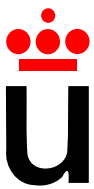
Tema 7: Shell Scripting

Enrique Soriano

Laboratorio de Sistemas,
Grupo de Sistemas y Comunicaciones,
URJC

22 de abril de 2010



Universidad
Rey Juan Carlos

¿Cuándo hago un script de Shell?

- Pasos para realizar una tarea:
  1. Mirar si hay alguna herramienta que haga lo que queremos → buscar en el manual.
  2. Si no encontramos, intentar combinar distintas herramientas → **programar un script de Shell**.
     IDEA: combinar herramientas que hacen bien una única tarea para llevar a cabo tareas más complejas.
  3. Si no podemos, hacer una herramienta → programar en C (u otro lenguaje).

Un script:

- Tiene que acabar en una nueva de línea.
- Tiene que tener permisos de ejecución.
- El comando *built-in* `exit` sale del script con el status indicado en su argumento. Si no se le especifica, sale con el valor de `$status`.

```
term% cat holamundo
#!/bin/rc
echo Hola mundo
exit ''

term%
```

Listas

- Cada elemento es una string (acabada en un caracter nulo).
- La llamada `getenv()` reemplaza los nulos por un espacio (menos el último).

Listas

```
term% x=(hola que tal)
term% x=(hola (que tal))
term% x=((hola que) (tal)) # es lo mismo!
term% echo $x(2) # el índice empieza en 1!
term% echo $#x # nº de elementos
term% cat /env/x
term% xd -c /env/x
```

Argumentos de un script

- $*
  Lista con todos los argumentos.
- $#*
  Número de argumentos (elementos de $*).
- $1,$2,..
  Primer argumento, segundo...
- $0
  Nombre del programa

Variables especiales

- `$home`
- `$pid`
- `$apid`
- `$prompt`
- `$path`
- `$status`
- `$ifs`

Operador de concatenación

```
term% z=$a^$b
```

- Si $#a es igual que $#b, se concatena el enésimo elemento de $a con el enésimo de $b.
- Si es una cadena o una lista de un único elemento, concatena ese elemento con todas (distribución).
- Si el número de elementos es distinto y mayor que uno, o alguna lista está vacía: error.

```
term% chmod +x '/un/path/largo/'^(file1 file2)
```

De lista a cadena

- `$"var`
  Se sustituye por una única cadena con todos los elementos de `$var` concatenados. Por tanto, el número de elementos es 1.

```
term% var=(uno dos tres)
term% b=$"var
term% echo $var $b
term% echo $#var $#b
term% cat /env/var /env/b
term% xd -c /env/var /env/b
```

Variables vacías

- lista vacía $\neq$ variable con una string vacía

```
term% var1=()
term% var2=''
term% echo $#var1 $#var2
term% echo $var1 $var2
```

Más vale prevenir...

- rfork(1) es un *built-in* que permite configurar el proceso en el que ejecuta el script.
- Padre e hijo comparten cosas por defecto: espacio de nombres y entorno, entre otras.
  - `rfork n` → nuevo espacio de nombres, copiado del padre.
  - `rfork N` → nuevo espacio de nombres, limpio.
  - `rfork e` → nuevo entorno, copiado del padre.
  - `rfork E` → nuevo entorno, limpio.

```
#!/bin/rc
rfork e # casi siempre queremos esto
...
```

Operaciones aritméticas

- La Shell no sabe nada sobre números.
- hoc(1) es potente para cálculos con coma flotante, tiene un lenguaje parecido a C, constantes definidas (PI, PHI, E)...
- bc(1) es otra calculadora, más cómoda para cambiar de bases (ibase y obase).

```
term% echo '7 + 4' | hoc
term% echo '7 + 4' | bc
term% result=`{echo '7 + 4' | bc}
term% echo 'obase=16 ; 77 ' | bc
```

Test

También nos permite

- Comparaciones
- Comprobar si existen ficheros y directorios
- Comprobar permiso para leer, escribir y ejecutar

Constructor de agrupaciones

- $\{\ldots\}$
  Los comandos se ejecutan como si fuera un único comando.

```
term% { echo hola ; echo adios } > /tmp/afile
term% { echo hola ; echo adios } | tr o 0
term% { sleep 100 ; echo tiempo } &
```

Subshell

- `@{...}`
  Los comandos se ejecutan en un nuevo shell.

```
term% @{ cd /sys/src/cmd ; 8c -FVw ls.c}
```

For

- `for ( var in lista ) {...}`
  Se itera por cada elemento en la lista. En n-ésima iteración, `$var` contendrá el n-ésimo elemento de la lista.

```
term% for (i in a b c ) { echo elemento: $i }
term% for (i in `{ ls } ) { echo entrada: $i }
term% for (i in `{ seq 1 10 } ) { echo contador: $i }
```

If

- `if ( comando ) {...}`
  `if not {...}`
  Se ejecutan los comandos si el comando acaba con un estatus correcto. En otro caso, se ejecutan los comandos del `if not`.
  No puede haber un `if not` sin `if`.

```
term% if ( test -f fich.txt ) { echo si existe } ; if not { echo no existe }
```

Comparación de cadenas

- `~ cadena patrón ...`
- Compara la cadena con cada patrón. Si algún patrón coincide con la cadena, el status será correcto. En otro caso, el status será erróneo.
- No confundir con **globbing**: no se intenta encajar el patrón con los nombres de los ficheros.
- Para el patrón: * ? []
- Si cadena es una lista, se pasa a una string.

```
term% if ( ~ $var fich1.txt ){echo var es fich1.txt}
term% if ( ~ $var *.txt *.pdf){echo acaba en .txt o en .pdf}
```

Negación

- `! comando`
  Si el status fue correcto, se cambia a erróneo, y viceversa.

```
term% if (! ~  $var *.txt)
    {
        echo var no acaba en .txt
    }
```

Switch

- `switch( cadena ){`
  `case patron1; ...;`
  `case patron2 ; ....`
  `}`
  Se ejecutan todos los comandos que correspondan al primer patrón con el que encaje la cadena.

```
term% switch( $var ){
case *.txt
     echo acaba en .txt
case *.pdf
     echo acaba en .pdf
case *
     echo acaba en otra cosa
}
```

And y Or

- `comando1 && comando2 && ...`
  Se ejecuta el siguiente comando si el actual acaba con un status correcto. Al final el status es un AND de todos.
- `comando1 || comando2 || ...`
  Se ejecuta el siguiente comando si el actual acaba con status incorrecto. Al final, el status es un OR de todos.

```
term% if ( test -f file1 && test -d dir1 )
      { echo existen }
term% test -f file1 && test -d dir1 && echo existen
```

While

- `while ( comando ) { ... }`
  Se itera mientras que el comando salga con status correcto.

```
term% while ( test -f /tmp/file ){
     echo /tmp/file sigue existiendo
     sleep 10
}
term% while ( sleep 1 ){
     # sleep siempre sale bien!!!
     cat /tmp/file
}
```

Filtros útiles

- `sort`
  ordena las líneas de varias formas.
- `uniq`
  elimina líneas contiguas repetidas.
- `tail`
  muestra las últimas líneas.

P. ej: `term% ps | tail +3 # a partir de la 3ª`

```
term% ps | tail -3 # las 3 ultimas
term% seq 1 1000 | sort
term% seq 1 1000 | sort -n
```

Tr

- Traduce caracteres. El primer argumento es el conjunto de caracteres a traducir. El segundo es el conjunto al que se traducen. El enésimo carácter del primer conjunto se traduce por el enésimo carácter del segundo.
- `-d`
  Borra los caracteres del único conjunto que se le pasa como argumento.
- Se le pueden dar rangos, p. ej.
  `term% cat fichero | tr a-z A-Z`

Expresiones regulares (*regexp*)

- Es un lenguaje formal para describir/buscar cadenas de caracteres.
- Parecidas a los patrones de la Shell o de globbing, pero más potentes.
- Una string encaja con sí misma, por ejemplo `'a'` con `'a'`.

Expresiones regulares (*regexp*)

- `.`
  encaja con cualquier carácter, por ejemplo 'a'.
- `[conjunto]`
  encaja con cualquier carácter en el conjunto, por ejemplo `[abc]` encaja con `'a'`. Se pueden especificar rangos, p. ej. `[a-zA-Z]`.
- `[^conjunto]`
  encaja con cualquier carácter que **no esté** en el conjunto, por ejemplo `[^abc]` NO encaja con `'a'`, sin embargo sí encaja con `'z'`.

Expresiones regulares (*regexp*)

- ^
  encaja con *principio de línea*.
- $
  encaja con *final de línea*.
- Una regexp e_1 concatenada a otra regexp e_2, $e_1 e_2$, encaja con una string si una parte p_1 de la string encaja con e_1 y otra parte contigua, p_2, encaja con e_2.

P. ej:
`'az'` encaja con la regexp `[a-d]z`

Expresiones regulares (*regexp*)

- `exp*`
  encaja si aparece **cero o más veces** la regexp que lo precede.
- `exp+`
  encaja si aparece **una o más veces** la regexp que lo precede.

P. ej:
'aaa' encaja con la regexp a*
'baaa' encaja con la regexp ba+
'bb' encaja con la regexp ba*
'bb' no encaja con la regexp ba+

Expresiones regulares (*regexp*)

- `exp?`
  encaja si aparece **cero o una vez** la regexp que lo precede. Se utiliza para partes opcionales.
- `(exp)`
  agrupa expresiones regulares.

P. ej:
`'az', 'av', 'a'` encajan con la regexp az?
`'abab'` encaja con la regexp `(ab)+`
`'abab', 'ababab', 'abababababa'` encajan con la regexp `(ab)+`

Expresiones regulares (*regexp*)

- `exp | exp`
  si encaja con alguna de las regexp que están separadas por la barras
- `\`
  carácter de escape: hace que el símbolo pierda su significado especial.

P. ej:

`'aass'` encaja con la regexp `(aass|booo)`

`'hola*'` encaja con la regexp `a\*`

Grep

- Filtra líneas usando expresiones regulares.
- `-v`
  Realiza lo inverso: imprime las líneas que no encajan.
- `-n`
  Indica el número de línea.
- `-e`
  indica que el siguiente argumento es una expresión.

Sed

- Es un editor: aplica el comando de sed a cada línea que lee y escribe el resultado por su salida. Sin el modificador `-n`, escribe todas las líneas después de procesarlas.
- Comandos:
  `q` → Sale del programa.
  `d` → Borra la línea.
  `p` → Imprime la línea.
  `r` → Lee e inserta un fichero.
  `s` → Sustituye. ← la que más se usa!!!

Sed

- Direcciones:
  `número` → actúa sobre esa línea.
  `/regexp/` → líneas que encajan con la regexp.
  `$` → la última línea.
- Se pueden usar intervalos:
  `número,número` → actúa en ese intervalo.
  `número,$` → desde la línea *número* hasta la última.
  `número,/regexp/` → desde la línea *número* hasta la primera que encaje con regexp.

Sed

Ejemplos:
`sed '3,6d'` → borra las líneas de la 3 a la 6
`sed -n '/BEGIN|begin/,/END|end/p'` → imprime las líneas entre esas regexp
`sed '3q'` → imprime las 3 primeras líneas.
`sed -n '13,$p'` → imprime desde la línea 13 hasta la última.
`sed '/[Hh]ola/d'` → borra las líneas que contienen 'Hola' u 'hola'.

Sed

Sustitución

- `sed 's/regexp/sustitución/'` → sustituye la primero subcadena que encaja con la exp. por la cadena *sustitución*.
- `sed 's/regexp/sustitución/g'` → sustituye todas las subcadenas de la línea que encajan con la exp. por la cadena *sustitución*.
- `sed 's/(regexp)regexp.../ \1 sustitución/g'` → usa las subcadenas que encajaron con las agrupaciones en la cadena de sustitución.

Sed

Ejemplos

`sed 's/[0-9]/X/'` → el primer número de la línea se sustituye por una X.

`sed 's/[0-9]/X/g'` → todos los números de la línea se sustituyen por una X.

```
sed 's/^([A-Za-z]+)[ ]+([A-Z]+)/NOMBRE:\1 NOTA:\2/g'
```

hacer mykill.rc

Read

Lee una línea de su entrada estándar y lo escribe en su salida estándar. Muy útil si no queremos partir por la salida de un comando con los caracteres de `$ifs`.
Ejemplos:

```
term% echo nombre; n=‘{read} ; echo tu nombre es $n

term% ps | while(l=‘{read}) { echo linea:$"l }
```

NO ES LO MISMO QUE:

```
term% for(i in ‘{ps}) { echo item:$i }
```

AWK

- AWK es un lenguaje de programación completo pensado para procesar líneas separadas en campos.
- Se le puede pasar el programa como parámetro.
- Lo puede leer de un fichero si usamos el modificador -f

ver total.awk

AWK

- El modificador -F indica el separador de campos, en forma de regexp. Por omisión, se usa [ ]+

```
term% ls -q | awk -F '\)' '{ print $2 }'
```

AWK

Imprimir:

- print
  Sentencia que imprime los operandos. Si se separan con comas, inserta un espacio. Al final imprime un salto de línea.
- printf()
  Función que imprime, ofrece control sobre el formato de forma similar a la función de libc para C:

```
term% ls -l | awk '{ printf("Size: %08d KBytes\n", $6) }'
```

AWK

Variables:

- $0
  La línea que está procesando.
- $1, $2 ...
  El primer, segundo... campo de la línea.
- NR
  Número del registro (línea) que se está procesando.

AWK

Variables:

- NF
  Número del campos del registro que se está procesando.
- var=contenido
  Se pueden declarar variables dentro del programa. Con el modificador -v se pueden pasar variables al programa.

```
term% ls -l | awk '
{
size=$6 ; printf("Size: %08d KBytes\n", size)
}'
```

AWK

`patrón { programa }`

Actuando sólo en unas líneas, que se ajustan a un patrón, que puede ser:

- Expresión regular
  Se procesan las líneas que encajen con la regexp.

```
term% ls -l | awk '/[Dd]esktop/{ print $1 }'

term% ls -l | awk '$1 ~  /[Dd]esktop/ { print $1 }'
```

AWK

- Expresión de relación
  Se comparan valores y se evalúa la expresión.
  `term% ls -l | awk ' NR >= 5 && NR <= 10 { print $1 }'`

AWK

Inicialización y finalización:

```
BEGIN{
...
}

patrón{
...
}

END{
...
}
```