entre los paréntesis usando el punto de referencia `\1`.

```

1      $_ = "abba";
2      if (/(\.)\1/){    # Hace match de 'bb'
3          "Esto hace match para un carácter seguido de si mismo. \n";
4      }
```

El punto de referencia no tiene que estar precisamente justo a la derecha de el grupo de paréntesis. El siguiente patrón, hace match de cualquier caracteres que no sea una nueva linea después del literal `y`, y luego usamos el punto de referencia `\1` para indicar que queremos hacer match de los mismos cuatro caracteres después de el literal `d`:

```

1      $_ = "yabba dabba doo";
2      if (/y(....) d\1/){
3          print "Hace match del conjunto de caracteres que están antes y después
4              de el caracteres literal d! \n";
5      }
```

Podemos usar multiples grupos de paréntesis, y cada grupo posee su propio punto de referencia. Podemos hacer match de cualquier carácter que no sea una nueva linea en

grupos de paréntesis, seguido por cualquier otro carácter que no sea una nueva línea en otro grupo de paréntesis. Después de tener estos dos grupos, usamos \1 para referirnos al primer grupo y \2 para referirnos al segundo grupo. En efecto, con el próximo ejemplo, va ha hacer match si se encuentra el palindrome abba:

```

1     $_ = "yabba dabba doo";
2     if (/y(.)\2\1/){      # Hace match de 'abba'
3         print "Hizo match para los caracteres después de d y y \n";
4     }

```

Ahora, se debe estar haciendo una pregunta, ¿ Como se sabe que grupo obtiene que número ?. Afortunadamente, Larry hizo que esto fuera fácil de comprender para los humanos: simplemente cuente el orden de apertura de los paréntesis.

```

1     $_ = "abba dabba doo";
2     if (/y((.)\3\2) d\1/){
3         print "Matched\n";
4     }

```

Probablemente es mas fácil que lo vea, si escribe la expresión regular que forma que se observe cada parte por separado:

```

1     (          # Primera apertura de paréntesis
2         (.)    # Segundo apertura de paréntesis
3         (.)    # Tercera apertura de paréntesis
4         \2
5         \3
6     )

```

En Perl 5.10 existe una nueva forma para denotar un punto de referencia. En lugar de usar un backslash y un número, usamos \g{N}, donde N es el número del punto de referencia que desea usar.

Supongamos que debe usar un punto de referencia como parte de un patrón que contiene un número. En esta expresión regular, nosotros vamos a usar \1 para repetir el carácter que hizo match en los paréntesis seguido por la cadena literal 11:

```

1     $_ = "aa11bb";
2     if (/(\.)\111/){
3         print "Matched ! \n";
4     }

```

Perl va a tener que adivinar que significa eso. ¿ Es un \1, \11, o \111 ?. Perl crea referencias como sea necesario, entonces el asume que eso significa \111. Pero no tenemos 111 grupos de paréntesis, o 11 grupos, Perl va a quejarse cuando intente compilar el programa.

Con la notación \g{N}, evitamos la ambigüedad entre la referencia y las partes literales del patrón.

```

1     use 5.010;
2
3     $_ = "aa11bb";
4     if (/(\.)\g{1}11/){
5         print "Esto hace match \n";
6     }

```

Con la notación `\g{N}`, podemos usar números negativos. En lugar de especificar un número absoluto del grupo de paréntesis, podemos especificar un punto de referencia relativa. Podemos reescribir el ejemplo anterior usando `-1`.

```

1      use 5.010;
2
3      $_ = "aa11bb";
4      if (/(.)\g{-1}11/){
5          print "Esto hace match";
6      }

```

7.6. Alternación

La barra vertical (`|`), normalmente llamada “or”, indica que, o hace match el lado derecho de la expresión o, hace match el lado izquierdo. Esto significa, que si la parte izquierda del patrón falla, la parte de la derecha aun tiene una oportunidad par hacer match. Entonces, `/fred|barney|betty/` va a hacer match con cualquier cadena que contenga `fred`, o `barney`, o `betty`.

Ahora, usted puede hacer patrones como `/fred(|\t)+barney`, que hace match si `fred` y `barney` están separados por espacios, tabs, o una mezcla de ambos.

El patrón `/fred (and|or) barney/` hace match con cualquier cadena que tenga una de las dos posibles cadenas: `fred and barney` o `fred or barney`.

7.7. Clases o Grupos de Caracteres

Una clase o grupo de caracteres, es una lista de caracteres posibles dentro de corchetes (`[]`), entonces cualquier caracterer que este especificado en la clase va hacer match. Esto hace match solo para un simple caracterer, y este caracterer debe estar en la lista declarada entre los corchetes.

Por ejemplo, para la clase `[abcwxyz]` va a hacer match cualquiera de estos siete caracteres. Por conveniencia, puedes especificar un rango de caracteres usando el guión (`-`). Entonces se puede escribir la anterior clase de la siguiente manera: `[a-cw-z]`.

Es usual encontrar esta clase de caracteres: `[a-zA-Z]` para hacer match de cualquier letra que se encuentre en este grupo.

Por supuesto, una clase de caracteres va a ser siembre parte de un patrón. Por ejemplo

```

1      $_ = "The HAL-9000 requires authorization to continue.";
2      if (/HAL-[0-9]+)/ {
3          print "The string mentions some model of HAL computer.\n";
4      }
5

```

Algunas veces, es mas fácil especificar que grupo de caracteres no va a hacer match, en lugar de usar una clase de caracteres permitida. El acento circunflejo `^` al inicio de una clase de caracteres, niega esta clase de caracteres. Esto significa, que `[^def]` va a hacer match con cualquier caracter simple excepto para uno de estos tres. Y `[^n\~z]` va hacer match con cualquier caracter excepto con `n`, guión (`-`) o `z`

7.8. Atajos para Clases de Caracteres

Algunas clases de caracteres aparecen con tanta frecuencia que tienen un atajo. Por ejemplo, la clase de caracteres de cualquier dígito, `[0-9]`, puede ser abreviado como `\d`. Entonces, el patrón para ejemplo sobre HAL, pudo haberse escrito así: `/HAL-\d+/.`

El atajo `\w`, representa la clase: `[A-Za-z]`

El atajo `\s`, representa espacio en blanco. Es común ver `\s*` para cualquier cantidad de espacios en blanco, incluyendo ninguno.

7.9. Atajos Negativos

Algunas veces usted va a querer lo opuesto a uno de los atajos. Puede ser que quiera `[^\d]`, `[^\w]` o `[^\s]`, indicando, ningún caracter de tipo dígito, o ningún caracter de palabra, o ningún espacio en blanco. Esto es bastante fácil de lograr usando sus homólogos en mayúsculas: `\D`, `\W`, o `\S`.

Otra composición de clase de caracteres es `[\d\D]`, que contempla cualquier dígito y cualquier no dígito incluso una nueva línea. Y esta el patrón totalmente innecesario: `[^\d\D]`, que hace match de cualquier cosa que no sea ni un dígito ni un no dígito. Es decir, Nada.

7.10. Ejercicios

1. Escriba un programa que imprima cada línea de la entrada que mencione la palabra **fred**. Haga un archivo de texto que contenga la palabra **fred** en cadenas y nombres que incluyan **fred** y utilice este archivo como entrada del programa.
2. Modifique el programa anterior, para que también haga match con **Fred**.
3. Escriba un programa que imprima cada línea de su entrada que contenga el punto (**.**), ignore las otras líneas.
4. Escriba un programa que imprima cada línea que tenga la palabras con la primera letra en mayúscula, pero no mayúscula sostenida.
5. Escriba un programa que imprima cada línea que tenga dos caracteres iguales juntos el uno al otro y que no sea dos espacios en blanco. Debe hacer match de líneas que contengan cosas como **Mississippi**, **Bamm-Bamm**, o **llama**.
6. Escriba un programa que imprima cualquier línea de la entrada que contenga los siguientes dos cadenas en la misma línea: **willma** y **fred**.