Programación en Shell (bash/Bourne)

Gabriel Saldaña — gsaldana@gmail.com
Gunnar Wolf — gwolf@gwolf.org

Agosto 2015

Resumen

En este tutorial veremos desde los aspectos básicos del Shell hasta cómo utilizarlo como un lenguaje completo de propósito general.

Asumimos que el asistente al tutorial tiene conocimientos básicos de uso de un sistema Unix, si bien puede nunca haberse internado al maravilloso mundo de la programación.

Índice

1. Introducción

1.1. Las computadoras antiguas

Para poder comprender la funcion del shell es necesario comprender el proceso historico del computo, desde las primeras computadoras hasta los sistemas Unix modernos.

Las primeras computadoras, en los 40s y 50s, no tenian contemplada la ejecucion de mas que un solo programa — Eran programadas en un principio fijando a mano interruptores, y fue un muy gran avance para su usabilidad la introduccion de las primeras lectoras de tarjetas perforadas, pues, ademas de reducir sensiblemente el tiempo muerto en que un programa tenia que ser introducido, permitia guardarlo para uso futuro de una manera conveniente.

1.1.1. Procesamiento por lotes

Tras la aparicion de las lectoras de tarjetas, fue solo cuestion de tiempo el que se serializaran procesos en lotes — Cada usuario dejaba listo su programa en la lectora, la cual lo alimentaba a la computadora, esperaba a que ejecutara e imprimiera resultados, y continuaba con el siguiente proceso. Al automatizarse la carga, los tiempos muertos para la computadora se redujeron al minimo.

1.1.2. Concurrencia primitiva

Ahora, el uso del procesador se podia aprovechar aun mas: El tiempo en que la lectora alimentaba a la memoria central o en que se imprimian los resultados era, a todas luces, desperdiciado. Ademas, habia trabajos de alta prioridad, que debian esperar su turno como cualquier otro pese a su importancia.

Si bien antes de la concurrencia ya existian sistemas operativos pequenos y limitados encargados de una abstraccion basica del hardware, su rol como asignadores de recursos nace cuando hay varios programas simultaneos en ejecucion. Y al haber ya un complejo ambiente en el que diferentes usuarios desde diferentes consolas requieren cargar y ejecutar diferentes programas a la vez, nace la necesidad real de un programa base que permita al usuario especificar a la computadora que quiere hacer — un shell.

1.2. El shell como un lanzaprogramas

Un shell basico es simplemente un lanzaprogramas. Acepta ordenes del usuario, las cuales se traducen directamente en el nombre de un programa a ejecutar. Claro, es necesario proveer al usuario tambien de los comandos basicos para manejar los archivos en la computadora, ya que un sistema con concurrencia, para ser eficiente, requiere tener espacio de almacenamiento permanente para los programas —un disco— y, claro, proveer los mecanismos para administrarlo — crear, copiar, eliminar, compilar, etc.

1.3. El shell de hoy

Ha pasado ya mucho tiempo desde aquellos primitivos shells. En un sistema Unix moderno, el shell es ya un entorno completo de programacion, proporcionando al usuario todas las herramientas necesarias para automatizar la administracion de sistemas, facilitar diversas labores cotidianas, e incluso jugar con un ambiente de desarrollo agradable.

En Unix hay muchos diferentes shells, y los usuarios de cada uno de ellos lo defienden como el mejor con fervor religioso (lo cual no debe de sorprender a nadie). Las principales familias son el C Shell y el Bourne Shell; el Bourne Shell está estandarizado en POSIX 1003.2a.[1]

Cada una de estas familias tiene varias implementaciones, y para todos hay cuando menos una implementacion libre. En este tutorial nos enfocamos al *bash* (Bourne Again Shell), por ser el que mas gente encuentra como primera experiencia en un sistema Unix al ser el predeterminado de casi cualquier Linux, y por existir en todos los demas Unixes.

1.3.1. Shells mínimos

En entornos de sistemas embebidos es conveniente recordar que hay shells mucho más ligeros que *bash*, que compormeten algo de facilidades pero siguen

[1] `http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_title.html`

siendo plenamente programables. Destacan por su usabilidad y conveniencia dos de ellos: *dash* y *busybox*.

Dash (*Debian Alquimist Shell*) busca ser meramente un reemplazo ligero de bash; depende de menos bibliotecas, ejecuta con mayor velocidad, y es empleado por default en los sistemas Debian y derivados.

Busybox busca implementar un sistema completo mínimo, e incluye compilado en el mismo binario no únicamente el shell sino que los principales programas que se emplean en su uso diario (*ls*, *cp*, *mv*, *mount*, *tar*, etc.), aunque no implementa plenamente los estándares POSIX. *Busybox* es frecuentemente utilizado en sistemas de instalación u otros embebidos.

1.3.2. Los *bashismos*

Como programadores, nos conviene crear código *portable* a otros sistemas. No podemos asumir que siempre habrá un *bash* instalado, por lo que conviene escribir para cualquier shell que cubra los estándares POSIX correspondientes. En este tutorial no se entra en detalles acerca de los *bashismos*, pero se invita al lector a revisar el programa *checkbashisms* que forma parte del paquete *devscripts* de Debian GNU/Linux[2] y a la página Wiki relativa a esta migración en Ubuntu.[3]

2. Sintaxis general

Usar el shell como un lanzaprogramas es muy simple: Cada que le damos una linea, ejecuta el comando cuyo nombre le escribimos. Podemos indicar la ruta completa al archivo, o ejecutar los archivos que aparezcan en el path de ejecucion (ver seccion 2.3.2).

2.0.1. Flujos: STDIN, STDOUT, STDERR

En el shell —al igual que en cualquier programa de consola de Unix— tenemos tres flujos o descriptores de archivo abiertos por default: La entrada estandar (STDIN), la salida estandar (STDOUT) y el error estandar (STDERR). El primero puede ser utilizado para leer de el, y los otros dos para enviar datos hacia ellos. Tipicamente, STDIN viene del teclado de la terminal actualmente en uso, y tanto STDOUT como STDERR van hacia su pantalla. STDOUT muestra los datos normales o esperados durante la ejecucion, y STDERR se utiliza para enviar informacion de depuracion y errores. Cualquier programa iniciado desde el shell, a menos que se lo indiquemos explicitamente, hereda estos tres descriptores de archivo, permitiendole interactuar con el usuario.

[2] `https://anonscm.debian.org/cgit/collab-maint/devscripts.git/tree/scripts/checkbashisms.pl`

[3] `https://wiki.ubuntu.com/DashAsBinSh`

2.1. Más allá del lanzaprogramas

Si bien su principal mision es ser un lanzaprogramas, un shell tiene una funcionalidad mucho mayor. En Unix, muchas veces se utiliza al shell para interconectar programas independientes —recuerden la filosofia Unix, que nos da una gran cantidad de herramientas que hacen solamente una cosa simple, pero permiten interactuar con otros programas similares creando asi construcciones complejas y utiles. El shell ademas da facilidades al usuario como la expansion, el *globbing* y los aliases.

2.1.1. Redirecciones (>, <, >>, &>, &>>)

>, >> La salida de un programa muchas veces es util como tal enviada a pantalla. Sin embargo, muchas veces esta salida puede ser mucho mayor de lo que podemos manejar a ojo. Tal vez queremos generar, por ejemplo, el listado completo de archivos bajo cierta parte del arbol. Para enviar la salida estandar de un comando a un archivo usamos > (para borrar cualquier contenido previo del archivo) o >> (para agregar nuestros datos al final del archivo). Por ejemplo, `ls -R /usr > /tmp/archUsr` crea un archivo `/tmp/archUsr` con el listado de archivos bajo /usr.

< Con esto podemos indicar a un proceso que tome su entrada estandar de un archivo existente. Por ejemplo, si quiero ejecutar de manera automatica un proceso que siempre me pide confirmacion durante la ejecucion, puedo grabar en un archivo todos los comandos que este espera, e invocarlo asi: `/home/gwolf/miprog < /home/gwolf/miprog.comandos` (claro esta, esto puede ser peligroso. Cuando un proceso requiere confirmacion, normalmente es por alguna buena razon. Sin embargo, les pido que me lo valgan como ejemplo).

&>, &>> Esto es equivalente a lo que mencionabamos respecto a la salida estandar, pero aplicado al error estandar. Por ejemplo, si quiero revisar los errores resultantes de ejecutar un proceso determinado, puedo enviarlos a un archivo, de la siguiente manera: `tar czvf archivo.tar.gz /usr /var /lib &> erroresTar` me indicara en el archivo erroresTar cualquier problema que se me pueda presentar, como permisos incorrectos.

Ahora, cabe recordar aqui que en Unix los dispositivos son representados por y tratados como un archivo. Por tanto, `ls -l > /dev/ttyS0` enviara un listado de archivos al primer puerto serial.

2.1.2. Pipes (|, &|)

Muchas veces necesito pasar a un proceso la salida de otro. Por ejemplo, para contar la cantidad de lineas en un archivo sin repeticion, primero ordeno el archivo (con `sort`), despues elimino lineas duplicadas (con `uniq`) y por ultimo cuento las lineas (con `wc`). Si bien podria hacerlo de esta manera:

```
sort archivo > /tmp/ordenado
uniq /tmp/ordenado > /tmp>unico
wc /tmp/unico
```

esto claramente no es optimo. Puedo, mejor, *entubar* la salida estandar de un proceso y enviarlo al siguiente utilizando los *pipes*, simbolizados con el caracter `|`, de esta manera:

```
sort archivo | uniq | wc
```

Lo cual, ademas de mas compacto, es mas facil de leer y entender.

Si quiero redireccionar el error estandar en vez de la salida estandar, puedo usar `&|`.

2.1.3. Control de procesos (&)

Muchas veces iniciamos procesos, como la descompresion de un .tar con muchos archivos o la transferencia de un archivo remoto, que pueden tomar mucho tiempo y que no nos interesa la respuesta que puedan enviar a la consola, sino que su resultado final. Aprovechando que Unix es un sistema multitareas, podemos enviar cualquier comando que ejecutemos a un segundo plano agregando `&` al final de la linea de comando, de la siguiente manera:

```
$ wget http://iso.softwarelibre.org.mx/debian-2.2r5-1.iso
&
```

Podemos lograr este mismo efecto si, una vez lanzado el comando, lo suspendemos con `^Z` y lo enviamos a ejecucion en segundo plano con `bg` :

```
$ wget http://iso.sofwarelibre.org.mx/debian-2.2r5-1.iso
^Z
[1]+ Stopped wget http://iso.softwarelibre.org.mx/debian-2.2r5-1.iso
$ bg
[1]+ Stopped wget http://iso.softwarelibre.org.mx/debian-2.2r5-1.iso
&
$
```

2.1.4. *Globbing* y expansión ({}, ~)

El shell es un gran aliado cuando se trata de escribir menos. Hay varios mecanismos para ahorrarnos teclazos:

Globbing es probablemente el mas comun, el que todos conocemos — La expansion de nombres usando el caracter `*` , que significa "todo lo que puedas acomodar ahi". Por ejemplo, si en un directorio tengo los archivos `salida.tmp, asdf, otroarchivo` y `prueba`, y en ese directorio corro `cat *`, el shell lo interpretara como si hubiera escrito `cat asdf otroarchivo prueba salida.tmp` (en orden alfabetico). Si le pongo `cat o*`, procesara unicamente los archivos que inician con o, y dara por tanto `cat otroarchivo`.

La expansion trabaja con argumentos mucho mas claramente definidos que el globbing. La expansion esta claramente hecha para ahorrar teclazos y hacer mas inteligente al shell.

Cuando especificamos una lista de valores separada por comas entre llaves, el shell la expande, convirtiendola en la cadena con cada uno de los argumentos. Por ejemplo,

```
echo un/path/{algo,muy,demasiado}/largo
```

Nos produce la siguiente salida:

```
un/path/algo/largo un/path/muy/largo un/path/demasiado/largo
```

Ahora, tenemos que acordarnos de un par de reglas del juego:

- La expansión va sobre una sóla palabra. Si ponemos:

  ```
  echo un texto {muy,algo} confuso
  ```

  obtendremos como resultado:

  ```
  un texto muy algo confuso
  ```

- La expansión no se efectúa dentro de comillas, ni siquiera dobles (ver 2.3.3), por lo que no sirve que intentemos corregir de esta manera el ejemplo anterior:

  ```
  echo "un texto {muy,algo} confuso"
  ```

  pues nos dará por resultado:

  ```
  un texto {muy,algo} confuso
  ```

- Si escapamos los espacios, dejan de ser delimitadores y se convierten en parte de una palabra. Por tanto, podemos hacer:

  ```
  echo un\ texto\ {muy,algo}\ confuso
  ```

  y obtendremos:

  ```
  un texto muy confuso un texto algo confuso
  ```

- Podemos tener múltiples expansiones en una sóla línea. Estas se expandirán generando todas las posibles combinaciones:

  ```
  echo {Un,Otro}\ texto\ {muy,algo}\ confuso.
  ```

  nos da:

  ```
  Un texto muy confuso. Un texto algo confuso. Otro texto muy
  confuso. Otro texto algo confuso.
  ```

2.1.5. Aliases

Podemos indicarle al shell que cada que le demos determinada cadena la substituya por otra. Para esto utilizamos el comando interno `alias` .

Cuando el shell ejecuta cualquier comando, revisa si la *primera* palabra de cada comando que le demos, y si la encuentra en su tabla de aliases (y el operador no requiere que se inteprete literalmente usando comillas) la substituye antes de continuar procesando la línea.

Para crear un alias podemos hacerlo de la siguiente manera:

```
alias cosa='ls -l'
```

con lo que cada que indiquemos el comando `cosa` el shell ejecutará `ls -l` . Ahora, si le pedimos `echo cosa` , como no es la primera palabra, nos va a regresar a secas `cosa` .

Ahora, si indicamos lo siguiente:

```
echo `cosa`
```

nos va a dar el *resultado de ejecutar* `ls -l`. Recuerda que lo que encerremos en comillas inversas (ver 5.2) es ejecutado en un sub-shell, y aquel sub-shell ve a `cosa` como la primera palabra del comando.

Al tener los aliases precedencia aún sobre los comandos internos del shell, son muy útiles para ahorrarnos teclazos. Por ejemplo, mucha gente tiene los siguiente aliases definidos:

```
alias ls='ls -color'
alias rm='rm -i'
alias mv='mv -i'
```

con lo que sin tener que agregarle opciones, siempre le pasaremos `-color'` al `ls` , y siempre eliminaremos y moveremos requiriendo confirmar.

Para consultar qué aliases tenemos definidos, damos `alias` sin argumentos.

Para eliminar un alias, damos `unalias`.

2.2. Bourne y bash vs. otros shells

En Unix tenemos muchos diferentes shells. ¿Por qué recomendamos elegir a los derivados del Bourne?

2.2.1. Edición de comandos

2.2.2. Historia

2.3. Variables y cadenas

Prácticamente cualquier lenguaje de programación nos proporciona variables con las que podemos trabajar. En shell manejarlas sigue unas reglas un tanto particulares.

2.3.1. Variables en general

En shell, los nombres de variables pueden contener letras, números y guiones bajos. Si bien ninguna regla lo marca, es una convención muy común que los nombres de las variables vayan completamente en mayúsculas. No es necesario declarar de qué tipo será cada variable (al shell le da igual si las variables guardan cadenas o números), pero sí tenemos que comprender bien cómo manejarlas, pues es una fuente muy frecuente de errores el referirnos a una variable cuando requerimos su contenido, o a la inversa.

Para asignar un valor a una variable lo hacemos de esta manera:

```
VARIABLE=valor
```

Cuando queramos utilizar el valor de la variable podemos hacerlo anteponiendo a su nombre un signo `$` :

```
echo $VARIABLE
```

Siempre que queramos imprimir, comparar, hacer cuentas o en general utilizar el valor *contenido* en la variable, lo haremos refiriéndonos a su valor con `$` . Siempre que queramos indicar al shell que haga algo *utilizando la variable*, lo haremos refiriéndonos a su nombre, sin `$` .

2.3.2. Variables de ambiente

2.3.3. Cadenas e interpolación

3. Control de flujo

Como en cualquier lenguaje de programación, un 'script' de shell tiene que poder decidir que acciones tomar según el resultao operaciones anteriores. Además, es necesario automatizar el repetir acciones, ya sea un número fijo de veces, o hasta que se cumpla alguna condición.

3.0.1. Estados de salida

Puesto que la función primaria del shell es ejecutar a otros programas, resulta natural desear controlar la ejecución de un script de acuerdo al resultado de la ejecución de estos. Para lograr eso, cada programa que se ejecuta en un sistema UNIX devuelve al programa que lo ejecutó un número, que representa el resultado obtenido. Si el programa se ejecutó sin errores, devuelve un cero. Si hubo algún error, devuelve un número distinto de cero. Este número depende del error especifico, y por lo tanto varía de programa a programa. Siempre que el shell necesita tomar una decisión basada en el resultado de otro programa, considera como 'cierto' a un valor 0, y como 'falso' a cualquier otro.

3.1. Operadores

3.1.1. Booleanos (&& || !)

Los operadores booleanos permiten combinar los resultados de varias pruebas. Funcionan de forma identica a los de C. En particular, `&&` y `||` evaluan sus argumentos de izquierda a derecha, deteniendose en cuanto se sabe el resultado total. Esto permite efectuar combinaciones de control sencillas. Por ejemplo, cuando se compila e instala un paquete de software, es común dar el comando `make` seguido de `make install`, si no hubieron errores. Es posible automatizar esta secuencia, dando el comando `make&&make install` que efectuara al segundo solo si el primero termina sin errores.

El operador `!` invierte el sentido del valor de retorno de un programa.

Estos operadores se usan para encadenar pruebas, en particular aquellas que involucran el estado de retorno de un programa.

3.1.2. De archivo (test -*x*)

Para efectuar otras pruebas el shell provee el operador `test`. Este permite realizar una serie de pruebas acerca del sistema de archivos, así como pruebas que dependan del valor de las variables del shell. Estos operadores son:

De archivo:

```
-b ARCHIVO
Verdadero si el ARCHIVO es un dispositivo de bloque.
-c ARCHIVO
Verdadero si el ARCHIVO es un dispositivo de caracteres.
-d ARCHIVO
Verdadero si el ARCHIVO es un directorio.
-e ARCHIVO
Verdadero si el ARCHIVO existe.
-f ARCHIVO
Verdadero si ARCHIVO existe y es un archivo normal.
-g ARCHIVO
Verdadero si el ARCHIVO tiene encendido el bit sgid.
-h ARCHIVO
-L ARCHIVO
Verdadero si el ARCHIVO es un vínculo simbolico.
-k ARCHIVO
Verdadero si el ARCHIVO tiene encendido el bit 'sticky'.
-p ARCHIVO
Verdadero si el ARCHIVO es un 'named pipe'.
-r ARCHIVO
Verdadero si el ARCHIVO es legible por este usuario.
-s ARCHIVO
Verdadero si el ARCHIVO existe y es no vacio.
-S ARCHIVO
```

```
Verdadero si el ARCHIVO es un 'socket'.
-t FD
Verdadero si el descriptor FD esta abierto a una terminal.
-u ARCHIVO
Verdadero si el ARCHIVO tiene prendido el bit 'suid'.
-w ARCHIVO
Verdadero si el usuario tiene permiso de escribir en el ARCHIVO.
-x ARCHIVO
Verdadero si el usuario tiene permiso de ejecutar el ARCHIVO.
-O ARCHIVO
Verdadero si el usuario es el dueño del ARCHIVO.
-G ARCHIVO
Verdadero si el ARCHIVO pertenece al grupo del usuario.
-N ARCHIVO
Verdadero si el ARCHIVO ha sido modificado desde la última lectura.
ARCHIVO1 -nt ARCHIVO2
Verdadero si el ARCHIVO1 es más nuevo que el ARCHIVO2
(de acuerdo a la fecha de modificación).
ARCHIVO1 -ot ARCHIVO2
Verdadero si el ARCHIVO1 es más viejo que el ARCHIVO2.
ARCHIVO1 -ef ARCHIVO2
Verdadero si el ARCHIVO1 es un vínculo duro al ARCHIVO2.
```

Operadores de cadenas:

```
-z CADENA
Verdadero si la CADENA es vacia.
-n CADENA
CADENA
Verdadero si la CADENA es no vacia.
CADENA1 = CADENA2
Verdadero si las cadenas son iguales.
CADENA1 != CADENA2
Verdadero si las cadenas son distintas.
CADENA1 < CADENA2
Verdadero si la CADENA1 va antes que la CADENA2 en orden lexicográfico.
CADENA1 > CADENA2
Verdadero si la CADENA1 va despues que la CADENA2 en orden lexicográfico.
```

Otros operadores:

```
-o OPCION
Verdadero si la opcion del shell OPCION esta activada.
! EXPR
Verdadero si la EXPR es falsa.
EXPR1 -a EXPR2
Verdadero si ambas expresiones son verdaderas.
```

```
EXPR1 -o EXPR2
Verdadero si alguna expresión es verdadera.
arg1 OP arg2
Pruebas aritméticas. OP es uno de -eq, -ne, -lt, -le, -gt, o -ge.
```

Las pruebas aritméticas regresan verdadero si `ARG1` es igual, distinto, menor que, menor o igual, mayor que, o mayor o igual que `ARG2`, respectivamente.

Existe otra sintaxis para el operador `test`. En ves de usar esta palabra clave, se puede encerrar la prueba entre un par `[...]`. Es importante que ambos parentesis queden aislados, para que el shell los reconozca como unidades independientes. Una prueba de este estilo se puede encadenar con cualquier otra por medio de los operadores de la sección anterior. Por ejemplo, si queremos asegurarnos que un programa existe antes de intentar ejecutarlo, podemos usar la siguiente construcción:

```
[ -f /usr/bin/true -a -x /usr/bin/true ] && /usr/bin/true
```

Que prueba que el archivo existe, es un archivo regular y es ejecutable antes de llamarlo.

3.2. Listas

La primera manera de controlar el flujo de un script de shell, es mediante la ejecución secuancial, es decir, el efectuar acciones una tras otra. Una lista es una secuencia de "tubos", separados por alguno de los operadores `&&`, `||`, `&`, `;` y terminados por `;`, `&`, o un fin de linea. De entre estos, `&&` y `||` tienen mayor precedencia, seguidos por `&` y `;`. Los dos primeros tienen el efecto que se describió en la sección anterior.

Cuando dos secuencias están separadas por `;` el efecto es que se efectuan ambas una tras la otra, sin importar el resultado de las anteriores. El resultado de la lista es el resultado de la última.

Cuando una secuencia está terminada por `&` el shell la ejecuta en el fondo, sin esperar a que termine. El resultado de la lista es 0 (verdadero).

```
ls -R /usr >/tmp/listado; wc -l /tmp/listado
```

Esta secuencia genera un listado de todos los archivos bajo `/usr`, lo guarda en `/tmp/listado` y, cuando termina, cuenta cuantas lineas se generaron.

3.3. Condicionales

Presentamos a continuación las sentencias de control de flujo del shell. Estas nos permiten elegir ejecutar una u otra opción dentro de un script, de acuerdo al resultado de alguna prueba.

3.3.1. if

La sentencia `if` nos permite elegir entre dos alterntativas de acuerdo al resultado verdadero o falso de una prueba. Tiene la sintaxis:

```
if lista then lista [ elif lista then lista ] ... [ else lista ] fi
```

Se ejecuta la primera lista. Si el resultado de esta es verdadero, se ejecuta la segunda. Si no, si hay presente una sección `elif` se efectua esa prueba, y si es exitosa se efectua la lista asociada. Si ninguna de las secciones `if` o `elif` son exitosas, y hay una sección `else`, se efectuan los comandos dados en esta. Si bien esta sintaxis se presenta en una sola línea, no es necesario escribirla así, y de hecho, al escribir un script es común separarlo en varias lineas, y además indentar el código.

```
if [ ! -f $HOME/.example.conf ] then
    echo Ejecutando por primera vez
    ./configura
else
    echo Ejecutando por segunda vez
    ./corre
fi
```

Recuerden que tambien se puede poner un comando arbitrario como prueba:

```
if ! grep 'From:.*Juanita' $MAIL; then
    echo Juanita no ha escrito
else
    mail juanita -s 'gracias por tu carta'
fi
```

3.3.2. case

La sentencia `case` nos permite elegir una entre varias alternativas, dependiendo del valor de una expresión. La sintaxis es:

```
case palabra in [ patrón [| patrón ]... ) lista ;; ] ... esac
```

La palabra es expandida de acuerdo a las reglas usuales (expansión de variables (2.3.2), comillas inversas (5.2), etc.) y luego se le compara con cada patrón, en el orden en que aparecen. Cada opción puede tener más de un patrón asociado. El ajuste es el usado para los nombres de archivos. Cuando alguno de los patrones concuerda, se ejecuta la lista asociada, y la busqueda de concordancias se

detiene. El resultado total del comando es 0 si ningún patrón ajustó, o el valor de la lista si sí.

Si bien la sentencia `case` no proporciona una acción por defecto, es fácil proporcionar una, poniendo al final de la lista un patrón que ajuste contra cualquier cosa, es decir `*)`.

```
case $USER in
    gabriel|gunnar)
        echo Bienvenidos
        /usr/local/bin/root-shell
        ;;
    juan)
        echo Tu no eres bienvenido
        /usr/local/bin/exit
        ;;
    *)
        echo No te conozco
        ;;
esac
```

3.4. Ciclos

Además de permitir elegir entre varias opciones, el shell nos permite repetir una secuencia de acciones un cierto número de veces, ya sea fijo o determinado por el cumplimiento de una condición.

3.4.1. while y until

Estas dos sentencias nos permiten efectuar repetidamente una acción, hasta que alguna condición dada se cumpla. La sintaxis es:

```
while lista do lista done
until lista do lista done
```

`while` repite la segunda lista siempre que la primera regrese verdadero. `until` la repite mientras sea falso. El valor de retorno de ambas es el de la lista ejecutada, o cero si no se ejecuto nada.

```
until who | grep -q gunnar; do
    echo Gunnar no ha llegado
done
```

3.4.2. for

La sentencia `for` nos permite hacer ciclos sobre listas de valores definidos antes de entrar al ciclo. Puesto que el ciclo no se limita a una cantidad de iteraciones, sino que la lista puede ser generada por otro comando, tiene una gran cantidad de usos. La sintaxis es:

```
for nombre [ in palabras; ] do lista; done
```

La lista de palabras se expande de acuerdo a las reglas usuales. La variable nombre se ajusta a cada valor de la lista resultante en turno, y se efectua la lista para cada valor. Si se omite la parte `in`, el ciclo se efectua para cada uno de los parametros posicionales (4.1.1) del script.

El siguiente ciclo entra a cada subdirectorio del directorio actual, y borra todos los archivos *.tmp dentro del mismo.

```
for file in *; do
    if [ -d $file ] then
        cd $file
        rm *.tmp
        cd ..
    fi
done
```

Este otro ciclo imprime un texto cinco veces:

```
for i in 1 2 3 4 5; do
    echo texto
done
```

4. Funciones

Como en todo lenguaje de programación, es conveniente encapsular acciones complejas que se repiten varias veces. Para esto, el shell permite definir funciones, que encierran varias acciones y les dan un nombre por el que pueden ser invocadas. La sintaxis para hacer esto es:

```
[function] nombre () { lista; }
```

que asocia al nombre con la lista de comandos entre llaves. Los comandos no se ejecutan en el momento de la definición. Notese que la palabra clave `function` es opcional, y solo sirve para documentar el script. La funcion es llamada especificando el nombre, como si fuera un comando sencillo.

```
busca_gunnar() { who | grep -q gunnar; }
if busca_gunnar then
    mail gunnar
fi
```

4.1. Parámetros

Por supuesto, es deseable que la ejecución de una funcion, o de un script, varien de acuerdo a parametros que se proporcionen al momento de ser ejecutados. Para esto, el shell reserva algunos nombres de variables especiales, cuyos valores son ajustados de forma automática.

4.1.1. Parámetros posicionales

El primer tipo de variables de este estilo, son los parametros posicionales. Estos son los parametros que se dan el script o funcion al momento de ser llamados. Se utilizan las variables especiales `$digitos`, por ejemplo `$1`, `$2`, etc. Estas variables reciben el valor del parametro pasado en la posición que su número indica. Para referir a los parametros mas allá del `$9`, se debe encerrar al número entre llaves: `${10}`, de lo contrario, el shell lo interpreta como el parametro `$1`, seguido de la cadena '0'. No es posible asignar valores a estos parametros.

```
busca_alguien() { who | grep -q $1; }
if busca_alguien gunnar || busca_alguien gabriel; then
    echo Ya llegaron
fi
```

O, ejemplificando la sintaxis alterna de `for`:

```
borra_en_dir() {
    for dir; do
        if [ -d $dir ] then
            cd $dir
            rm *.tmp
            cd ..
        fi
    done;
}
borra_en_dir trabajo tmp casa
```

4.1.2. Parámetros especiales — *, @, #, ?, -, $, !, 0, _

Además de los parametros posicionales, hay otras variables especiales que el shell mantiene. Los valores de estas son asignados automáticamente, y no es posible modificarlos. Describimos a continuación sus funciones.

`*` Expande a la lista de parametros posicionales. Cuando la expansión ocurre dentro de comillas dobles, expande a una sola palabra, que consta de los parametros posicionales separados por el primer caracter de la variable

`IFS`. Si esta variable no tiene un valor, se les separa con espacios. Es util para pasar la lista completa de argumentos recibidos a otro programa. Notese que el otro programa recibirá un solo argumento.

`@` Expande a la lista de parametros posicionales. A diferencia del anterior, cuando la expansión ocurre dentro de comillas dobles, expande a una palabra para cada parametro posicional. En esta forma, se preserva la lista de argumentos tal a como fué dada a este script, y se puede pasar a otro programa tal cual.

`#` Expande al número de argumentos posicionales.

`?` Expande al código de salida del último comando ejecutado.

`-` Expande a la lista de opciones del shell activadas, ya sea al ser invocada, o por el comando interno `set`.

`$` Expande al identificador de proceso del shell.

`!` Expande al identificador de proceso del último comando ejecutado en el fondo.

`0` Expande al nombre del shell, o del script.

`_` Expande al último argumento del último comando ejecutado.

5. Asuntos esotéricos en el shell

Describimos a continuación algunas de las caracteristicas más "esotericas" del shell. Si bien estas no son necesarias para el trabajo diario con el mismo, resulta útil conocerlas, puesto que permiten realizar cosas que son difíciles o imposibles de otra manera.

5.1. Here Documents (<<*algo*)

En ocasiones queremos ejecutar un comando con una entrada fija, pero no queremos poner ésta en un archivo desde el cual redireccionar la entrada. Ponerla en un archivo nos obliga a mantenerla en una ruta fija, confiendo en que nadie lo borre o modifique por no saber para que se usa (incluso nosotros mismos). Para evitar eso, podemos poner esta entrada en el texto mismo del script, y pedir al shell que alimente al comando con esta entrada. Para eso, decimos

```
comando <<algo
Aqui ponemos el texto
tantas lineas como queramos
algo
```

El texto comienza en la linea siguiente al comando, y está delimitado por una linea que contenga solamente la palabra que pusimos despues del `<<`. Al ejecutarse, el shell alimenta al comando con el texto, y luego se salta el texto y

continua la ejecución en la linea siguiente. Es posible alimentar a una tubería de este modo, por ejemplo, sabiendo que el programa `fmt` formatea su entrada a parrafos con lineas de una longitud dada, por defecto 75 caracteres, podemos enviar un correo sin preocuparnos de escribirlo en lineas cortas:

```
fmt <<EOF | mail gunnar -s ejemplo
Este es el texto
del correo, que puedo escribir sin preocuparme del formato
EOF
```

Es convencional usar el delimitador `EOF`, que significa End of file, pero no es de modo alguno requerido, y en caso de tener más de uno de estos documentos en un script, es recomandable dar a cada uno su delimitador, para beneficio de lectores posteriores.

5.2. Comillas inversas y xargs

Otra caracteristica complicada del shell es la sustitución de comillas inversas. Cuando escribimos una cadena dentro de estas, el shell la interpreta como un comando. Este comando es ejecutado, se captura su salida estandar, se separa en palabras, y estas son sustituidas en vez de la cadena completa. Es posible tener comillas dobles o sencillas anidadas dentro de estas, y a la vez es posible anidar estas dentro de comillas dobles. En este caso, la salida no es separada en palabras, sino que se deja como una sola. Esta caracteristica permite capturar la salida de un comando para usarla como argumentos de otro, o para iterar sobre la misma usando `for`.

```
for file in `find . -name '*.c'`; do

    cp $file $file.orig;
    sed -e 's/call_me/call_him/' $file >$file.new
    mv $file.new $file

done
```

Si bien esta caracteristica es poderosa, puede llevar facilmente a una sintaxis enmarañada, con multiples niveles de anidamiento de comillas que es necesario proteger de la expansión. Podemos evitar esto en la mayoría de los casos, usando el comando `xargs`. Este comando toma su entrada estandar, la separa en palabras, y ejecuta al comando dado como argumento con estas palabras como argumentos extra. Si la linea de argumentos resultaría demasiado larga, `xargs` la parte en varias, y ejecuta al comando varias veces. Por ejemplo, para averiguar en que archivo .h se define a cierta estructura, podemos decir:

```
find . -name '*.h' | xargs grep 'struct *cierta_estructura'
```

5.2.1. Redirecciones complejas

El shell nos permite efectuar algunas redirecciones más complejas que las mostradas al inicio. Por ejemplo, es posible redirigir descriptores de archivo especificos. Si decimos `n>archivo`, el archivo será abierto para escritura, en el descriptor de archivo n. Analogamente, es posible hacer esto para lectura, o para escribir al final. Por supuesto, cualquier descriptor fuera de 0, 1 y 2 es no estandar, de modo que el programa que ejecutemos debe estar esperando que dichos descriptores esten abiertos y usarlos, o no pasará nada.

Pero de cualquier forma es útil esta caracteristica. Supongamos que queremos desacernos de la salida estandar, y guardar los mensajes de error en un archivo. Entonces podemos decir `comando >/dev/null 2>/tmp/log`, lo cual logra nuestro proposito. Comunmente, queremos procesar la salida de error, junto con la normal. Para esto, podemos 'duplicar' un descriptor de archivo, diciendo, por ejemplo `comando 2>&1`, que manda la salida de error a donde va la estandar. Si queremos procesar la salida de error en una tubería, y deshacernos de la salida normal, primero duplicamos la normal sobre la de error, y luego redirigimos la normal a `/dev/null`: `comando 2>&1 >/dev/null`. Notese que esto funciona por que la primera redirección copia el descriptor uno sobre el dos, no lo liga. Además, el orden importa. Si decimos `comando >/dev/null 2>&1` primero se redirige la salida a `/dev/null`, y luego esa salida redirigida se copia sobre la salida de error, de modo que ambas se van al mismo lugar. Esta última operación es tan comun, que hay una sintaxis especial para ella: `comando &>archivo` redirige tanto la salida estandar como la salida de error al archivo nombrado.

5.3. Agrupación de procesos

En ocasiones queremos ejecutar un proceso complejo o tardado, con un `for` o un `while`, por ejemplo, y queremos dejarlo trabajando en el fondo como una unidad. Para hacer esto, podemos encerrar el comando completo entre parentesis. Entonces, el shell tratará al comando como una unidad, suspendiendolo o reanudandolo junto. Esto resulta también util cuando el proceso a ejecutarse cambia el entorno del shell. Como el proceso es ejecutado por un subshell, cualquier modificación al entorno solo afecta al del subshell, y no al original.

5.4. Substitución de procesos

La sustitución de procesos es una caracteristica que no está disponible en todos los sistemas. Es necesario que el sistema soporte los 'named pipes' o el sistema de archivos `/dev/fd`. Si existe el soporte, bash puede efectuar una sustitución del estilo `<(lista)` o `>(lista)`. La lista de procesos se efectua con su salida o su entrada, respectivamente, conectada a un 'named pipe', o a un archivo en `/dev/fd`. El nombre de este archivo se sustituye en vez de la lista, como un argumento al comando. El comando puede abrir el archivo y usarlo para entrada o salida, segun el caso.

Si bien no es fácil ver el uso de esta caracteristica, puede resultar muy poderosa en algunos casos. Por ejemplo, el programa `diff` compara el contenido de dos archivos. Usando este metodo, podemos usarlo para comparar la salida de dos comandos distintos, sin necesidad de grabar la salida de estos a archivos. Uniendo esto con un programa como `lynx`, podemos por ejemplo comparar dos versiones de una página web, guardadas en distintos servidores, sin tener que guardarlas en ningun lado:

```
diff -u <(lynx -source url1) <(lynx -source url2)
```

Referencias

[1] Página de manual de bash(1)