Programación del Shell

El intérprete de comandos o "shell" de UNIX es también un lenguage de programación completo. La programación de shell se usa mucho para realizar tareas repetidas con frecuencia. Los diseñadores de sistemas suelen escribir aplicaciones en el lenguaje de base del sistema operativo, C en el caso de UNIX, por razones de rapidez y eficiencia. Sin embargo, el shell de UNIX tiene un excelente rendimiento en la ejecución de "scripts" (guiones); ésta es la denominación aplicada a los programas escritos en el lenguaje del shell. Se han creado aplicaciones completas usando solamente scripts.

Comandos multilínea.

Una línea de comando termina con un caracter nuevalínea. El caracter nuevalínea se ingresa digitando la tecla ENTER. Varios comandos pueden escribirse en una misma línea usando el separador ";"

```
echo $LOGNAME; pwd; date
```

Si un comando no cabe en una línea, la mayoría de los intérpretes continúan la digitación en la línea siguiente. Para establecer específicamente que un comando continúa en la línea siguiente, hay dos formas, mutuamente excluyentes (se usa una u otra, pero no ambas al mismo tiempo):
- terminar la primera línea con \ :

```
$ echo $LOGNAME \
> $HOME
```

que muestra algo como

```
jperez /home/jperez
```

- dejar una comilla sin cerrar:

```
$ echo "$LOGNAME
> $HOME"
```

que produce el mismo resultado.

En los ejemplos anteriores hemos escrito los indicadores de comando. Al continuar el comando en la segunda línea, el indicador de comandos cambia de $ a >, es decir, del indicador de comando de primer nivel PS1 al indicador de comando de segundo nivel PS2. Si no se quiere terminar el comando multilínea, puede interrumpirse el ingreso con Ctrl-C, volviendo el indicador de comando a PS1 inmediatamente.

El archivo de comandos (script).

Es cómodo poder retener una lista larga de comandos en un archivo, y ejecutarlos todos de una sola vez sólo invocando el nombre del archivo. Crear el archivo `misdatos.sh` con las siguientes líneas:

```
# misdatos.sh
# muestra datos relativos al usuario que lo invoca
#
echo "MIS DATOS."
echo " Nombre: "$LOGNAME
echo "Directorio: "$HOME
echo -n "Fecha: "
date
echo
# fin misdatos.sh
```

El símbolo # indica comentario. Para poder ejecutar los comandos contenidos en este archivo, es preciso dar al mismo permisos de ejecución:

```
chmod ug+x misdatos.sh
```

La invocación (ejecución) del archivo puede realizarse dando el nombre de archivo como argumento a `bash`

```
bash misdatos.sh
```

o invocándolo directamente como un comando

```
misdatos.sh
```

Puede requerirse indicar una vía absoluta o relativa, o referirse al directorio actual,

```
./misdatos.sh
```

si el directorio actual no está contenido en la variable PATH.

Comentarios en los scripts.

En un script todo lo que venga después del símbolo # y hasta el próximo caracter nuevalínea se toma como comentario y no se ejecuta.

```
echo Hola todos     # comentario hasta fin de línea
```

sólo imprime "Hola todos".

```
# cat /etc/passwd
```

no ejecuta nada, pues el símbolo # convierte toda la línea en comentario.

Los scripts suelen encabezarse con comentarios que indican el nombre de archivo y lo que hace el script. Se colocan comentarios de documentación en diferentes partes del script para mejorar la comprensión y facilitar el mantenimiento. Un caso especial es el uso de # en la primera línea para indicar el intérprete con que se ejecutará el script. El script anterior con comentarios quedaría así:

```
#!/bin/bash
# misdatos.sh
```

```
#
# muestra datos propios del usuario que lo invoca
#
echo "MIS DATOS."
echo " Nombre: "$LOGNAME
echo "Directorio: "$HOME
echo -n "Fecha: "
date    # muestra fecha y hora
echo    # línea en blanco para presentación
# fin misdatos.sh
```

La primera línea indica que el script será ejecutado con el intérprete de comandos `bash`. Esta indicación debe ser siempre la primera línea del script y no puede tener blancos.

Estructuras básicas de programación.

Las estructuras básicas de programación son sólo dos: la estructura repetitiva y la estructura alternativa. Cada forma tiene sus variaciones, y la combinación de todas ellas generan múltples posibilidades, pero detrás de cualquiera de ellas, por compleja que parezca, se encuentran siempre repeticiones o alternativas.

Estructura repetitiva: se realiza una acción un cierto número de veces, o mientras dure una condición.

```
mientras haya manzanas,
  pelarlas;

desde i = 1 hasta i = 7
  escribir dia_semana(i);
```

Esta escritura informal se denomina "pseudocódigo", por contraposición al término "código", que sería la escritura formal en un lenguaje de programación. En el segundo ejemplo, `dia_semana(i)` sería una función que devuelve el nombre del día de la semana cuando se le da su número ordinal.

Estructura alternativa: en base a la comprobación de una condición, se decide una acción diferente para cada caso.

```
si manzana está pelada,
  comerla,
en otro caso,
  pelarla;

# oráculo
caso $estado en
soltero)
  escribir "El casamiento será su felicidad";
casado)
  escribir "El divorcio le devolverá la felicidad";
divorciado)
  escribir "Sólo será feliz si se vuelve a casar";
fin caso
```

Funciones: una tarea que se realiza repetidamente dentro del mismo programa puede escribirse

aparte e invocarse como una "función". Para definir una función es preciso elegir un nombre y escribir un trozo de código asociado a ese nombre. La función suele recibir algún valor como "parámetro" en base al cual realiza su tarea. Definida así la función, para usarla basta escribir su nombre y colocar el valor del parámetro entre paréntesis.

```
función area_cuadrado (lado)
devolver lado * lado;

función dia_semana(día_hoy)
  caso $dia_hoy en
  1)
    devolver "Lunes";;
  2)
    devolver "Martes";;
  3)
    devolver "Miércoles";:
  4)
    devolver "Jueves";;
  5)
    devolver "Viernes;;
  6)
    devolver "Sábado";;
  7)
    devolver "Domingo";;
  fin caso;
```

Comandos de programación.

En esta sección veremos los comandos típicos de programación del shell. Obsérvese que el shell toma la convención inversa de C para cierto y falso: cierto es 0, y falso es distinto de 0. El shell adopta esta convención porque los comandos retornan 0 cuando no hubo error. Veremos dos comandos, `true` y `false`, que retornan siempre estos valores; se usan en algunas situaciones de programación para fijar una condición.

true

Este comando no hace nada, sólo devuelve siempre 0, el valor verdadero. La ejecución correcta de un comando cualquiera devuelve 0.

```
  true
  echo $?
```

muestra el valor 0; la variable $? retiene el valor de retorno del último comando ejecutado.

false

Este comando tampoco hace nada sólo devuelve siempre 1; cualquier valor diferente de 0 se toma como falso. Las diversas condiciones de error de ejecución de los comandos devuelven valores diferentes de 0; su significado es propio de cada comando.

```
  false
  echo $?
```

muestra el valor 1.

if

El comando `if` implementa una estructura alternativa. Su sintaxis es

```
  if expresión ; then comandos1 ; [else comandos2 ;] fi
```

o también

```
  if expresión
  then
    comandos1
  [else
    comandos2]
 fi
```

Si se usa la forma multilínea cuando se trabaja en la línea de comandos, el indicador cambia a > hasta que termina el comando.

La expresión puede ser cualquier expresión lógica o comando que retorne un valor; si el valor retornado es 0 (cierto) los `comandos1` se ejecutan; si el valor retornado es distinto de 0 (falso) los `comandos1` no se ejecutan. Si se usó la forma opcional con `else` se ejecutan los `comandos2`.

```
  if true; then echo Cierto ; else echo Falso ; fi
```

siempre imprime Cierto; no entra nunca en `else`.

```
  if false; then echo Cierto ; else echo Falso ; fi
```

siempre imprime Falso, no entra nunca en `then`.

Construcciones más complejas pueden hacerse usando `elif` para anidar alternativas. Escribir en un archivo las líneas que siguen

```
  # ciertofalso.sh: escribe cierto o falso según parámetro
numérico
  #
  if [ $1 = "0" ]
  then
    echo "Cierto: el parámetro es 0."
  else
    echo "Falso: el parámetro no es 0."
  fi
```

Convertir el script en ejecutable. Invocar este script con

```
  ./ciertofalso.sh N
```

donde N es un número entero 0, 1, 2, etc. La variable $1 hace referencia a este parámetro de invocación. Verificar el resultado.

Crear y ejecutar el siguiente script:

```
  # trabajo.sh: dice si se trabaja según el día
  #    invocar con parámetros:
  #      domingo, feriado, u otro nombre cualquiera
  #
  if [ $1 = "domingo" ]
  then
    echo "no se trabaja"
  elif [ $1 = "feriado" ]
  then
```

```
    echo "en algunos se trabaja"
  else
    echo "se trabaja"
  fi
```

for

Este comando implementa una estructura repetitiva, en la cual una secuencia de comandos se ejecuta una y otra vez. Su sintaxis es

```
  for variable in lista ; do comandos ; done
```

Se puede probar en la línea de comandos:

```
  for NUMERO in 1 2 3 4 ; do echo $NUMERO ; done
  for NOMBRE in alfa beta gamma ; do echo $NOMBRE ; done
  for ARCH in * ; do echo Nombre archivo $ARCH ; done
```

El caracter * es expandido por el shell colocando en su lugar todos los nombres de archivo del directorio actual.

Crear y probar el siguiente script.

```
  # listapal.sh: lista de palabras
  #   muestra palabras de una lista interna
  #
  LISTA="silla mesa banco cuadro armario"
  for I in $LISTA
  do
    echo $I
  done
  # fin listapal.sh
```

En el siguiente script, el comando `expr` calcula expresiones aritméticas; notar su sintaxis.

```
  # contarch.sh
  #   muestra nombres y cuenta archivos en el directorio actual
  #
  CUENTA=0
  for ARCH in *
  do
    echo $ARCH
    CUENTA=`expr $CUENTA + 1`      # agrega 1 a CUENTA
  done
  echo Hay $CUENTA archivos en el directorio `pwd`
  # fin contarch.sh
```

case

Este comando implementa alternativas o "casos"; elige entre múltiples secuencias de comandos la secuencia a ejecutar. La elección se realiza encontrando el primer patrón con el que aparea una cadena de caracteres. Su sintaxis es

```
  case $CADENA in
  patrón1)
```

```
  comandos1;;
patrón2)
  comandos2;;
...
*)
  comandosN;;
esac
```

El patrón * se coloca al final; aparea cualquier cadena, y permite ejecutar *comandosN* cuando ninguna de las opciones anteriores fue satisfecha. Crear el siguiente script:

```
# diasemana.sh: nombres de los días de la semana
#   invocar con número del 0 al 6; 0 es domingo
case $1 in
0)   echo Domingo;;
1)   echo Lunes;;
2)   echo Martes;;
3)   echo Miércoles;;
4)   echo Jueves;;
5)   echo Viernes;;
6)   echo Sábado;;
*)   echo Debe indicar un número de 0 a 6;;
esac
```

Invocarlo como

```
diasemana.sh N
```

donde *N* es cualquier número de 0 a 6, otro número, o una cadena cualquiera. Verificar el comportamiento.

Crear el archivo `estacion.sh` con estas líneas:

```
# estacion.sh
#   indica la estación del año aproximada según el mes
#
case $1 in
diciembre|enero|febrero)
   echo Verano;;
marzo|abril|mayo)
   echo Otoño;;
junio|julio|agosto)
   echo Invierno;;
setiembre|octubre |noviembre)
   echo Primavera;;
*)
   echo estacion.sh: debe invocarse como
   echo estacion.sh mes
   echo con el nombre del mes en minúscula;;
esac
# fin estacion.sh
```

El valor $1 es el parámetro recibido en la línea de comando. La opción *) aparea con cualquier cadena, por lo que actúa como "en otro caso"; es útil para dar instrucciones sobre el uso del

comando. En las opciones, | actúa como OR; pueden usarse también comodines * y ?. Invocar el script:

```
bash estacion.sh octubre
bash estacion.sh
```

¿Cómo podría modificarse el script anterior para que aceptara el mes en cualquier combinación de mayúsculas y minúsculas?

while

Este comando implementa una estructura repetitiva en la cual el conjunto de comandos se ejecuta *mientras* se mantenga válida una condición (while = mientras). La condición se examina al principio y luego cada vez que se completa la secuencia de comandos. Si la condición es falsa desde la primera vez, los comandos no se ejecutan nunca. Su sintaxis es

```
while condición ; do comandos ; done
```

En el guión que sigue la expresión entre paréntesis rectos es una forma de invocar el comando `test`, que realiza una prueba devolviendo cierto o falso. El operador `-lt`, "lower than", significa "menor que". Observar su sintaxis, sobre todo la posición de los espacios en blanco, obligatorios.

```
# crear1.sh
# crea archivos arch1....arch9, los lista y luego borra
VAL=1
while [ $VAL -lt 10 ]         # mientras $VAL < 10
do
  echo creando archivo arch$VAL
  touch arch$VAL
  VAL=`expr $VAL + 1`
done
ls -l arch[0-9]
rm arch[0-9]
# fin crear1.sh
```

until

Este comando implementa una estructura repetitiva en la cual el conjunto de comando se ejecuta hasta que se cumpla una condición. En cuanto la condición se cumple, dejan de ejecutarse los comandos. La condición se examina al principio; si es verdadera, los comandos no se ejecutan. Notar la diferencia con `while`. Su sintaxis es

```
until condición ; do comandos ; done
```

Usando `until`, el script anterior se escribiría

```
# crear2.sh
# crea archivos arch1....arch9, los lista y luego borra
VAL=1
until [ $VAL -eq 10 ]       # hasta que $VAL = 10
do
  echo creando archivo arch$VAL
  touch arch$VAL
  VAL=`expr $VAL + 1`
done
```

```
  ls -l arch[0-9]
  rm arch[0-9]
  # fin crear2.sh
```

exit

Este comando se utiliza en programación de shell para terminar inmediatamente un script y volver al shell original.

```
  exit
```

en un script, termina inmediatamente el script; en la línea de comando, termina la ejecución del shell actual, y por lo tanto la sesión de UNIX.

```
  exit 6
```

termina el script devolviendo el número indicado, lo que puede usarse para determinar condiciones de error.

```
  exit 0
```

termina el script devolviendo 0, para indicar la finalización exitosa de tareas. Escribir sólo `exit` también devuelve código de error 0.
El siguiente script ofrece un ejemplo de uso.

```
#!/bin/bash
# exitar.sh: prueba valores de retorno de exit
#
clear
echo "Prueba de valores de retorno"
echo "  Invocar con parámetros "
echo "       bien, error1, error2, cualquier cosa o nada"
echo "  Verificar valor de retorno con"
echo '       echo $?'
echo
VALOR=$1
case $VALOR in
bien)
  echo "    -> Terminación sin error."
  exit 0;;
error1)
  echo "    -> Terminación con error 1." ; exit 1;;
error2)
  echo "    -> Terminación con error 2." ; exit 2;;
*)
  echo "    -> Terminación con error 3."
  echo "       (invocado con parámetro no previsto o sin
parámetro."
  exit 3;;
esac
```

expr

Este comando recibe números y operadores aritméticos como argumentos, efectúa los cálculos indicados y devuelve el resultado. Cada argumento debe estar separado por blancos. Opera sólo con números enteros y realiza las operaciones suma (+), resta (-), multiplicación (*), división entera (/), resto de división entera (%). Los símbolos * y / deben ser escapados escribiendo * y \/, al igual que

los paréntesis, que deben escribirse \(y \).

El comando `expr` usa la convención de C para cierto y falso: 0 es falso, y distinto de 0 es cierto. No confundir con la convención que toma el shell en sus valores true y false, que es la contraria.

```
expr 4 + 5
```

devuelve 9 (4 + 5 = 9).

```
expr 3 \* 4 + 6 \/2
```

devuelve 15 (3 * 4 + 6 /2 = 15).

```
expr 3 \* \( 4 + 3 \) \/2
```

devuelve 10 (3 * (4 + 3) / 2 = 10).

`expr` también realiza operaciones lógicas de comparación, aceptando los operadores `=, !=, >, <, >= y <=`. El operador `!=` es "no igual"; el `!` se usa para negar. Estos caracteres también requieren ser escapados.

```
echo `expr 6 \< 10`
```

devuelve 1, cierto para `expr`.

```
echo `expr 6 \> 10`
```

devuelve 0, falso para `expr`.

```
echo `expr abc \< abd`
```

devuelve 1, cierto para `expr`.

test

Este comando prueba condiciones y devuelve valor cierto (0) o falso (distinto de 0) según el criterio de cierto y falso del shell; esto lo hace apto para usar en la condición de `if`. Tiene dos formas equivalentes

```
test expresión
[ expresión ]
```

Los blancos entre la expresión y los paréntesis rectos son necesarios.

`test` devuelve cierto ante una cadena no vacía, y falso ante una cadena vacía:

```
if test "cadena" ; then echo Cierto ; else echo Falso; fi
if test "" ; then echo Cierto ; else echo Falso ; fi
if [ cadena ] ; then echo Cierto ; else echo Falso; fi
if [ ] ; then echo Cierto ; else echo Falso ; fi
```

`test` prueba una cantidad de condiciones y situaciones:

```
if [ -f archivo ]; then echo "Existe archivo"; \
  else echo "No existe archivo"; fi
```

La condición `[ -f archivo ]` es cierta si archivo existe y es un archivo normal; análogamente, -r comprueba si es legible, -w si puede escribirse, -x si es ejecutable, -d si es un directorio, -s si tiene tamaño mayor que 0.

Las condiciones

```
[ $DIR = $HOME ]
[ $LOGNAME = "usuario1" ]
[ $RESULTADO != "error" ]
```

comparan cadenas de caracteres; = para igualdad y != para desigualdad.
La condición

```
[ "$VAR1" ]
```

devuelve falso si la variable no está definida. Las comillas dan la cadena nula cuando `VAR1` no está

definida; sin comillas no habría cadena y daría error de sintaxis.
La condición

```
[ expnum1 -eq expnum2 ]
```

compara igualdad de expresiones que resultan en un número. Pueden ser expresiones numéricas o lógicas, ya que éstas también resultan en números. Los operadores numéricos derivan sus letras del inglés, y son -eq (igualdad), -neq (no igual, desigualdad), -lt (menor), -gt (mayor), -le (menor o igual), -ge (mayor o igual).

El comando `test` se usa mucho para determinar si un comando se completó con éxito, en cuyo caso el valor de retorno es 0. El siguiente script crea un archivo si no existe.

```
# nvoarch.sh
# recibe un nombre y crea un archivo de ese nombre;
# si ya existe emite un mensaje
if [ -f $1 ]
then
  echo El archivo $1 ya existe
else
  touch $1
  echo Fue creado el archivo $1
fi
echo
# fin nvoarch.sh
```

Para comprobar su acción,

```
bash nvoarch.sh nuevo1
ls -l nuevo1
```

crea el archivo; `ls` comprueba que existe;

```
bash nvoarch.sh nuevo1
```

da mensaje indicando que el archivo ya existe.

Otros operadores aceptados por test son `-a` (AND) y `-o` (OR).

```
# rwsi.sh
# indica si un archivo tiene permiso de lectura y escritura
ARCH=$1
if [ -r $ARCH -a -w $ARCH ]
then
  echo El archivo $ARCH se puede leer y escribir
else
  echo Al archivo $ARCH le falta algún permiso
fi
ls -l $ARCH
# fin rwsi.sh
```

read

Este comando tiene como propósito solicitar información al usuario. Su ejecución captura las digitaciones del usuario, hasta obtener un caracter nueva línea (techa Enter). El ejemplo siguiente obtiene datos del usuario, los repite en pantalla, solicita confirmación y emite un mensaje en consecuencia.

```
# yo.sh: captura datos del usuario
```

```
#
clear
echo "Datos del usuario."
echo -n "Nombre y apellido: "; read NOMBRE
echo -n "Cédula de identidad: "; read CEDULA
echo
echo "Ha ingresado los siguientes datos:"
echo "   Nombre y apellido: $NOMBRE"
echo "   Cédula de Identidad: $CEDULA"
echo -n "¿Es correcto?(sN):"; read RESP
if [ "$RESP" = "s" -o $RESP = "S" ]
then
  echo "Fin de ingreso."
else
  echo "Debe ingresar sus datos nuevamente."
fi
```

Parámetros.

El siguiente programa muestra los parámetros que recibe al ser invocado:

```
# ecopars.sh
# muestra los parámetros recibidos
echo Cantidad de parámetros: $#
for VAR in $*
do
  echo $VAR
done
# fin ecopars.sh

ecopars.sh Enero Febrero Marzo
```

muestra la cantidad y los parámetros recibidos. La variable $* contiene la lista de parámetros, y $# la cantidad.

Dentro del script, los parámetros recibidos pueden referenciarse con $1, $2, $3, ..., $9, siendo $0 el nombre del propio programa. Debido a que se los reconoce por su ubicación, se llaman parámetros posicionales. El siguiente programa se invoca con tres parámetros y muestra sus valores:

```
# mostrar3.sh
# se invoca con 3 parámetros y los muestra
echo nombre del programa: $0
echo parámetros recibidos:
echo $1; echo $2; echo $3
echo
# fin mostrar3.sh
```

¿Cómo se podría verificar la invocación con 3 parámetros, y emitir un mensaje de error en caso contrario (cuando el usuario ingresa menos de 3 parámetros)?

Depuración.

Se llama depuración ("debug") de un programa al proceso de verificar su funcionamiento en todos

los casos posibles y corregir sus errores ("bugs", "pulgas"; del inglés, literalmente, "chinche"; por extensión, insecto pequeño).

Cuando se está escribiendo un script, puede convenir invocarlo de forma especial para generar información de comandos ejecutados y errores, para ayudar en la depuración. Las salidas se imprimen en el error estándar, por lo que pueden direccionarse a un archivo sin mezclarse con la salida del comando.

```
bash -x ecopars.sh
```

imprime cada comando en la salida;

```
bash -v ecopars.sh
```

invoca el script obteniendo una salida verbosa con información sobre cada comando ejecutado.

```
bash -xv ecopars.sh 2>ecopars.err
```

reúne las dos fuentes de información y direcciona el error estándar a un archivo.

Bibliografía y Referencias.

Comandos: `true false if for case while until exit expr test`
Referencias: Coffin[1989], Kernighan-Pike[1984]; páginas man de Linux "bash" ("Bourne-Again shell").

Víctor A. González Barbone *vagonbar en fing edu uy*
Instituto de Ingeniería Eléctrica - *Facultad de Ingeniería* - *Montevideo, Uruguay.*