



Recursosinformáticos

# Programación **shell** en **Unix/Linux**

## **sh, ksh, bash**

(con ejercicios corregidos)

3ª edición

Christine DEFFAIX RÉMY



# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

## Introducción

<b>1. Definición del shell</b> .....	<b>19</b>
<b>2. Características de un intérprete de comandos</b> .....	<b>19</b>
<b>3. Intérpretes de comando (shells)</b> .....	<b>20</b>
3.1 Historia .....	20
3.2 ¿ Con qué shell hay que programar ? .....	21
3.2.1 Scripts de inicio .....	21
3.2.2 Otros scripts .....	21
3.3 Nombre de los ejecutables .....	21
<b>4. Shells tratados en este libro</b> .....	<b>22</b>

## Mecanismos esenciales del shell

<b>1. Comandos internos y externos</b> .....	<b>23</b>
1.1 Comandos externos .....	23
1.2 Comandos internos .....	25
1.3 Implementación interna e implementación externa .....	26
<b>2. Impresión por pantalla</b> .....	<b>27</b>
2.1 El comando echo .....	27
2.1.1 El carácter "\n" .....	27
2.1.2 El carácter "\c" .....	28
2.1.3 El carácter "\t" .....	29
2.1.4 Listado de caracteres de escape .....	29
2.2 Los comandos print y printf .....	29
<b>3. El carácter ~ (tilde)</b> .....	<b>30</b>

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

<b>4. El comando interno cd</b> .....	<b>31</b>
<b>5. Sustitución de nombres de archivos</b> .....	<b>31</b>
5.1 Expresiones básicas .....	32
5.1.1 El carácter *	32
5.1.2 El carácter ?	32
5.1.3 Los caracteres [ ]	32
5.2 Expresiones complejas .....	34
5.2.1 ? (expresión)	34
5.2.2 *(expresión)	34
5.2.3 +(expresión)	35
5.2.4 @(expresión)	35
5.2.5 !(expresión)	35
5.2.6 Alternativas	36
5.3 Interpretación del shell	36
<b>6. Separador de comandos</b> .....	<b>37</b>
<b>7. Redirecciones</b> .....	<b>38</b>
7.1 Entrada y salidas estándar de los procesos .....	38
7.1.1 Entrada estándar	38
7.1.2 Salida estándar	38
7.1.3 Salida de error estándar	39
7.2 Herencia .....	39
7.3 Redirección de las salidas en escritura .....	40
7.3.1 Salida estándar	40
7.3.2 Salida de error estándar	41
7.3.3 Salida estándar y salida de error estándar	42
7.3.4 Protección ante borrado involuntario de un archivo	43
7.3.5 Eliminar las impresiones por pantalla	43
7.3.6 Mecanismo interno	44
7.4 Redirección de la entrada estándar	45
7.5 Redirecciones avanzadas .....	47
7.5.1 Redirigir los descriptores 1 y 2 hacia el mismo archivo	47

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

7.5.2 La redirección doble en lectura .....	53
7.5.3 Cierre de un descriptor .....	54
<b>8. Tuberías de comunicación .....</b>	<b>54</b>
8.1 Comandos que no leen su entrada estándar .....	56
8.2 Comandos que leen su entrada estándar .....	57
8.2.1 Ejemplos triviales .....	57
8.2.2 Caso de los filtros .....	57
8.3 Complementos .....	61
8.3.1 Encadenar tuberías .....	61
8.3.2 Duplicar las salidas .....	61
8.3.3 Enviar la salida estándar y la salida de error estándar por la tubería .....	62
<b>9. Agrupación de comandos .....</b>	<b>63</b>
9.1 Paréntesis .....	64
9.2 Las llaves .....	69
9.3 Conclusión .....	73
<b>10. Procesos en segundo plano .....</b>	<b>74</b>
<b>11. Ejercicios .....</b>	<b>74</b>
11.1 Funcionalidades varias .....	74
11.1.1 Ejercicio 1: comandos internos y externos .....	74
11.1.2 Ejercicio 2: generación de nombres de archivo .....	74
11.1.3 Ejercicio 3: separador de comandos .....	75
11.2 Redirecciones .....	75
11.2.1 Ejercicio 1 .....	75
11.2.2 Ejercicio 2 .....	75
11.2.3 Ejercicio 3 .....	76
11.2.4 Ejercicio 4 .....	76
11.2.5 Ejercicio 5 .....	76
11.2.6 Ejercicio 6 .....	76
11.3 Tuberías de comunicación .....	77
11.3.1 Ejercicio 1 .....	77
11.3.2 Ejercicio 2 .....	77

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

.11.3.3 Ejercicio 3.....	77
.11.3.4 Ejercicio 4.....	77

## Configuración del entorno de trabajo

<b>1. Variables de entorno.....</b>	<b>79</b>
1.1 Listado de variables.....	79
1.2 Mostrar el valor de una variable.....	80
1.3 Modificación del valor de una variable.....	80
1.4 Variables principales.....	81
1.4.1 HOME.....	81
1.4.2 PATH.....	81
1.4.3 PWD.....	83
1.4.4 PS1.....	83
1.4.5 PS2.....	87
1.4.6 TMOUT.....	87
1.4.7 TERM.....	88
1.4.8 LOGNAME.....	88
1.4.9 Procesos y variables de entorno.....	88
1.5 Exportación de variables.....	89
1.5.1 Listado de variables exportadas.....	89
1.5.2 Variables que deben exportarse.....	90
1.5.3 Exportar una variable.....	90
<b>2. Las opciones del shell.....</b>	<b>94</b>
2.1 Activar y desactivar una opción del shell.....	94
2.2 Visualizar la lista de opciones.....	94
2.3 Opciones principales.....	95
2.3.1 ignoreeof.....	95
2.3.2 noclobber.....	95
2.3.3 emacs y vi.....	96
2.3.4 xtrace.....	97
<b>3. Los alias.....</b>	<b>97</b>

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

3.1 Definir un alias .....	97
3.2 Visualizar la lista de alias .....	98
3.2.1 Visualizar todos los alias .....	98
3.2.2 Visualizar un alias en particular .....	98
3.3 Eliminar un alias .....	98
<b>4. Histórico de comandos .....</b>	<b>98</b>
4.1 Configurar la recuperación de comandos en ksh .....	100
4.1.1 Opción vi .....	100
4.1.2 Opción emacs .....	101
4.2 Configurar la recuperación de comandos en bash .....	105
4.3 Completar nombres de archivo .....	105
4.3.1 Completar en bash .....	105
4.3.2 Completar en ksh .....	106
4.3.3 Tabla resumen .....	108
<b>5. Los archivos de entorno .....</b>	<b>108</b>
5.1 Características de los archivos de entorno .....	108
5.1.1 Shell de conexión .....	108
5.1.2 Archivos de entorno leídos por el shell de conexión .....	109
5.2 Sesión utilizando un Bourne Shell .....	112
5.3 Sesión utilizando un Korn Shell .....	113
5.4 Sesión utilizando un Bourne Again Shell .....	116
<b>6. Ejercicios .....</b>	<b>118</b>
6.1 Variables de entorno .....	118
6.1.1 Ejercicio 1 .....	118
6.1.2 Ejercicio 2 .....	118
6.2 Alias de comando .....	119
6.2.1 Ejercicio 1 .....	119
6.2.2 Ejercicio 2 .....	119

Las bases de la programación shell

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

<b>1. Las variables de usuario</b> .....	<b>121</b>
1.1 Poner nombre a una variable .....	121
1.2 Definir una variable .....	121
1.2.1 Asignar un valor a una variable .....	122
1.2.2 Asignar un valor con al menos un espacio .....	122
1.2.3 Variable indefinida .....	122
1.2.4 Borrar la definición de una variable .....	123
1.2.5 Aislar el nombre de una variable .....	123
1.2.6 Variables numéricas .....	124
1.2.7 Variables complejas .....	125
1.3 Sustitución de variables .....	126
<b>2. Sustitución de comandos</b> .....	<b>128</b>
<b>3. Caracteres de protección</b> .....	<b>129</b>
3.1 Las comillas simples .....	129
3.2 El carácter \ .....	131
3.3 Las comillas dobles .....	132
<b>4. Recapitulación</b> .....	<b>132</b>
<b>5. Interpretación de una línea de comandos</b> .....	<b>133</b>
<b>6. Escritura y ejecución de un script en shell</b> .....	<b>134</b>
6.1 Definición .....	134
6.2 Ejecución de un script por un shell hijo .....	135
6.3 Ejecución de un script por el shell actual .....	141
6.4 Comentarios .....	143
<b>7. Variables reservadas del shell</b> .....	<b>144</b>
7.1 Los parámetros posicionales .....	144
7.2 El comando shift .....	146
7.2.1 Sintaxis .....	146

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

7.2.2 Principio.....	146
7.3 Código de retorno de un comando.....	148
7.3.1 La variable \$ ?.....	148
7.3.2 El comando exit.....	149
7.4 Otras variables especiales.....	150
7.4.1 PID del shell intérprete.....	150
7.4.2 PID del último proceso ejecutado en segundo plano.....	151
<b>8. El comando read.....</b>	<b>153</b>
8.1 Sintaxis.....	153
8.2 Lecturas del teclado.....	153
8.3 Código de retorno.....	155
8.4 La variable IFS.....	156
<b>9. Ejecución de verificaciones.....</b>	<b>157</b>
9.1 Introducción.....	157
9.2 El comando test.....	157
9.2.1 Sintaxis.....	158
9.2.2 Verificaciones de archivos.....	158
9.2.3 Verificaciones de cadenas de caracteres.....	161
9.2.4 Verificaciones de números.....	163
9.2.5 Los operadores.....	164
9.2.6 Ejemplo concreto de uso.....	165
9.3 El comando [[ ]].	166
<b>10. Los operadores del shell.....</b>	<b>170</b>
10.1 Evaluación del operador &&.....	171
10.2 Evaluación del operador   .....	172
<b>11. Aritmética.....</b>	<b>173</b>
11.1 El comando expr.....	173
11.1.1 Sintaxis.....	173
11.1.2 Operadores.....	173
11.2 El comando (( )). .....	177
11.2.1 Sintaxis.....	177



# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

.11.2.2 Uso .....	177
11.3 El comando let .....	180
11.4 Aritmética de punto flotante .....	180
11.4.1 ksh93 .....	180
.11.4.2 Otros shells .....	181
<b>12. Sustitución de expresiones aritméticas .....</b>	<b>182</b>
<b>13. Corrección de un script .....</b>	<b>183</b>
13.1 Opción -x .....	183
13.2 Otras opciones .....	186
<b>14. Las estructuras de control .....</b>	<b>187</b>
14.1 if .....	187
14.2 case .....	191
.14.2.1 Sintaxis .....	191
.14.2.2 Principio .....	191
.14.2.3 Uso .....	193
14.3 Bucle for .....	196
14.4 Bucle while .....	200
.14.4.1 Sintaxis .....	200
.14.4.2 Uso .....	200
.14.4.3 Bucle infinito .....	201
14.5 until .....	204
.14.5.1 Sintaxis .....	204
.14.5.2 Uso .....	204
14.6 break y continue .....	208
<b>15. Ejercicios .....</b>	<b>210</b>
15.1 Variables, caracteres especiales .....	210
.15.1.1 Ejercicio 1: variables .....	210
.15.1.2 Ejercicio 2: variables .....	211
.15.1.3 Ejercicio 3: sustitución de comando .....	211
.15.1.4 Ejercicio 4: caracteres de protección .....	211
15.2 Variables, visualización y lectura del teclado .....	212

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

.15.2.1 Ejercicio 1: variables	212
.15.2.2 Ejercicio 2: parámetros posicionales	212
.15.2.3 Ejercicio 3: lectura de teclado	212
15.3 Tests y aritmética	213
.15.3.1 Ejercicio 1: tests a los archivos	213
.15.3.2 Ejercicio 2: tests de cadenas de caracteres	213
.15.3.3 Ejercicio 3: tests numéricos	214
.15.3.4 Ejercicio 4: aritmética	214
.15.3.5 Ejercicio 5: operadores lógicos de los comandos [ ], [[ ]] y operadores lógicos del shell	214
15.4 Estructuras de control if, case, bucle for	215
.15.4.1 Ejercicio 1: los comandos [ ] y [[ ]], la estructura de control if	215
.15.4.2 Ejercicio 2: estructura de control case, bucle for	215
15.5 Bucles	216
.15.5.1 Ejercicio 1: bucle for, comando tr	216
.15.5.2 Ejercicio 2: bucle for, aritmética	216
.15.5.3 Ejercicio 3: bucles for, while	217

## Aspectos avanzados de la programación shell

<b>1. Comparación de las variables \$* y \$@</b>	<b>219</b>
1.1 Uso de \$* y de \$@	219
1.2 Uso de "\$*"	221
1.3 Uso de "\$@"	222
<b>2. Sustitución de variables</b>	<b>223</b>
2.1 Longitud del valor contenido en una variable	223
2.2 Manipulación de cadenas de caracteres	223
2.2.1 Eliminar el fragmento más pequeño de la izquierda	224
2.2.2 Eliminar el fragmento más grande de la izquierda	224
2.2.3 Eliminar el fragmento más pequeño de la derecha	225
2.2.4 Eliminar el fragmento más grande de la derecha	225

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

<b>3. Tablas</b>	<b>226</b>
3.1 Asignar un elemento	226
3.2 Referenciar un elemento	226
3.3 Asignación global de una tabla	227
3.4 Referenciar todos los elementos de una tabla	228
3.5 Obtener el número de elementos de una tabla	228
3.6 Obtener la longitud de un elemento de una tabla	228
3.7 Tablas asociativas	229
<b>4. Inicialización de parámetros posicionales con set</b>	<b>229</b>
<b>5. Funciones</b>	<b>230</b>
5.1 Definición de una función	230
5.2 Código de retorno de una función	232
5.3 Ámbito de las variables	234
5.4 Definición de variables locales	235
5.5 Paso de parámetros	237
5.6 Utilizar la salida de una función	239
5.7 Programa completo del ejemplo	240
<b>6. Comandos de salida</b>	<b>242</b>
6.1 El comando print	242
6.1.1 Uso simple	242
6.1.2 Supresión del salto de línea natural de print	242
6.1.3 Mostrar argumentos que comienzan por el carácter "-"	243
6.1.4 Escritura hacia un descriptor determinado	243
6.2 El comando printf	244
<b>7. Gestión de entradas/salidas de un script</b>	<b>245</b>
7.1 Redirección de entradas/salidas estándar	245
7.2 Gestión de archivos	249
7.2.1 Apertura de archivo	249
7.2.2 Lectura a partir de un archivo	249

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

7.2.3 Escritura en un archivo .....	249
7.2.4 Cierre de un archivo .....	250
7.3 Tratamiento de un archivo .....	251
7.3.1 Información previa .....	251
7.3.2 Las diferentes formas de explotar un archivo .....	252
7.3.3 Repartir una línea en campos .....	258
7.3.4 Modificar el separador de línea .....	259
<b>8. El comando eval .....</b>	<b>261</b>
<b>9. Gestión de señales .....</b>	<b>263</b>
9.1 Señales principales .....	263
9.2 Ignorar una señal .....	264
9.3 Modificar el comportamiento asociado a una señal .....	265
9.4 Restablecer el comportamiento por defecto del shell respecto a una señal .....	266
9.5 Usar trap desde un script de shell .....	267
<b>10. Gestión de menús con select .....</b>	<b>268</b>
<b>11. Análisis de las opciones de un script con getopt .....</b>	<b>270</b>
<b>12. Gestión de un proceso en segundo plano .....</b>	<b>276</b>
<b>13. Script de archivado incremental y transferencia sftp automática .....</b>	<b>278</b>
13.1 Objetivo .....	278
13.2 El archivo uploadBackup.sh .....	281
13.3 El archivo funciones.inc.sh .....	284
<b>14. Ejercicios .....</b>	<b>287</b>
14.1 Funciones .....	287
14.1.1 Ejercicio 1: funciones simples .....	287
14.1.2 Ejercicio 2: funciones simples, valor de retorno .....	288
14.1.3 Ejercicio 3: paso de parámetros, retorno de valor .....	289
14.1.4 Ejercicio 4: archivos .....	290

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

.14.1.5 Ejercicio 5: archivos, funciones, menú select .....	291
.14.1.6 Ejercicio 6: archivos, tablas asociativas (bash 4, ksh93).....	292

## Expresiones regulares

<b>1. Introducción</b> .....	<b>293</b>
<b>2. Caracteres comunes en ERb y ERe</b> .....	<b>294</b>
<b>3. Caracteres específicos de ERb</b> .....	<b>296</b>
<b>4. Caracteres específicos de ERe</b> .....	<b>297</b>
<b>5. Uso de expresiones regulares por comandos</b> .....	<b>299</b>
5.1 El comando vi .....	299
5.2 El comando grep .....	299
5.3 El comando expr .....	302
5.4 sed y awk .....	305
<b>6. Ejercicios</b> .....	<b>306</b>
6.1 Expresiones regulares .....	306
6.1.1 Ejercicio 1: expresiones regulares con vi .....	306
6.1.2 Ejercicio 2: grep .....	307

## El comando sed

<b>1. Uso del comando sed</b> .....	<b>309</b>
<b>2. Ejemplos</b> .....	<b>312</b>
2.1 Uso de sed en línea de comandos .....	312
2.1.1 El comando d (delete) .....	312

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

2.1.2 El comando p (print).....	313
2.1.3 El comando w (write).....	314
2.1.4 Negación de un comando (!).....	314
2.1.5 El comando s (sustitución).....	315
2.2 Script sed.....	317
<b>3. Ejercicios.....</b>	<b>319</b>
3.1 Expresiones regulares.....	319
3.1.1 Ejercicio 1: inserción de marcadores en un archivo.....	319
3.1.2 Ejercicio 2: formato de archivos.....	320

## El lenguaje de programación awk

<b>1. Principio.....</b>	<b>321</b>
1.1 Sintaxis.....	321
1.2 Variables especiales.....	322
1.2.1 Variables predefinidas a partir de la ejecución de awk.....	322
1.2.2 Variables inicializadas en el momento del tratamiento de una línea.....	323
1.2.3 Ejemplos simples.....	323
1.3 Criterios de selección.....	325
1.3.1 Expresiones regulares.....	326
1.3.2 Verificaciones lógicas.....	327
1.3.3 Intervalos de líneas.....	328
1.4 Estructura de un script awk.....	328
1.4.1 BEGIN.....	328
1.4.2 Secciones intermedias.....	328
1.4.3 END.....	328
1.4.4 Comentarios.....	328
1.4.5 Variables.....	328
1.4.6 Ejemplo.....	329
<b>2. Operadores.....</b>	<b>330</b>
<b>3. La función printf.....</b>	<b>332</b>

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

<b>4. Redirecciones</b> .....	<b>333</b>
<b>5. Lectura de la línea siguiente: next</b> .....	<b>335</b>
<b>6. Estructuras de control</b> .....	<b>336</b>
6.1 if .....	336
6.2 for .....	337
6.3 While .....	338
6.4 do-while .....	338
6.5 break .....	338
6.6 continue .....	338
<b>7. Finalizar un script</b> .....	<b>339</b>
<b>8. Tablas</b> .....	<b>339</b>
8.1 Tablas indexadas con un entero .....	339
8.2 Tablas asociativas .....	340
8.2.1 Definición .....	340
8.2.2 Verificar la existencia de un elemento .....	342
8.2.3 Eliminar un elemento .....	342
<b>9. Los argumentos de la línea de comandos</b> .....	<b>343</b>
<b>10. Funciones integradas</b> .....	<b>345</b>
10.1 Funciones que trabajan con cadenas .....	345
10.2 Funciones matemáticas .....	346
10.3 Otras funciones .....	346
10.3.1 La función getline .....	346
10.3.2 La función close .....	350
10.3.3 La función system .....	351
<b>11. Funciones de usuario</b> .....	<b>352</b>

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

<b>12. Ejercicios</b> .....	<b>354</b>
12.1 awk en línea de comandos .....	354
.12.1.1 Ejercicio 1: awk y otros filtros .....	354
12.1.2 Ejercicio 2: criterios de selección .....	354
12.1.3 Ejercicio 3: criterios de selección, visualización de campos, secciones BEGIN y END .....	355
12.2 Scripts awk .....	356
.12.2.1 Ejercicio 4: funciones .....	356
.12.2.2 Ejercicio 5: análisis de un archivo de log .....	358
.12.2.3 Ejercicio 6: generación de un archivo de etiquetas .....	359

## Los comandos filtro

<b>1. Introducción</b> .....	<b>361</b>
<b>2. Sintaxis de llamada a comandos filtro</b> .....	<b>361</b>
<b>3. Visualización de datos</b> .....	<b>362</b>
3.1 Consulta de datos, creación de archivos: cat .....	362
3.2 Valor de los bytes de un flujo de datos: od .....	363
3.3 Filtrado de líneas: grep .....	364
3.4 Últimas líneas de un flujo de datos: tail .....	368
3.5 Primeras líneas de un flujo de datos: head .....	370
3.6 Duplicación de la salida estándar: tee .....	370
3.7 Numeración de líneas: nl .....	371
3.8 Presentación de un flujo de datos: pr .....	372
<b>4. Tratamiento de datos</b> .....	<b>374</b>
4.1 Recuento de líneas, de palabras y caracteres: wc .....	374
4.2 Extracción de caracteres: cut .....	376
4.3 Ordenación de datos: sort .....	377



# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

4.4 paste .....	380
4.5 split .....	382
4.6 Transformación de caracteres: tr .....	383
4.7 Eliminación de líneas repetidas: uniq .....	384
<b>5. Compresión, archivado y conversión .....</b>	<b>387</b>
5.1 Compresión: gzip, bzip2 .....	387
5.2 Archivos tar .....	389
5.3 Archivos cpio .....	390
5.4 Copia física, transformaciones: dd .....	393
5.5 Cambio de codificación: iconv .....	394
<b>6. Comandos de red seguros .....</b>	<b>395</b>
6.1 Conexión remota: ssh .....	395
6.2 Transferencia de archivos: sftp .....	396
6.2.1 Comandos de sftp que se ejecutan en la máquina local .....	398
6.2.2 Comandos que se ejecutan en la máquina remota .....	399
6.2.3 Comandos de transferencia .....	399
6.2.4 Conexión automática sin contraseña .....	400
<b>7. Otros comandos .....</b>	<b>402</b>
7.1 El comando xargs .....	402
7.2 Comparar dos archivos: cmp .....	404
7.3 Líneas comunes entre dos archivos: comm .....	405

## Soluciones a los ejercicios

<b>1. Soluciones del capítulo Mecanismos esenciales del shell .....</b>	<b>409</b>
1.1 Funcionalidades varias .....	409
1.1.1 Ejercicio 1: comandos internos y externos .....	409
1.1.2 Ejercicio 2: generación de nombres de archivo .....	409
1.1.3 Ejercicio 3: separador de comandos .....	411
1.2 Redirecciones .....	411
1.2.1 Ejercicio 1 .....	411

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

1.2.2 Ejercicio 2 .....	411
1.2.3 Ejercicio 3 .....	412
1.2.4 Ejercicio 4 .....	412
1.2.5 Ejercicio 5 .....	412
1.2.6 Ejercicio 6 .....	412
1.3 Tuberías de comunicación .....	413
1.3.1 Ejercicio 1 .....	413
1.3.2 Ejercicio 2 .....	413
1.3.3 Ejercicio 3 .....	413
1.3.4 Ejercicio 4 .....	413
<b>2. Soluciones del capítulo Configuración del entorno de trabajo .....</b>	<b>414</b>
2.1 Variables de entorno .....	414
2.1.1 Ejercicio 1 .....	414
2.1.2 Ejercicio 2 .....	414
2.2 Alias de comando .....	415
2.2.1 Ejercicio 1 .....	415
2.2.2 Ejercicio 2 .....	416
<b>3. Soluciones del capítulo Las bases de la programación shell .....</b>	<b>416</b>
3.1 Variables, caracteres especiales .....	416
3.1.1 Ejercicio 1: variables .....	416
3.1.2 Ejercicio 2: variables .....	417
3.1.3 Ejercicio 3: sustitución de comando .....	417
3.1.4 Ejercicio 4: caracteres de protección .....	418
3.2 Variables, visualización y lectura del teclado .....	419
3.2.1 Ejercicio 1: variables .....	419
3.2.2 Ejercicio 2: parámetros posicionales .....	420
3.2.3 Ejercicio 3: lectura de teclado .....	420
3.3 Tests y aritmética .....	421
3.3.1 Ejercicio 1: tests a los archivos .....	421
3.3.2 Ejercicio 2: tests de cadenas de caracteres .....	422
3.3.3 Ejercicio 3: tests numéricos .....	424
3.3.4 Ejercicio 4: aritmética .....	424

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

3.3.5 Ejercicio 5: operadores lógicos de los comandos [ ], [[ ]] y operadores lógicos del shell .....	425
3.4 Estructuras de control if, case, bucle for .....	426
3.4.1 Ejercicio 1: los comandos [ ] y [[ ]], la estructura de control if .....	426
3.4.2 Ejercicio 2: estructuras de control case, bucle for .....	427
3.5 Bucles .....	428
3.5.1 Ejercicio 1: bucle for, comando tr .....	428
3.5.2 Ejercicio 2: bucle for, aritmética .....	429
3.5.3 Ejercicio 3: bucles for, while .....	430
<b>4. Soluciones del capítulo Aspectos avanzados de la programación shell .....</b>	<b>432</b>
4.1 Funciones .....	432
4.1.1 Ejercicio 1: funciones simples .....	432
4.1.2 Ejercicio 2: funciones simples, valor de retorno .....	433
4.1.3 Ejercicio 3: paso de parámetros, retorno de valor .....	435
4.1.4 Ejercicio 4: archivos .....	437
4.1.5 Ejercicio 5: archivos, funciones, menú select .....	438
4.1.6 Ejercicio 6: archivos, tablas asociativas (bash 4, ksh93) .....	440
<b>5. Soluciones del capítulo Las expresiones regulares .....</b>	<b>441</b>
5.1 Expresiones regulares .....	441
5.1.1 Ejercicio 1: expresiones regulares con vi .....	441
5.1.2 Ejercicio 2: grep .....	442
<b>6. Soluciones del capítulo El comando sed .....</b>	<b>443</b>
6.1 Expresiones regulares .....	443
6.1.1 Ejercicio 1: inserción de marcadores en un archivo .....	443
6.1.2 Ejercicio 2: formato de archivos .....	444
<b>7. Soluciones del capítulo El lenguaje de programación awk .....</b>	<b>445</b>
7.1 awk en línea de comandos .....	445
7.1.1 Ejercicio 1: awk y otros filtros .....	445
7.1.2 Ejercicio 2: criterios de selección .....	446

# Programación shell en Unix/Linux

sh, ksh, bash (con ejercicios corregidos) (3ª edición)

7.1.3 Ejercicio 3: criterios de selección, visualización de campos, secciones BEGIN y END .....	446
7.2 Scripts awk .....	447
7.2.1 Ejercicio 4: funciones .....	447
7.2.2 Ejercicio 5: análisis de un archivo de log .....	449
7.2.3 Ejercicio 6: generación de un archivo de etiquetas .....	451

## Anexos

<b>1. Caracteres especiales de shell .....</b>	<b>453</b>
<b>2. Comandos internos de shell .....</b>	<b>454</b>
<b>3. Orden de interpretación de un comando .....</b>	<b>457</b>
<b>índice .....</b>	<b>459</b>

# Programación shell en Unix/Linux

sh, ksh, bash (3ª edición)

Este libro de **programación shell** va dirigido a usuarios y administradores de sistemas Unix/Linux que desean aprender a programar **scripts shell**. Se detallan las funcionalidades de tres shells usados habitualmente (**Bourne Shell, ksh 88 y 93, bash**) y sus diferencias. Los conceptos se presentan de manera progresiva y pedagógica, convirtiendo este libro en un soporte ideal destinado tanto a la **formación profesional como a la autoformación**.

Los primeros capítulos se destinan al **funcionamiento del shell**: ejecución de un comando, caracteres especiales del shell usados habitualmente (**redirecciones, tuberías,...**), configuración del entorno de trabajo del usuario (variables y archivos de entorno,...). Los **mecanismos internos** se explican detalladamente y se ilustran con múltiples esquemas.

A continuación, el libro se centra en la programación propiamente dicha. Las **bases de la programación (variables, estructuras de control, comandos de verificación y cálculo,...**) se presentan e ilustran mediante una gran cantidad de ejemplos y, más adelante, se detallan los aspectos avanzados de la programación shell (**gestión de archivos, funciones,...**).

La última parte trata sobre las **utilidades anexas** indispensables para el tratamiento de cadenas de caracteres y de archivos de texto: **las expresiones regulares básicas y extendidas, el editor no interactivo sed**, una visión extendida **del lenguaje awk** y los principales comandos filtro de los sistemas unix.

Los **ejercicios** permitirán al lector practicar la escritura de scripts Shell a lo largo de todo el libro.

**Los ejemplos de scripts shells** incluidos en el libro se pueden descargar en esta página.

## Los capítulos del libro:

Prólogo – Introducción – Mecanismos esenciales del shell – Configuración del entorno de trabajo – Las bases de la programación shell – Aspectos avanzados de la programación shell – Expresiones regulares – El comando sed – El lenguaje de programación awk – Los comandos filtro – Soluciones a los ejercicios – Anexos

---

## Christine DEFFAIX RÉMY

Ingeniera informática en la empresa Ociensa Technologies, especialista en la administración y desarrollo en Unix y Linux, **Christine Deffaix Rémy** interviene en tareas de desarrollo y formación en grandes cuentas. Su sólida experiencia junto a sus cualidades pedagógicas proporcionan una obra realmente eficaz para el aprendizaje en la creación de scripts shell.

# Prológo

Esta obra está dirigida a usuarios y administradores de sistemas Unix y Linux que desean automatizar tareas y que para ello necesitan formarse en la programación de scripts de shell. Para la lectura de este libro, se debe conocer previamente los principios básicos de estos sistemas operativos y hay que saber gestionar archivos y procesos con facilidad. También es deseable el conocimiento de conceptos básicos de programación (noción de variable, de estructuras de control y de algoritmo).

Se explican con detalle las funcionalidades de tres shell actualmente en uso (Bourne Shell, Korn Shell y Bourne Again Shell) y se muestran sus diferencias. Se presentan los temas de forma progresiva y pedagógica, lo que convierte este libro en una herramienta ideal destinada tanto a la formación de profesionales como a la autoformación.

Los tres primeros capítulos de este libro se vuelcan en el funcionamiento del shell:

- El capítulo Introducción presenta los intérpretes de comandos disponibles en Unix/Linux.
- El capítulo Mecanismos esenciales del shell presenta los caracteres especiales del shell que se usan en línea de comandos o en scripts. Es importante estar familiarizado con los conceptos expuestos en este capítulo para aprender fácilmente los capítulos dedicados a la programación.
- El capítulo Configuración del entorno de trabajo permitirá al lector comprender cómo se forma el entorno de trabajo del shell y cómo modificar los parámetros de inicialización. Este capítulo puede estudiarse de forma independiente y no es necesario para tratar los siguientes capítulos.

Los capítulos Las bases de la programación shell y Aspectos avanzados de la programación shell están dedicados completamente a la programación. Al final del capítulo Las bases de la programación shell, el lector iniciado o medianamente experimentado habrá asimilado las principales funcionalidades del lenguaje que le permitirán escribir scripts: variables, condiciones, bucles, cálculos... El capítulo Aspectos avanzados de la programación shell presenta funcionalidades complementarias, entre las cuales: lectura/escritura de archivos de texto, gestión de argumentos de la línea de comandos, gestión de menús...

El capítulo Expresiones regulares presenta dos conjuntos de expresiones usadas en la búsqueda o en la sustitución de cadenas de caracteres por los comandos `expr`, `grep`, `vi`, `sed` y `awk`.

Los capítulos El comando `sed` y El lenguaje de programación `awk` presentan dos comandos Unix que permiten tratar cadenas de caracteres y archivos de texto.

El penúltimo capítulo, Los comandos filtro, presenta de forma detallada los principales comandos filtro de Unix que pueden usarse tanto en línea de comandos como en scripts de shell.

El último capítulo se dedica a la solución de los ejercicios.

## **Definición del shell**

El shell es un programa que tiene como función la de proporcionar la interfaz entre el usuario y el sistema Unix. Es un intérprete de comandos.

Hay varios shells disponibles para las plataformas Unix.

## Características de un intérprete de comandos

Los intérpretes de comandos disponibles en los entornos Unix tienen en común las siguientes funcionalidades:

- Proponen un juego de caracteres especiales que permiten desencadenar acciones concretas.
- Tienen comandos internos y palabras clave mediante algunos de los cuales se puede programar.
- Utilizan archivos de inicialización que permiten a un usuario parametrizar su entorno de trabajo.

Cada shell propone sus propios caracteres especiales, comandos internos, palabras clave y archivos de configuración. Afortunadamente, los intérpretes de comandos más utilizados en la actualidad derivan todos del shell Bourne y tienen, por consiguiente, un cierto número de funcionalidades en común.



# Intérpretes de comando (shells)

## 1. Historia

El shell considerado más antiguo es el Bourne Shell (**sh**). Fue escrito en los años 1970 por Steve Bourne en los laboratorios AT&T. Además de ejecutar comandos, dispone de funcionalidades de programación. El Bourne Shell es un Shell antiguo cada vez menos utilizado en las plataformas Unix.

Durante el mismo periodo, Bill Joy inventa el C-Shell (**csh**), incompatible con el Bourne, pero que ofrece funcionalidades suplementarias, como el histórico de comandos, el control de tareas, así como la posibilidad de crear alias de comandos. Estos tres aspectos se retomarán más tarde en el Korn Shell. El C-shell se usa poco en el mundo Unix.

En 1983, David Korn retoma el Bourne Shell y lo enriquece. Este nuevo intérprete tomará el nombre de Korn Shell (**ksh**). Este último se usará cada vez más, hasta el punto de convertirse en un estándar de hecho. El **ksh88** (versión de 1988) es, junto con el Bourne Again Shell (ver a continuación), el shell más utilizado actualmente. Ha servido como base para la estandarización del shell (IEEE POSIX 1003.2), representado por el **shell POSIX** (similar al ksh88).

En 1993, una nueva versión de Korn Shell ve la luz (**ksh93**). Presenta una retrocompatibilidad con ksh88, con algunas excepciones. ksh93 está disponible en algunas versiones Unix recientes: Solaris 11, AIX 6 y 7.

La Free Software Foundation propone el Bourne Again Shell (**bash**). Sigue la normativa **POSIX**, a la que ha añadido algunas extensiones. Este shell es el intérprete forjado como estándar en los sistemas Linux. Está a su vez disponible en estándar o en descarga para los sistemas Unix. En el momento de escribir este libro, la última versión de bash llevaba el número **4.3**.

## 2. ¿Con qué shell hay que programar?

### a. Scripts de inicio

En algunos sistemas, Bourne Shell interpreta los scripts de inicio. Si se desea modificar estos scripts o crear nuevos scripts de inicio, hay que restringirse a la sintaxis Bourne Shell. De forma más general, se debe emplear la sintaxis que corresponda al Shell que interpreta los scripts de inicio.

### b. Otros scripts

En el caso más común (scripts de tratamiento de ejecutables en modo de funcionamiento normal de Unix), el desarrollador elegirá o bien bash o bien ksh (88 o 93), según cuál sea el shell disponible en su sistema. Como ksh y bash tienen muchas características en común, se puede escribir un script compatible con ambos shell sin dificultad.

➤ Si se usan las características específicas de ksh93, el script no será compatible con sh, ksh (88) ni bash.

## 3. Nombre de los ejecutables

El nombre de los ejecutables difiere según el sistema Unix empleado y su versión. Por ejemplo, el Bourne Shell se encuentra en `/sbin/sh` en Solaris 10, `/usr/sunos/bin/sh` en Solaris 11 y `/usr/bin/bsh` en AIX 7.1. Este se ha eliminado en HP-UX a partir de la versión 11. El Korn Shell 88 se encuentra en `/usr/bin/ksh` en Solaris 10, AIX y HP-UX, pero está en `/usr/sunos/bin/ksh` en Solaris 11. El Korn Shell

93 está en `/usr/bin/ksh93` en AIX y en `/usr/bin/ksh` en Solaris 11. HP-UX pone a disposición el Shell POSIX bajo el nombre de `/usr/bin/sh`.

- Las páginas del manual `sh`, `ksh`, etc., informan acerca de la correspondencia entre los shells y los ejecutables.

## Shells tratados en este libro

Este libro detalla las funcionalidades de los shells Bourne, Korn y Bourne Again. Se tratan las extensiones principales de ksh93. Salvo que se especifique lo contrario, los temas abordados son compatibles con los tres shells. Cuando no sea así, la lista de shells que soportan la funcionalidad se indicará con una tabla como la siguiente:

ksh	bash
-----	------

**ksh** representa a ksh88 y a ksh93.

Los temas específicos del **Korn Shell 93** o del **bash versión 4** se destacan por las tablas siguientes:

ksh93
-------

bash4
-------

En este libro, el intérprete `/usr/bin/sh` hace referencia a **Bourne Shell**.

# Comandos internos y externos

Este capítulo presenta y explica de manera detallada las funcionalidades básicas del shell comúnmente utilizadas en los comandos Unix.

Un comando Unix pertenece a una de las dos siguientes categorías:

## 1. Comandos externos

Un comando externo es un archivo localizado en el árbol del sistema de archivos. Por ejemplo, cuando un usuario ejecuta el comando **ls**, el shell pide al núcleo de Unix cargar en memoria el archivo **/usr/bin/ls**.

Se consideran comandos externos los archivos que tengan uno de los formatos siguientes:

- Archivos con formato binario ejecutable.
- Archivos con formato de texto que representan un script de comandos (que puede estar escrito en Shell o en otro lenguaje, como por ejemplo Perl).

El comando **file** indica el tipo de datos contenidos en un archivo.

### Ejemplos

*El comando **ls** es un archivo con formato binario ejecutable. Resultado del comando **file**:*

```
$ file /usr/bin/ls
/usr/bin/ls: ELF 32-bit MSB executable SPARC Version 1,
dynamically linked, stripped
```

*El comando **miscript** es un script en shell. Resultado del comando **file**:*

```
$ file /home/cristina/miscript
miscript: ascii text
```

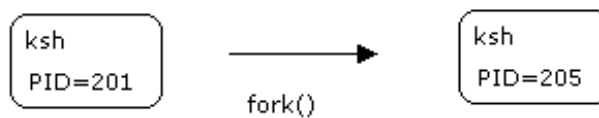
*El comando **miscript.pl** es un script en perl. Resultado del comando **file**:*

```
$ file /home/cristina/miscript.pl
miscript.pl: ascii text
```

- El argumento del comando **file** es un nombre de archivo expresado con ruta relativa o absoluta.

Los comandos externos se ejecutan por un shell hijo que actúa de intermediario (ver figura 1).

1ª etapa: Duplicación del shell



\$ ls

2ª etapa: El código del shell hijo se sustituye por el código del comando "ls"

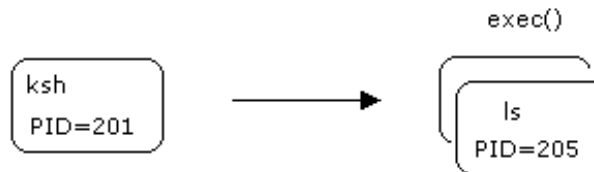


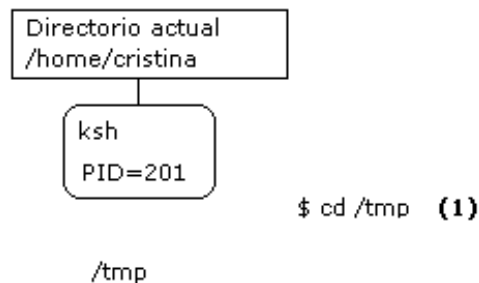
Figura 1: Ejecución de un comando externo

## 2. Comandos internos

Un comando interno se integra en los procesos shell (es el shell quien ejecuta la acción). Por lo tanto, no se corresponde en ningún caso a un archivo almacenado en disco.

El comando interno se ejecuta por el shell actual (ver figura 2).

1ª etapa: Ejecución del comando "cd". El shell reconoce uno de sus comandos internos



2ª etapa: El shell modifica el valor de su directorio actual

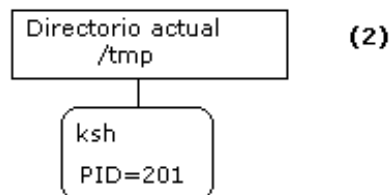


Figura 2: Ejecución de un comando interno

El comando **type** indica si un comando tiene una implementación interna.


### Ejemplos

El comando **cd** es un comando interno.

```
$ type cd
cd is a shell builtin
```

El comando **ls** es un comando externo.

```
$ type ls
ls is /usr/bin/ls
```

 **type** tiene como argumento el nombre de un comando. Si este último no es interno, se busca dentro de los directorios listados en PATH (ver el capítulo Configuración del entorno de trabajo - Variables de entorno).

### 3. Implementación interna e implementación externa

Ciertos comandos tienen una implementación interna y una implementación externa. En este caso:

- El comando interno se ejecuta con mayor prioridad.
- La ejecución de un comando interno es más rápida que la ejecución de un comando externo.
- El comando **type** indica que el comando es interno, pero no precisa que existe igualmente una implementación externa.

#### Ejemplo

El comando **pwd** es un comando interno al shell:

```
$ type pwd
pwd is a shell builtin
```

El comando **pwd** posee igualmente una implementación externa:

```
$ ls -l /usr/bin/pwd
-r-xr-xr-x 1 bin  bin   4616 Oct 6 1998 /usr/bin/pwd
```

Es el comando interno el que se ejecuta con mayor prioridad:

```
$ pwd
/home/cristina
```

Para forzar la ejecución del comando externo, es necesario indicar explícitamente la ruta del comando (ruta absoluta o relativa):

```
$ /usr/bin/pwd
/home/cristina
$ cd /usr/bin
$ ./pwd
/usr/bin
$
```

# Impresión por pantalla

## 1. El comando echo

El comando interno **echo** permite realizar impresiones por pantalla.

### Ejemplo

```
$ echo ¡He aquí un libro de programación shell!  
¡He aquí un libro de programación shell!  
$
```

Ciertos caracteres tienen un significado especial cuando se ponen entre comillas (apóstrofes o comillas dobles). Estos caracteres son los caracteres de escape.

- El comando **echo** de bash debe ser usado con la opción **-e** para que se realice la interpretación de los caracteres de escape.

### a. El carácter "\n"

Sirve para provocar un salto de línea.

#### Ejemplo con un shell diferente de bash

```
$ echo "He aquí un salto de línea\ny otro\ny el salto de línea natural del comando echo"  
He aquí un salto de línea  
y otro  
y el salto de línea natural del comando echo  
$
```

Las comillas son obligatorias:

```
$ echo a\nb  
a  
b  
$
```

#### Ejemplos con el shell bash

```
$ echo "a\nb"  
a\nb  
$ echo -e "a\nb"  
a  
b  
$
```

### b. El carácter "\c"

Sirve para eliminar el salto de línea natural del comando **echo**.

- El carácter **"\c"** se debe situar obligatoriamente en la última posición del argumento de **echo** (justo antes de las comillas de cierre).

### Ejemplos con un shell diferente de bash

```
$ echo "Primera línea" ; echo "Segunda línea"
Primera Línea
Segunda Línea
$ echo "Primera línea\c" ; echo "Segunda línea\c"
Primera líneaSegunda línea$
```

### Ejemplos con el shell bash

La opción **-e** es indispensable para interpretar "**\c**" correctamente.

```
$ echo -e "Primera línea\c" ; echo -e "Segunda línea\c"
Primera líneaSegunda línea$
```

La opción **-n** reemplaza "**\c**".

```
$ echo -n "Primera línea" ; echo -n "Segunda línea"
Primera líneaSegunda línea$
```

## **c. El carácter "\t"**

Permite mostrar una tabulación.

### Ejemplo con un shell diferente de bash

```
$ echo "A continuación 2 tabulaciones\t\tlisto ..."
A continuación 2 tabulaciones      listo ...
$
```

### Ejemplo con un shell bash

```
$ echo -e "A continuación 2 tabulaciones\t\tlisto ..."
A continuación 2 tabulaciones      listo ...
$
```

## **d. Listado de caracteres de escape**

Carácter de escape	Significado
\\	Contrabarra
\a	Tono
\b	Borrado del carácter anterior
\c	Eliminación del salto de línea al final de la línea
\f	Salto de página
\n	Salto de línea
\r	Retorno de carro
\t	Tabulación horizontal
\v	Tabulación vertical
\0xxx	Valor de un carácter expresado en octal

## **2. Los comandos print y printf**



Estos dos comandos de impresión son más recientes y se presentarán posteriormente (ver capítulo Aspectos avanzados de la programación shell - Comandos de salida). Su disponibilidad depende del shell usado.

# El carácter ~ (tilde)

ksh	bash
-----	------

El carácter ~ representa el directorio de inicio del usuario actual.

## Ejemplos

*El usuario actual se llama **crístina**:*

```
$ id
uid=505(cristina) gid=505(ociensa)
```

*El directorio actual es /tmp:*

```
$ pwd
/tmp
```

*Copia del archivo **/tmp/f1** en el directorio inicial (/home/cristina):*

```
$ cp f1 ~
```

*Copia del archivo **/tmp/f1** en el directorio /home/cristina/docs:*

```
$ cp f1 ~/docs
```

Si el carácter ~ está inmediatamente seguido de una palabra, esta última se considera como un nombre de usuario.

## Ejemplos

*Copia del archivo **f1** en el directorio de inicio del usuario **sebastian** (suponemos que se dispone de los permisos adecuados):*

```
$ cp f1 ~sebastian
$ ls /home/sebastian/f1
f1
$
```

*Copia del archivo **f1** en el directorio /home/sebastian/dir:*

```
$ cp f1 ~sebastian/dir
$ ls /home/sebastian/dir
f1
```

## El comando interno **cd**

ksh	bash
-----	------

A continuación presentamos las sintaxis particulares del comando **cd**.

### Ejemplos

El comando **cd** sin argumento permite al usuario volver a su directorio de inicio:

```
$ cd
```

Lo mismo que utilizando el carácter **~** :

```
$ cd ~
```

Cambiar al directorio de inicio del usuario **sebastian**:

```
$ pwd  
/home/cristina  
$ cd ~sebastian  
$ pwd  
/home/sebastian
```

Volver al directorio anterior con **cd -**:

```
$ cd -  
/home/cristina
```

# Sustitución de nombres de archivos

Muchos comandos toman nombres de archivo como argumento. Estos últimos se pueden citar literalmente o se pueden especificar de forma más genérica. El shell propone un cierto número de caracteres especiales que permiten fabricar expresiones utilizadas como modelos de nombres de archivo.

## 1. Expresiones básicas

### a. El carácter \*

Representa un conjunto de caracteres cualquiera.

#### Ejemplo

```
$ ls
f12 f1.i FICa fic.c fic.s miscript.pl MISCRIP.T.pl ours.c
```

Mostrar todos los nombres de archivo que terminen por `.c`:

```
$ ls *.c
fic.c ours.c
```

Mostrar todos los nombres de archivo que comiencen por la letra `f`:

```
$ ls f*
f12 f1.i fic.c fic.s
```

### b. El carácter ?

Representa un carácter cualquiera.

#### Ejemplos

Mostrar todos los nombres de archivo que tengan una extensión compuesta de un solo carácter:

```
$ ls *.*
f1.i fic.c fic.s ours.c
```

Mostrar todos los nombres de archivo compuestos de cuatro caracteres:

```
$ ls ????
f1.i FICa
```

### c. Los caracteres [ ]

Los corchetes permiten especificar la lista de caracteres que se esperan en una posición concreta en el nombre del archivo. También es posible usar las nociones de intervalo y negación.

#### Ejemplos

Archivos cuyo nombre empiece por `f u o` y termine por el carácter `.` seguido de minúscula:

```
$ ls [fo]*.[a-z]
f1.i fic.c fic.s ours.c
```

Archivos cuyo nombre tiene en el segundo carácter una mayúscula, una cifra o la letra *i*. Los dos primeros caracteres tendrán a continuación una cadena cualquiera:

```
$ ls ?[A-Z0-9i]*
f12 f1.i FICa fic.c fic.s MISCRIPt.pl
```

También es posible expresar la negación de todos los caracteres especificados en el interior de un par de corchetes. Se realiza mediante la adición de una **!** en la primera posición de su interior.

### Ejemplo

*Nombres de archivo que no comiencen por minúscula:*

```
$ ls [!a-z]*
FICa MISCRIPt.pl
```

Por supuesto, es posible especificar múltiples expresiones en la línea de comandos. Estas deberán estar separadas por un espacio.


### Ejemplo

*Eliminar todos los archivos cuyo nombre termine por .c o .s:*

```
$ rm -i *.c *.s
rm: remove `fic.c'? y
rm: remove `ours.c'? y
rm: remove `fic.s'? y
$
```

## 2. Expresiones complejas

ksh	bash
-----	------

 Para usar estas expresiones en bash, es necesario previamente activar la opción **extglob** con el comando **shopt** (`shopt -s extglob`).

Estas expresiones sirven no solo para generar nombres de archivo, sino que además se usan en otros dos contextos que serán detallados más adelante (ver capítulo Las bases de la programación shell - Ejecución de verificaciones y Las estructuras de control). A continuación se muestra el contenido del directorio que se usará en los ejemplos siguientes:

```
$ ls
fic          fic866866.log    fic867.log      typescript
fic.log      fic866866866.log  fic868.log
fic866.log   fic866868.log    readme.txt
```

### a. **?(expresión)**

La expresión estará presente 0 o 1 veces.

### Ejemplo

*Archivos cuyo nombre comience por "fic" seguido de 0 o 1 ocurrencias de "866", seguido de ".log":*

```
$ ls fic?(866).log
fic.log  fic866.log
```

## b. \*(expresión)

La expresión estará presente entre 0 y n veces.

### Ejemplo

Archivos cuyo nombre comience por "fic", seguido de 0 a n ocurrencias de "866", seguido de ".log":

```
$ ls fic*(866).log
fic.log      fic866.log   fic866866.log  fic866866866.log
```

## c. +(expresión)

La expresión estará presente entre 1 y n veces.

### Ejemplo

Archivos cuyo nombre comience por "fic", seguido de al menos 1 ocurrencia de "866", seguido de ".log":

```
$ ls fic+(866).log
fic866.log   fic866866.log  fic866866866.log
```

## d. @(expresión)

La expresión estará presente solo 1 vez.

### Ejemplo

Archivos cuyo nombre comience por "fic", seguido de exactamente una ocurrencia de "866", seguido de ".log":

```
$ ls fic@(866).log
fic866.log
```

## e. !(expresión)

La expresión no estará presente.

### Ejemplos

Archivos cuyo nombre comience por "fic", seguido de una expresión que no sea la cadena "866", seguido de ".log":

```
$ ls fic!(866).log
fic.log      fic866866866.log  fic867.log
fic866866.log  fic866868.log    fic868.log
$
```

Archivos cuyo nombre no empiece por "fic":

```
$ ls !(fic*)
readme.txt typescript
$
```

## f. Alternativas

Una barra vertical en el interior de una expresión compleja significa "o bien".

? (expresión|expresión|...)  
\* (expresión|expresión|...)  
+ (expresión|expresión|...)  
@ (expresión|expresión|...)  
! (expresión|expresión|...)

### Ejemplos

*Archivos cuyo nombre comience por "fic", seguido de "866" o de "867", seguido de ".log":*

```
$ ls fic@(866|867).log
fic866.log fic867.log
$
```

*Archivos cuyo nombre comience por "fic", seguido de 1 a n ocurrencias de "866" u "868", seguido de ".log":*

```
$ ls fic+(866|868).log
fic866.log    fic866866866.log fic868.log
fic866866.log  fic866868.log
$
```

*Archivos cuyo nombre comience por "fic", seguido de 1 ocurrencia de 1 a n veces "866" o de 1 a n veces "868", seguido de ".log":*

```
$ ls fic@(+ (866) |+(868)) .log
fic866.log    fic866866.log    fic866866866.log fic868.log
$
```

## 3. Interpretación del shell

Todos estos caracteres especiales se sustituyen por el shell y no por el comando. La figura 3 representa el mecanismo interno asociado.

### Ejemplo

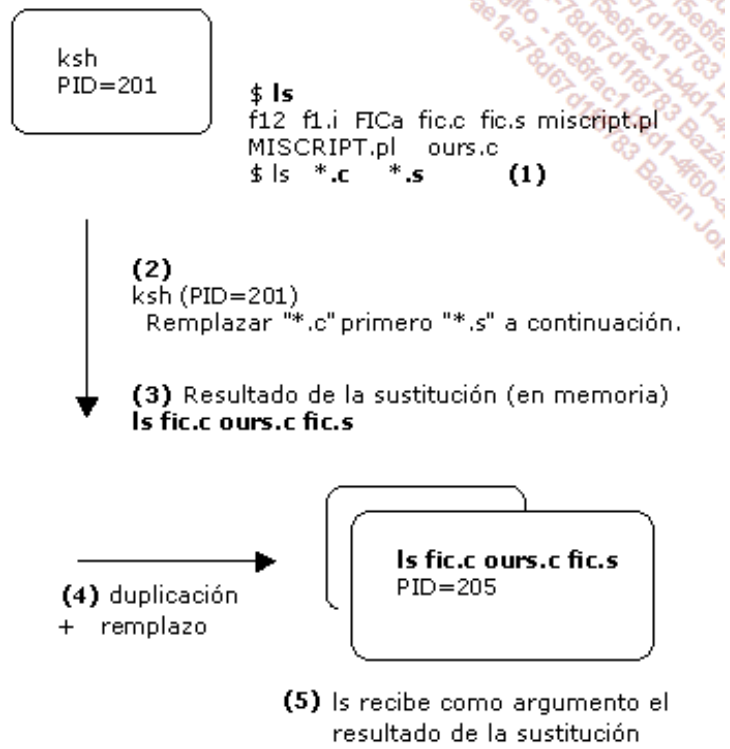


Figura 3: Tratamiento de los caracteres de generación de nombres de archivo



## Separador de comandos

El carácter especial ; del shell permite escribir varios comandos en una misma línea. Los comandos se ejecutan secuencialmente.

### Ejemplo

```
$ pwd
/home/cristina
$ ls
$ mkdir dir ; cd dir ; pwd
/home/cristina/dir
```

# Redirecciones

Las redirecciones se suelen usar en los comandos Unix. Permiten recuperar el resultado de uno o varios comandos en un archivo o provocar la lectura de un archivo por un comando. Esta sección explica detalladamente las diferentes sintaxis posibles con sus mecanismos internos asociados.


Las redirecciones son ejecutadas por el shell.

## 1. Entrada y salidas estándar de los procesos

Los procesos de Unix tienen, por defecto, su archivo terminal abierto tres veces, mediante tresdescriptores de archivo diferentes.

### a. Entrada estándar

Al descriptor de archivo 0 se le llama también **entrada estándar del proceso**. Los procesos que esperan la entrada de información por parte del usuario desencadenan una solicitud de lectura sobre el descriptor 0. Si este último está asociado al terminal, que es como está por defecto, se materializa al usuario como una petición de lectura del teclado.

 La mayoría de los comandos utilizan la entrada estándar para desencadenar una lectura. Sin embargo, hay excepciones. Por ejemplo, el comando `passwd` abre el archivo terminal con otro descriptor.

### b. Salida estándar

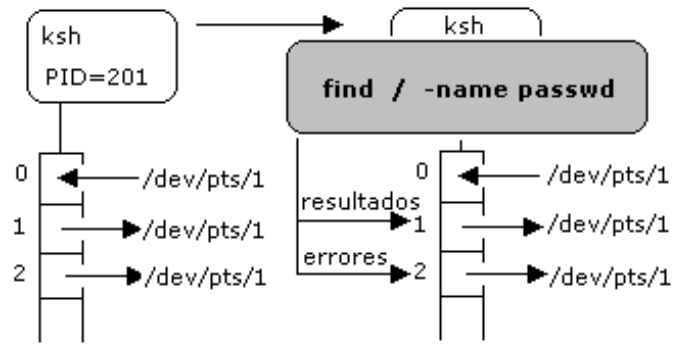
Al descriptor de archivo 1 se le llama también **salida estándar del proceso**. Por convenio, un proceso que desea enviar un mensaje de resultado al usuario tiene que hacerlo a través del descriptor 1. Si este último está asociado al terminal, que es como está por defecto, se materializa al usuario como una impresión por pantalla.

### c. Salida de error estándar

Al descriptor de archivo 2 se le llama también **salida de error estándar del proceso**. Por convenio, un proceso que desea enviar un mensaje de error al usuario tiene que hacerlo a través del descriptor 2. Si este último está asociado al terminal, que es como está por defecto, se materializa al usuario como una impresión por pantalla.

## 2. Herencia

Los descriptores de archivos son heredados en la duplicación (`fork()`) y el remplazo (`exec()`). La figura 4 muestra que el shell `ksh` hijo, después del comando `find`, hereda la tabla de descriptores de archivos del shell padre. Este comando envía por la salida estándar los resultados y por la salida de error estándar la lista de directorios que el usuario no tiene permisos para explorar. Como las salidas 1 y 2 están orientadas al mismo terminal, las impresiones en pantalla se entrelazan. Una de las características interesantes de las redirecciones es la posibilidad de separar los mensajes procedentes de cada descriptor.



```
$ find / -name passwd
find: cannot read dir /lost+found: Permission denied
/var/adm/passwd
/usr/bin/passwd
find: cannot read dir /usr/aset: Permission denied
/etc/default/passwd
/etc/passwd
...
$
```

Figura 4: Herencia de la tabla de descriptores de archivos

### 3. Redirección de las salidas en escritura

La redirección en escritura permite enviar las impresiones vinculadas a un descriptor en concreto; no al terminal, sino a un archivo.

#### a. Salida estándar

##### Redirección simple

##### Sintaxis

```
$ comando 1> archivo
```

o lo que es lo mismo:

```
$ comando > archivo
```

El nombre del archivo se expresa con ruta relativa o absoluta.

Si el archivo no existe, se crea. Si el archivo ya existe, se sobrescribe.

➤ La operación de creación o sobrescritura del archivo se realiza por parte del shell.

##### Ejemplo

Recuperar el resultado del comando **ls** en el archivo **resu**:

```
$ ls > resu
$ cat resu
FIC
error
salidal
resu
```

```
$
```

## Redirección doble

Permite concatenar los mensajes resultantes de un comando al contenido de un archivo existente.

### Sintaxis

```
$ comando 1>> archivo
```

o lo que es lo mismo:

```
$ comando >> archivo
```

Si el archivo no existe, se crea. Si el archivo ya existe, se abre en modo adición.

### Ejemplo

Añadir el resultado del comando **date** al final del archivo **resu** creado anteriormente:

```
$ date >> resu
$ cat resu
FIC
error
salidal
resu
Tue Jan 21 18:31:56 WET 2014
$
```

## b. Salida de error estándar

### Redirección simple

#### Sintaxis

```
$ comando 2> archivo
```

### Ejemplo

Redirección de la salida de error estándar. Los mensajes de error van al archivo **error** y los resultados quedan en pantalla:

```
$ find / -name passwd 2> error
/var/adm/passwd
/usr/bin/passwd
/etc/default/passwd
/etc/passwd
$ cat error
find: cannot read dir /lost+found: Permission denied
find: cannot read dir /usr/aset: Permission denied
...
$
```

### Redirección doble

Permite concatenar los mensajes de error de un comando al contenido de un archivo existente.

#### Sintaxis

```
$ comando 2>> archivo
```

### Ejemplo

Concatenación de los mensajes de error de **ls -z** al final del archivo **error**:

```
$ ls -z
ls: illegal option -- z
usage: ls -lRaAdCxmnlgrtucpFbqisfL [files]
$ ls -z 2>> error
$ cat error
find: cannot read dir /lost+found: Permission denied
find: cannot read dir /usr/aset: Permission denied
ls: illegal option -- z
usage: ls -lRaAdCxmnlgrtucpFbqisfL [files]
$
```

### c. Salida estándar y salida de error estándar

Es posible redirigir múltiples descriptores en una misma línea de comandos.

```
$ comando 1> archivo_a 2> archivo_b
```

o

```
$ comando 2> archivo_b 1> archivo_a
```

➤ Las redirecciones son siempre tratadas de izquierda a derecha. En este caso, el orden de escritura de las dos redirecciones no importa, pero no siempre será así (ver el punto Redirecciones - Redirecciones avanzadas).

### Ejemplo

```
$ find / -name passwd 1> resu 2> error
$ cat resu
/var/adm/passwd
/usr/bin/passwd
/etc/default/passwd
/etc/passwd
$ cat error
find: cannot read dir /lost+found: Permission denied
find: cannot read dir /usr/aset: Permission denied
...
$
```

### d. Protección ante borrado involuntario de un archivo

ksh
-----

bash
------

La opción **noclobber** del shell permite protegerse de un borrado involuntario de archivo. Esta opción se encuentra desactivada por defecto. Puede configurarse de forma permanente en los archivos **.kshrc** o **.bashrc** (ver Configuración del entorno de trabajo - Los archivos de entorno).

### Ejemplo

```
$ ls resu
resu
$ set -o noclobber
$ date > resu
-bash: resu : no se puede sobrescribir un archivo existente
```

Para forzar el borrado, debemos usar el símbolo de redirección >|:

```
$ ls >| resu
```

### e. Eliminar las impresiones por pantalla

Todas las plataformas Unix poseen un archivo especial llamado **/dev/null** que permite hacer desaparecer las impresiones por pantalla. Este archivo se crea como un periférico y no tiene contenido. Por lo tanto, se puede considerar que está siempre vacío.

#### Ejemplo

```
$ find / -name passwd 1> resu 2> /dev/null
$ cat resu
/var/adm/passwd
/usr/bin/passwd
/etc/default/passwd
/etc/passwd
$ ls -lL /dev/null
crw-rw-rw- 1 root  sys   13, 2 Jan 21 17:22 /dev/null
$ cat /dev/null
$
```

### f. Mecanismo interno

#### Con un comando externo

Es el shell hijo el que se encarga de las redirecciones (ver figura 5 y figura 6).

```
$ find / -name passwd 1>resu 2> /dev/null
```

1ª etapa: Las redirecciones se establecen por el shell hijo

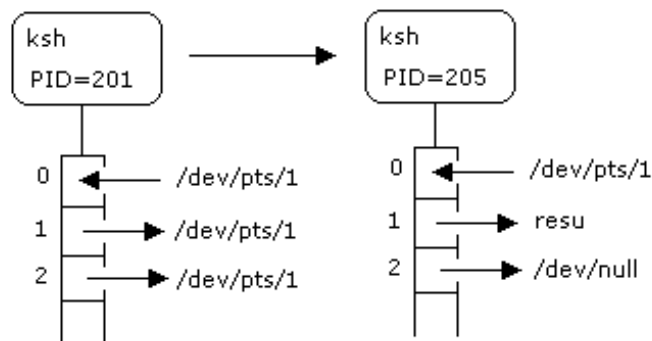


Figura 5: Mecanismo interno de las redirecciones en escritura - Primera etapa

2ª etapa: El shell hijo se reemplaza por find, que hereda la tabla de los descriptores de archivo del shell

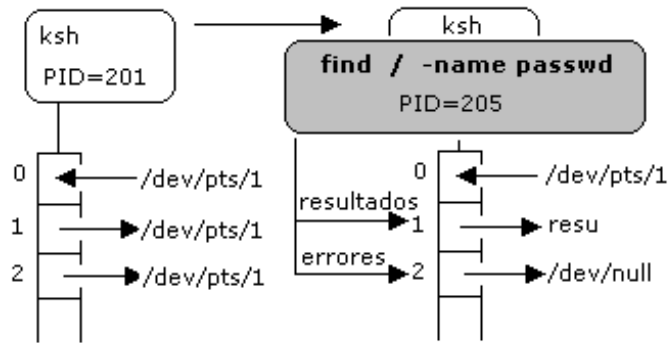


Figura 6: Mecanismo interno de las redirecciones en escritura - Segunda etapa

### Con un comando interno

Un comando interno se ejecuta por el shell actual. Este último se gestiona él mismo las redirecciones. Para ello, se guarda una copia de las asociaciones **descriptor-archivo** actuales, conecta los descriptores a los archivos pedidos, ejecuta el comando y finalmente restaura el entorno **descriptor-archivo** anterior.

## 4. Redirección de la entrada estándar

La redirección de la entrada estándar concierne a los comandos que usan el descriptor 0, es decir, aquellos que esperan una entrada de datos por el teclado.

### Ejemplo

```
$ mail olivia
Cita a las 13 horas en el restaurante      (Entrada estándar)
Cristina                                  (Entrada estándar)
^d                                         (Entrada estándar)
$
```

El comando **mail** lee la entrada estándar hasta la recepción de un final de archivo (teclas ^d). Los datos introducidos serán enviados al buzón del usuario **olivia**.

Si se desea hacer leer al comando **mail**, no del teclado, sino del contenido de un archivo, es suficiente con conectar el descriptor 0 al archivo deseado.

### **Sintaxis**

```
$ comando 0< archivo_mensaje
```

O lo que es lo mismo

```
$ comando < archivo_mensaje
```

### Ejemplo

```
$ cat mensaje
Cita a las 13 horas en el restaurante
Cristina
```

```
$ mail olivia < mensaje
$
```

Las figuras 7 y 8 representan el mecanismo interno asociado.

```
$ mail olivia < mensaje
```

1ª etapa: La redirección se establece por el shell hijo

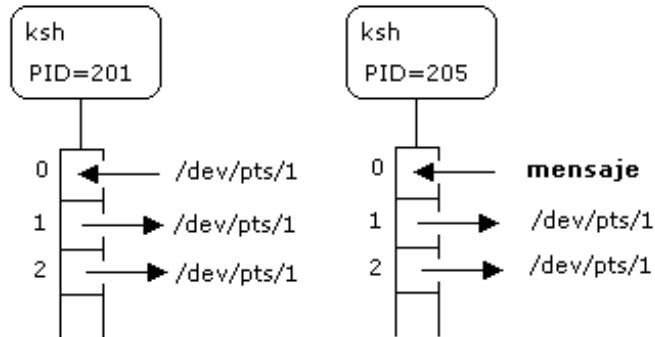


Figura 7: Mecanismo interno de la redirección en lectura - Primera etapa

2ª etapa: El comando mail realiza una lectura del descriptor 0

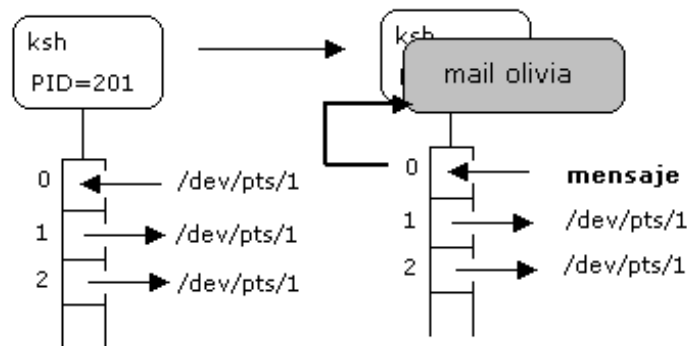


Figura 8: Mecanismo interno de la redirección en lectura - Segunda etapa

## 5. Redirecciones avanzadas

### a. Redirigir los descriptores 1 y 2 hacia el mismo archivo

Para enviar la salida estándar y la salida de error estándar hacia el mismo archivo, es necesario emplear una sintaxis particular. A continuación, se muestra lo que no hay que escribir y la razón por la que no funcionará.

#### Sintaxis incorrectas

```
$ comando 1> archivo 2> archivo
$ comando 1> archivo 2>> archivo
```

El problema no reside en el hecho de abrir dos veces el mismo archivo (es perfectamente legal dentro del mismo proceso), sino en que hay un offset (posición actual dentro del archivo) asociado a cada apertura. ¿Cuáles son las consecuencias?

- La secuencia de resultados en el archivo no será forzosamente representativa del orden en



el que se desarrollan los eventos.

- Los resultados emitidos a través de los descriptores 1 y 2 corren el riesgo de sufrir superposiciones.

Las figuras 9 y 10 presentan el mecanismo interno asociado al comando siguiente:

```
$ find / -name passwd 1> resu 2> resu
```

### **Primera etapa (ver figura 9)**

#### **Tratamiento de la redirección 1> resu:**

El shell abre el archivo **resu** (el archivo se crea con un tamaño igual a 0) y lo asocia al descriptor 1 (**1, 2, 3, 4, 5**). Cuando un proceso abre un archivo, siempre hay un registro asignado en la tabla de archivos abiertos del núcleo (**2**) (esta recoge todas las aperturas de archivo del sistema en un momento dado). El registro contiene el modo de apertura del archivo (en este caso, escritura), así como la posición actual dentro del archivo (en este caso, la escritura en el descriptor 1 empezará en el inicio del archivo). La posición 1 (**1**) de la tabla de descriptores de archivos del proceso contiene un puntero (p2) hacia el registro de la tabla de archivos abiertos.

#### **Tratamiento de la redirección 2> resu:**

El shell abre de nuevo el archivo **resu** en escritura (**6, 7, 3, 4, 5**) (el archivo es sobrescrito, lo que no presenta un problema puesto que el tamaño era 0) y lo asocia al descriptor 2. Un nuevo registro se crea en la tabla de archivos abiertos (p3) (**7**). Las operaciones de escritura posteriores en el descriptor 2 se harán a partir del inicio del archivo.

- En el caso de **2>> resu**, el archivo se abrió en modo adición: colocación al final del archivo (por lo tanto en el byte 0). Las escrituras posteriores en el descriptor 2 garantizan la escritura al final del archivo. Esta solución puede producir un resultado aceptable en algunos casos, pero en el resto sigue siendo incorrecta (2 posiciones diferentes).

```
$ find / -name passwd 1> resu 2> resu
```

1ª etapa: Tratamiento de las redirecciones por el shell hijo

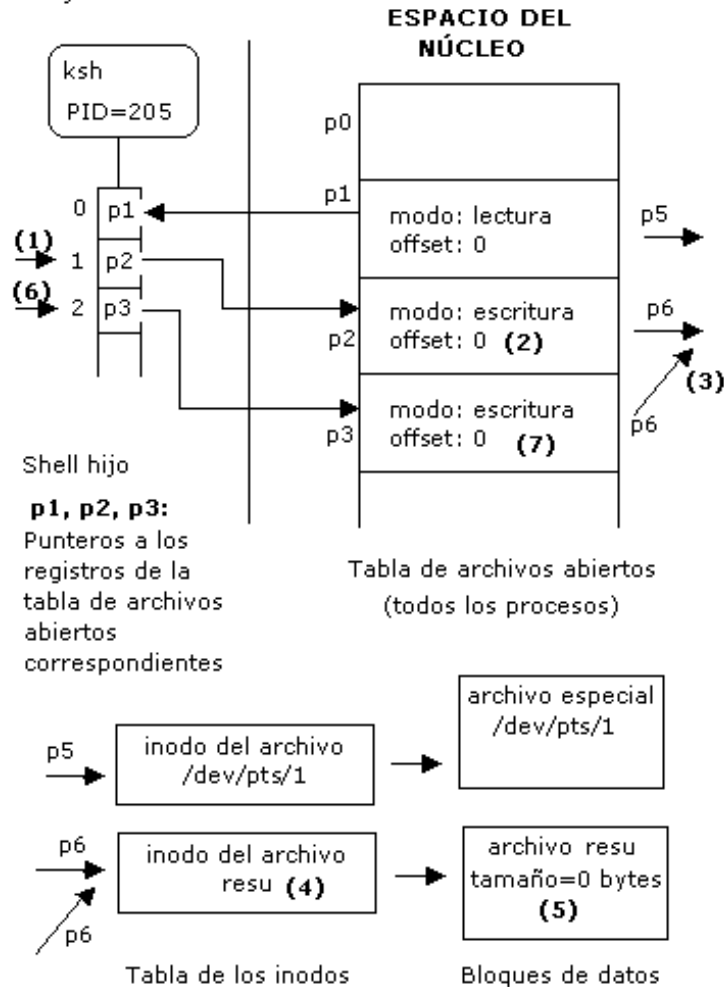


Figura 9: Gestión de la colocación en las redirecciones - Primera etapa

## **Segunda etapa (ver figura 10)**

El shell se reemplaza por el comando **find**, que hereda la tabla de descriptores de archivos. Simulación del funcionamiento del comando **find**:

- **find** envía al descriptor 1 un mensaje resultado de 20 bytes **(1, 2, 3...)**.
- Estos 20 bytes se escriben a partir de la posición 0 **(2)**. Después, el valor del offset es 20**(3)**, que será utilizado en la próxima escritura en la salida estándar.
- El archivo crece 20 bytes **(4)**.
- **find** envía al descriptor 2 un mensaje de error de 10 bytes **(5)**.
- Estos se escribirán al inicio del archivo (el offset del descriptor 2 vale 0 **(7)**), y por lo tanto sobrescribirán el mensaje escrito anteriormente **(8)**. El offset de la salida de error pasará a ser 10 **(7)**.

Conclusión: el archivo es inutilizable porque los mensajes de error y resultado se superponen.

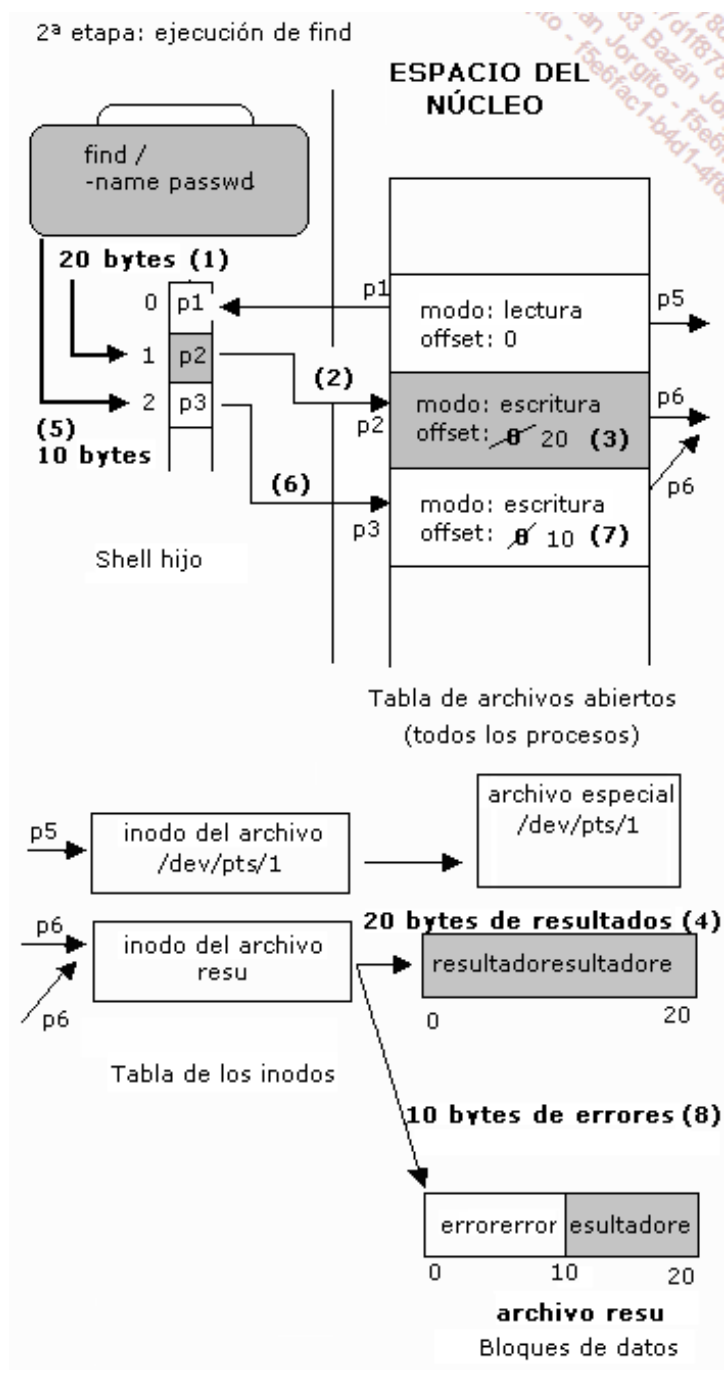


Figura 10: Gestión de la colocación en las redirecciones - Segunda etapa

### Sintaxis correctas

Para obtener un resultado correcto, hay que usar una de las dos sintaxis siguientes:

```
$ comando 1> archivo 2>&1
```

o

```
$ comando 2> archivo 1>&2
```

La figura 11 representa el mecanismo interno generado por la primera sintaxis:

#### **Tratamiento de la redirección 1 > resu:**

El mismo mecanismo que el anterior: creación del archivo **resu**, asignación de un registro en la tabla de archivos abiertos, offset a 0 (1).

## Tratamiento de la redirección 2>&1:

Para traducir esta expresión al castellano, se puede decir que el descriptor 2 está redirigido al descriptor 1 (representado por &1). El concepto importante que hay que comprender es que el shell, gracias a esta sintaxis, duplica **(2)** simplemente la dirección del descriptor 1 en el descriptor 2. Por lo tanto, los dos descriptores apuntan (**p2**) al mismo registro de la tabla de archivos abiertos y, por consiguiente, comparten el mismo offset. No hay asignación de un nuevo registro en la tabla de archivos abiertos del núcleo. Las escrituras posteriores, ya sean emitidas a través de la salida estándar o de la salida de error estándar, se servirán del mismo offset.

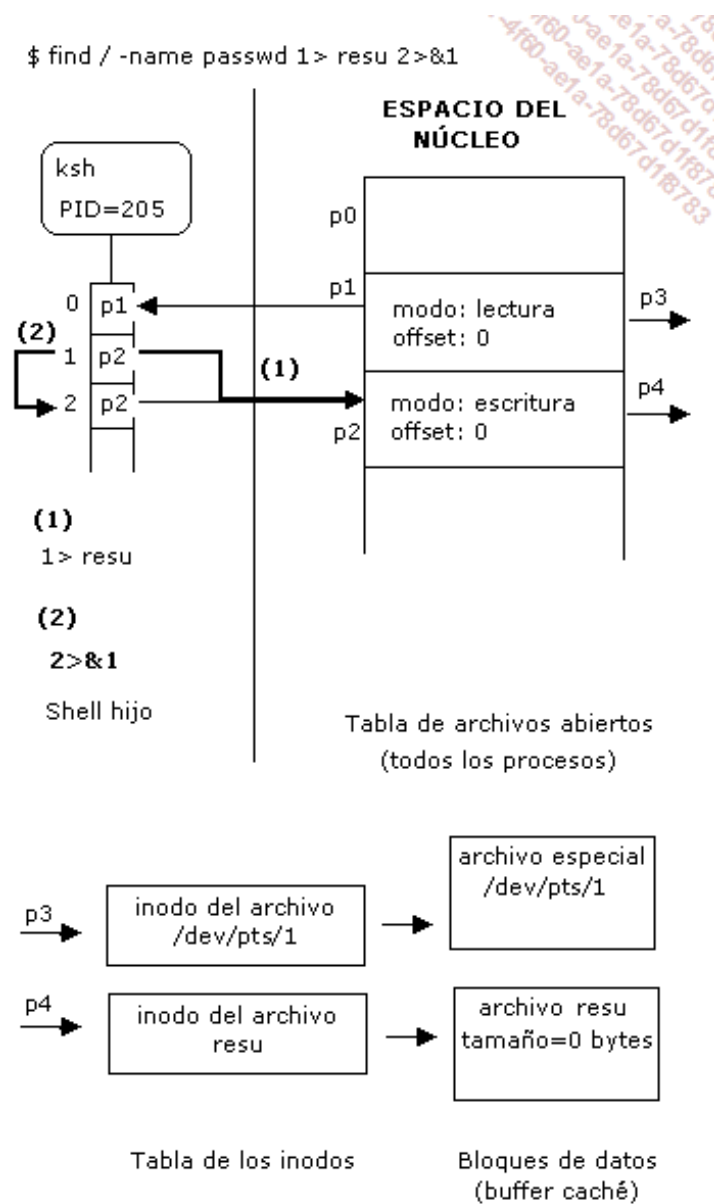


Figura 11: Redirección de los dos descriptores al mismo archivo

La sintaxis `1> resu 2>&1` no es equivalente a `2>&1 1> resu`. En el segundo caso, la salida de error estándar se redirige hacia la salida estándar, es decir, el terminal. Después, la salida estándar se asocia al archivo **resu**. Conclusión: los mensajes de error se dirigen hacia el terminal y los mensajes de resultado hacia el archivo **resu**.

## b. La redirección doble en lectura

Se usa principalmente en los scripts de shell. Permite conectar la entrada estándar de un comando a una porción del script.

## Primera sintaxis

```
comando <<ETIQUETA
datos
datos
datos
ETIQUETA
```

## Segunda sintaxis

```
comando <<-ETIQUETA

datos
datos
datos
    ETIQUETA
```

El símbolo situado después de los caracteres < es una declaración de etiqueta. Esta etiqueta se usará para marcar el final de los datos que tendrá que leer el comando. Las líneas insertadas entre las dos palabras **ETIQUETA** serán enviadas a la entrada estándar del comando.

### Ejemplo

```
$ mail olivia <<FIN
> Cita a las 13 horas en el restaurante
> Cristina
> FIN
```

## Algunos apuntes sintácticos

- En la primera sintaxis, la etiqueta debe estar obligatoriamente pegada al margen izquierdo.
- En la segunda sintaxis, el hecho de colocar un carácter - delante de la etiqueta permite al usuario poner la última etiqueta después de una o varias tabulaciones.
- Las etiquetas tienen que estar seguidas inmediatamente de un salto de línea.

### **c. Cierre de un descriptor**

<b>Descriptor cerrado</b>	<b>Sintaxis</b>	<b>Efecto</b>
0	comando <&-	La entrada estándar del comando se cierra, con lo que no se podrá usar para recibir más datos.
1	comando >&-	La salida estándar del comando se cierra, con lo que no se podrá usar. Ningún mensaje se podrá mostrar por pantalla.
2	comando 2>&-	La salida de error estándar del comando se cierra, con lo que no se podrá usar. Ningún mensaje de error se podrá mostrar por pantalla.

# Tuberías de comunicación

Una tubería (**pipe** en inglés) permite la comunicación entre dos procesos. La tubería se representa con una barra vertical (pulsar [Alt Gr] **1** en un teclado QWERTY) situada entre dos comandos Unix. El resultado del comando de la izquierda va a parar a la tubería, mientras que el comando de la derecha leerá los datos que hay en ella para tratarlos.

Las figuras 12 y 13 representan el mecanismo interno asociado a la tubería de comunicación.

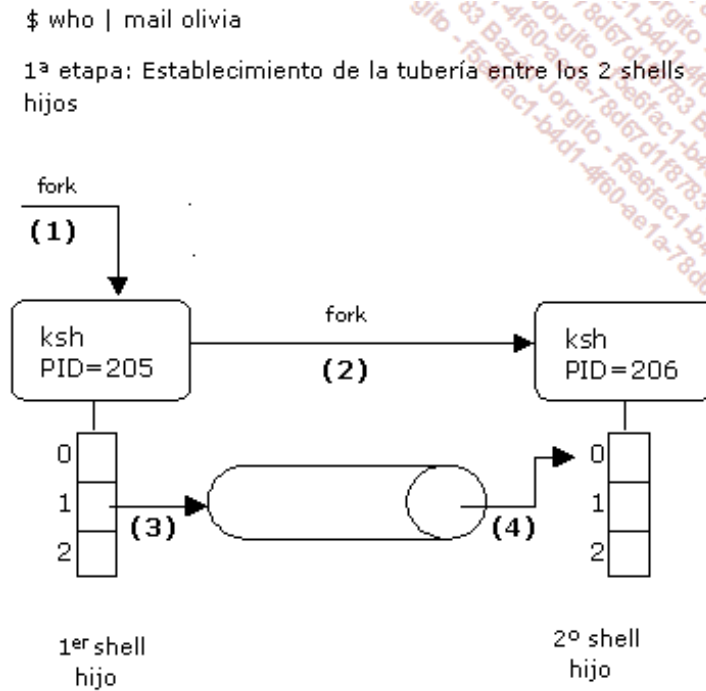


Figura 12: Mecanismo interno de la tubería de comunicación - Primera etapa

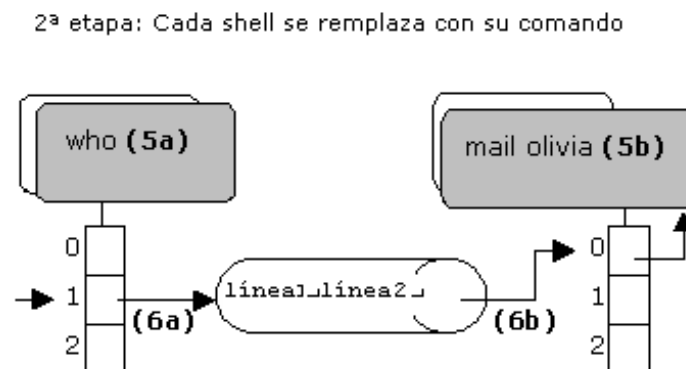


Figura 13: Mecanismo interno de la tubería de comunicación - Segunda etapa

Sean cuales sean los comandos presentes a cada lado de la tubería, el shell de trabajo detecta el carácter | en la línea de comandos y creará un shell hijo **(1)** que, a su vez, hace lo mismo **(2)**. El primer shell hijo (PID=205) disocia su salida estándar del terminal y la conecta a la entrada de la tubería **(3)**. El segundo shell hijo (PID=206) disocia su entrada estándar del terminal y la conecta a la salida de la tubería **(4)**.

Cada shell hijo se reemplaza con su comando **(5a y 5b)**. Cada comando empieza entonces a ejecutarse. Cuando el comando **who** escribe en su salida estándar, los mensajes van dentro de la tubería **(6a)**. Paralelamente, el comando **mail** lee su entrada estándar **(6b)**, lo que provoca la extracción de los datos contenidos en la tubería.

## Algunos aspectos importantes

- La salida de error estándar del comando de la izquierda no va a la tubería.
- Para que el uso de una tubería tenga sentido, es necesario que el comando situado a la izquierda de la tubería envíe datos a su salida estándar y que el comando situado a la derecha lea su entrada estándar.

### 1. Comandos que no leen su entrada estándar

Hay una serie de comandos Unix que no interesa ponerlos detrás de una tubería, ya que no realizan lecturas en su entrada estándar. Este es el caso, por ejemplo, de los comandos siguientes: ls, who, find, chmod, cp, mv, rm, ln, mkdir, rmdir, date, kill, file, type, echo...

### 2. Comandos que leen su entrada estándar

Los comandos que leen su entrada estándar son fácilmente identificables, puesto que requieren una entrada de datos por el teclado.

#### a. Ejemplos triviales

```
$ mail olivia
entrada de datos por teclado
entrada de datos por teclado
^d
$
$ write olivia
entrada de datos por teclado
entrada de datos por teclado
^d
$
```

Por lo tanto, estos dos comandos pueden colocarse detrás de una tubería:

```
$ who | mail olivia
$ echo "Cita para comer a las 13 horas" | write olivia
```

#### b. Caso de los filtros

En Unix, hay una serie de comandos que se agrupan bajo el nombre de filtros. Los más comunes son: grep, cat, sort, cut, wc, lp, sed, awk... Estos comandos pueden funcionar de dos maneras:

##### Primera manera

Si el comando recibe al menos un nombre de archivo como argumento, este trata el/los archivo(s) y no inicia la lectura de la entrada estándar.

##### Ejemplo

```
$ wc -l /etc/passwd
46 /etc/passwd
```

##### Segunda manera

El comando no recibe ningún nombre de archivo como argumento. En este caso, el comando trata

los datos que le lleguen por su entrada estándar.

### Ejemplo

El comando **wc** cuenta el número de líneas que llegan por su entrada estándar y muestra el resultado por su salida estándar:

```
$ wc -l
entrada de datos por el teclado      (Entrada estándar)
entrada de datos por el teclado      (Entrada estándar)
entrada de datos por el teclado      (Entrada estándar)
^d                                     (Entrada estándar)
  3                                     (Entrada estándar)
$
```

Por lo tanto, es posible colocar este comando detrás de una tubería:

```
$ who | wc -l
  4
$
```

### ¿Cómo saber si un comando lee su entrada estándar?

A continuación se muestran dos métodos que permiten saber si un comando lee su entrada estándar:

- La información está en el manual del comando.

### Ejemplo

A continuación se detalla una parte del manual del comando **wc**. Se constata que el argumento [ **file** ... ] es opcional, lo que es una primera señal.

```
$ man wc
...
SYNOPSIS
wc [ -c | -m | -C ] [ -lw ] [ file ... ]
...
```

Un poco más adelante se encuentra la explicación del argumento **file**; si el nombre del archivo se omite, el comando lee su entrada estándar.

```
...
file A path name of an input file. If no file operands
    are specified, the standard input will be used.
...
```

Otra posibilidad consiste en probar el comando sin dar ningún nombre de archivo como • argumento.

### Primer ejemplo

A continuación se prueba un comando que trata un archivo. No desencadena una lectura de la entrada estándar:

```
$ cut -d' :' -f1,3 /etc/passwd
root:0          (Salida estándar)
bin:1           (Salida estándar)
daemon:2       (Salida estándar)
adm:3          (Salida estándar)
```



```
...  
$
```

A continuación, el mismo comando sin el nombre del archivo. El comando espera una entrada de datos por el teclado:

```
$ cut -d ':' -f1,3  
1:2:3:4           (Entrada estándar)  
1:3               (Salida estándar)  
10:20:30:40      (Entrada estándar)  
10:30            (Salida estándar)  
100:200:300:400 (Entrada estándar)  
100:300          (Salida estándar)  
^d               (Entrada estándar)  
$
```

Por lo tanto, este comando puede colocarse detrás de una tubería:

```
$ echo "1:2:3:4" | cut -d ':' -f1,3  
1:3  
$
```

### Segundo ejemplo

A continuación se muestra otro comando que trata un archivo.

```
$ file /etc/passwd  
/etc/passwd: ASCII text
```

El mismo comando sin el nombre del archivo genera un mensaje de error. Por lo tanto, el nombre del archivo es obligatorio. Este comando no lee su entrada estándar y no puede ser ubicado a la derecha de un tubo:

```
$ file  
Usage: file [-bciknvzL] [-f namefile] [-m magicfiles] file...  
Usage: file -C [-m magic]
```

## Caso particular de algunos comandos

A la mayoría de los comandos no les importa si están situados detrás de una tubería o no. Para un comando determinado, la acción será siempre la misma:

### Ejemplo

**wc -l** lee su entrada estándar en los dos casos:

```
$ wc -l  
$ who | wc -l
```

Algunos comandos son la excepción de la regla. Verifican si su entrada estándar está conectada a la salida de una tubería o a un terminal. Este es el caso del comando **more**:

### Ejemplos

El comando **more** recibe el nombre de un archivo como argumento y pagina su contenido por pantalla. No lee su entrada estándar:

```
$ more /etc/passwd
```

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
...
--Más-- (43%)
```

Sin el nombre del archivo, el comando muestra un mensaje de error (iel argumento [filename] es, sin embargo, opcional!). Aquí, **more** no lee su entrada estándar:

```
$ more
Usage: more [-cdfllrsuw] [-lines] [+linenumber] [+/pattern]
[filename ...].
```

El nombre del archivo se puede omitir cuando **more** se coloca a la derecha de una tubería. En este caso, lee su entrada estándar y pagina las líneas que extrae:

```
$ cat /etc/passwd | more
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
...
--Más-- (43%)
$
```

### 3. Complementos

#### a. Encadenar tuberías

Se pueden encadenar múltiples tuberías en una línea de comandos.

##### Ejemplo

Mostrar el número de conexiones del usuario cristina:

```
$ who | grep cristina | wc -l
3
```

#### b. Duplicar las salidas

El comando **tee** permite visualizar un resultado por pantalla y guardarlo a la vez en un archivo.

##### Ejemplos

El comando **tee** muestra por su salida estándar las líneas extraídas de la tubería y las escribe en el archivo **listaarch**. Si **listaarch** ya existe, se sobrescribe:

```
$ ls | tee listaarch
Desktop
FIC
archivo
$ cat listaarch
Desktop
FIC
archivo
$
```

El resultado del comando **date** se muestra por pantalla y concatenado (adición) con el archivo **listaarch** existente:

```
$ date | tee -a listaarch
lun ene 27 17:54:21 CET 2014
$ cat listaarch
Desktop
FIC
archivo
lun ene 27 17:54:21 CET 2014
$
```

### c. Enviar la salida estándar y la salida de error estándar por la tubería

#### Ejemplos

El comando siguiente muestra un mensaje de error y una línea de resultado:

```
$ ls -l a* Z*
ls: Z*: No existe el archivo o directorio
-rw-rw-r-- 1 cristina curso 110 ene 22 14:21 archivo
$
```

Solamente la salida estándar llega a la tubería:

```
$ ls -l a* Z* | tee listaarch
ls: Z*: No existe el archivo o directorio (mostrado por ls)
-rw-rw-r-- 1 cristina curso 110 ene 22 14:21 archivo
(mostrado por tee)
$ cat listaarch
-rw-rw-r-- 1 cristina curso 110 ene 22 14:21 archivo
$
```

Usando la duplicación de descriptor (ver Redirecciones - Redirecciones avanzadas), la salida 2 se dirige a la salida 1 (terminal):

```
$ ls -l a* Z* 2>&1 | tee listaarch
ls: Z*: No existe el archivo o directorio (mostrado por tee)
-rw-rw-r-- 1 cristina curso 110 ene 22 14:21 archivo
(mostrado por tee)

$ cat listaarch
ls: Z*: No existe el archivo o directorio
-rw-rw-r-- 1 cristina curso 110 ene 22 14:21 archivo
$
```

# Agrupación de comandos

La agrupación de comandos se puede usar para:

- Redirigir la salida de varios comandos por pantalla hacia un mismo archivo o hacia una tubería.
- Ejecutar múltiples comandos en el mismo entorno.

## Ejemplo

Sólo la salida estándar del segundo comando se redirige al archivo **resultado**.

```
$ date ; ls > resultado
mar ene 28 05:16:30 CET 2014
$ cat resultado
FIC
archivo
$
```

Los paréntesis ( ) y las llaves { } permiten agrupar los comandos. En el primer caso, los comandos se ejecutan en un shell hijo, en el segundo caso en el shell actual.

## 1. Paréntesis

### Sintaxis

```
(cmdo1 ; cmdo2 ; cmdo3)
```

Con los paréntesis, un shell hijo se crea sistemáticamente y es este el que trata la línea de comandos (con duplicaciones posteriores si es necesario).

### Primer ejemplo

En este caso, el usuario se sirve de los paréntesis para redirigir la salida estándar de dos comandos:

```
$ (date ; ls) > resultado
$ cat resultado
mar ene 28 05:21:36 CET 2014
FIC
archivo
$
```

Las figuras 14, 15 y 16 representan el mecanismo interno asociado. El shell actual (PID=201) se duplica **(1)**. El shell hijo (PID=205) se ocupa primeramente de la redirección **(2)** y se duplica a continuación para la ejecución del comando externo **date (5)**. Cuando este ha terminado **(7)**, se duplica de nuevo para ejecutar **ls (8)**. Mediante el mecanismo de herencia, los dos comandos utilizan el mismo offset **(3)**. Por lo tanto, las escrituras en el archivo se suceden correctamente.

\$ ( date ; ls ) > resultado

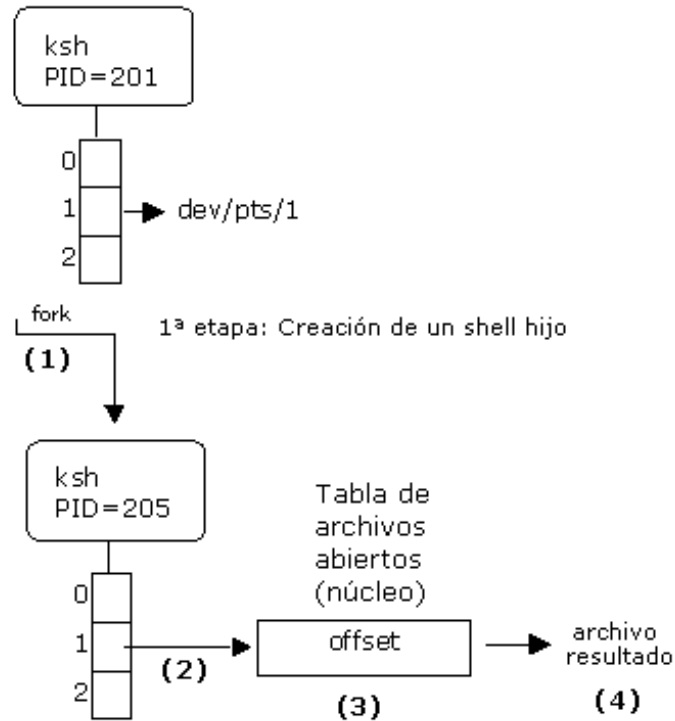


Figura 14: Primer ejemplo de agrupación con paréntesis - Primera etapa

2ª etapa: Ejecución del comando date

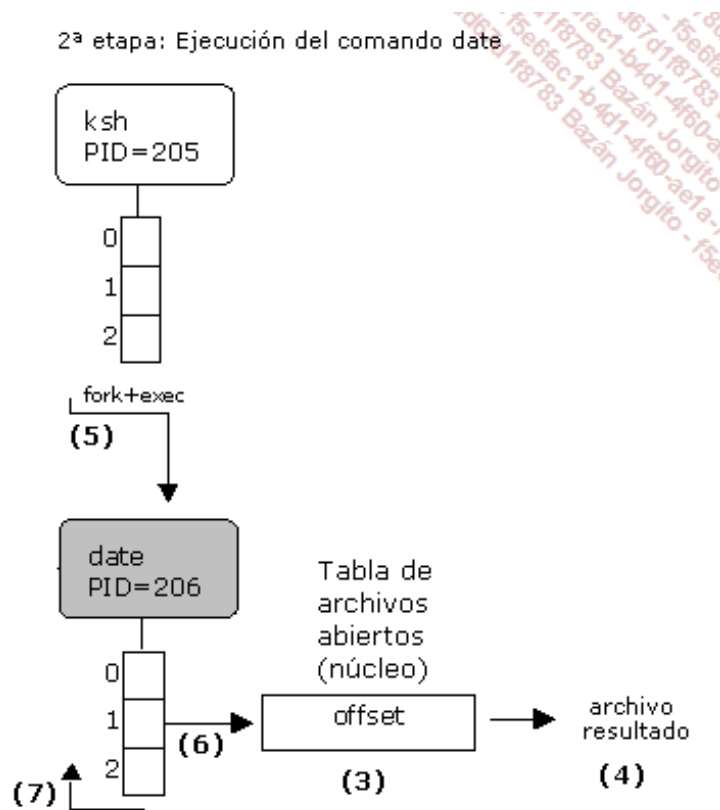


Figura 15: Primer ejemplo de agrupación con paréntesis - Segunda etapa

3ª etapa: Ejecución del comando ls

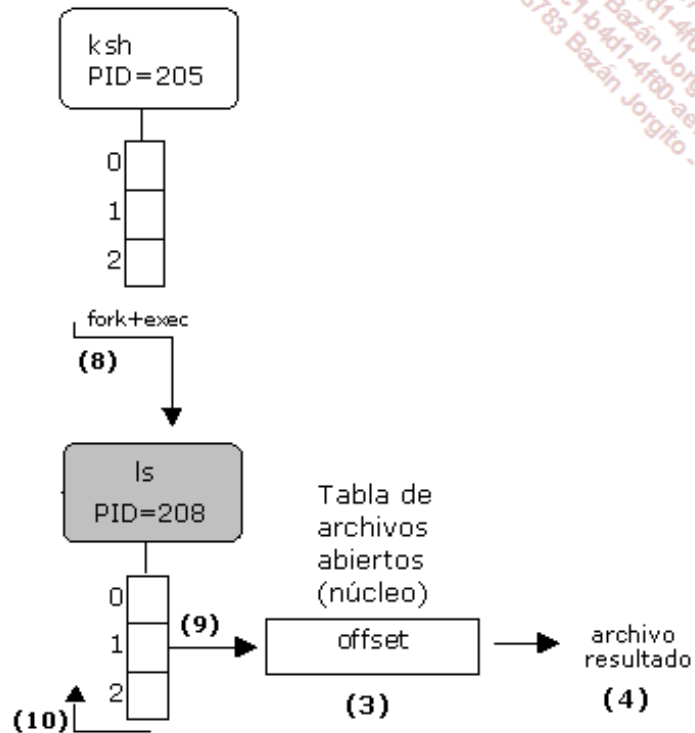


Figura 16: Primer ejemplo de agrupación con paréntesis - Tercera etapa

### Segundo ejemplo

Los comandos `pwd` y `ls` tienen como directorio actual `/tmp`:

```
$ pwd
/home/cristina
$ (cd /tmp ; pwd ; ls) > listaarch
$ cat listaarch
/tmp                               (resultado de pwd)
dcopNYSrKn                         (lista de los archivos de /tmp)
listatmp
...
```

Cuando la ejecución de los tres comandos ha terminado, el shell de primer nivel continúa su ejecución. Su directorio actual sigue siendo `/home/cristina`.

```
$ pwd
/home/cristina
$
```

Las figuras 17 y 18 representan el mecanismo interno asociado. El shell de trabajo tiene como directorio actual `/home/cristina` (1). Como en el ejemplo, hay la creación de un shell hijo (2). Este último ejecutará el comando interno `cd` (3) (el directorio actual del shell hijo cambia (4)), después el comando interno `pwd` (5) (que muestra `/tmp`) y después se duplica (6) para la ejecución del comando externo `ls`, que hereda del directorio `/tmp` (7).

Cuando todos los comandos se han ejecutado (8), el shell de primer nivel retoma el control. Su directorio actual sigue siendo `/home/cristina`.

```
$ ( cd /tmp ; pwd ; ls )
```

1ª etapa: Creación de un shell hijo

Ejecución de los comandos cd y pwd

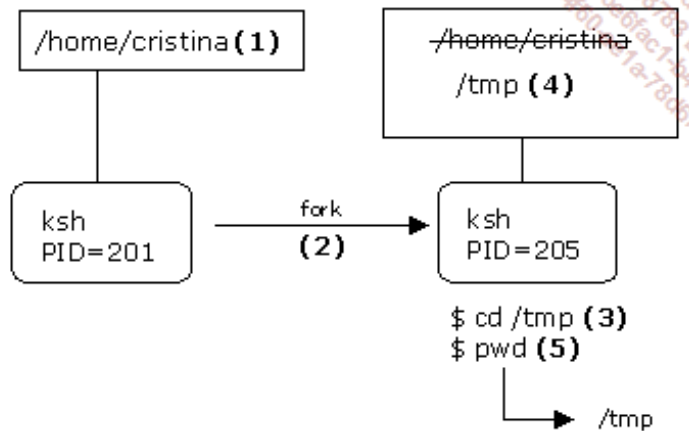


Figura 17: Segundo ejemplo de agrupación con paréntesis - Primera etapa

2ª etapa: Ejecución del comando ls

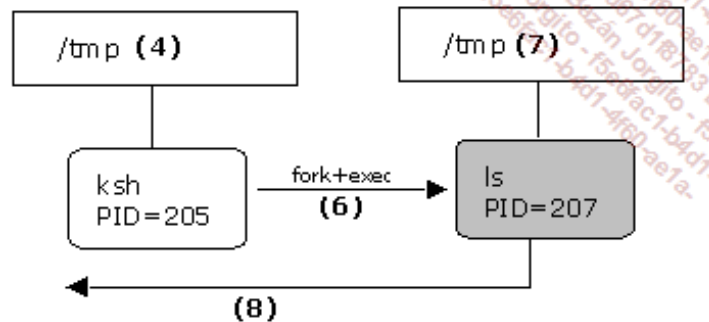


Figura 18: Segundo ejemplo de agrupación con paréntesis - Segunda etapa

## 2. Las llaves

### Sintaxis

```
{ cmdo1 ; cmdo2 ; cmdo3 ; }
```

- Las llaves de apertura y cierre tienen que estar precedidas y seguidas de un espacio.
- El último comando tiene que estar seguido de un ;.

La línea de comandos se trata por el shell actual (con duplicaciones posteriores si fuera necesario).

#### Primer ejemplo

Los dos comandos siguientes producen el mismo resultado, pero la versión con llaves es más rápida:

```
$ ( date ; ls ) > resultado
$ { date ; ls ; } > resultado
```

Las figuras 19, 20, 21 y 22 representan el mecanismo interno asociado a las llaves. El shell de trabajo guarda una copia de sus asociaciones **descriptor-archivo** actuales **(1)**, trata él mismo la redirección solicitada **(2)**, se duplica para la ejecución del comando externo **date** **(5)** y después, cuando este último ha terminado **(7)**, se duplica de nuevo para ejecutar **ls** **(8)**. Cuando los comandos han terminado, el shell de primer nivel retoma el control **(10)** y restaura su entorno **descriptor-archivo** **(11)**.

```
{ date ; ls ; } > resultado
```

1ª etapa: Guardar copia del descriptor 1

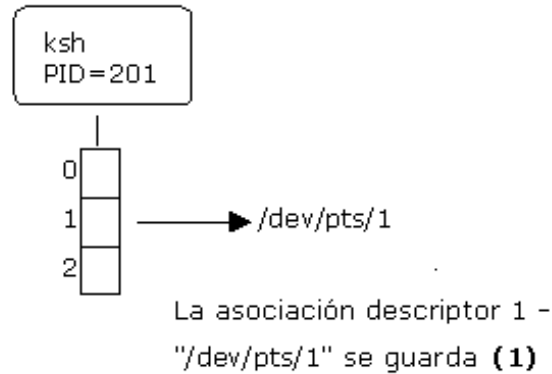


Figura 19: Primer ejemplo de agrupación con llaves - Primera etapa

2ª etapa: Ejecución del comando date

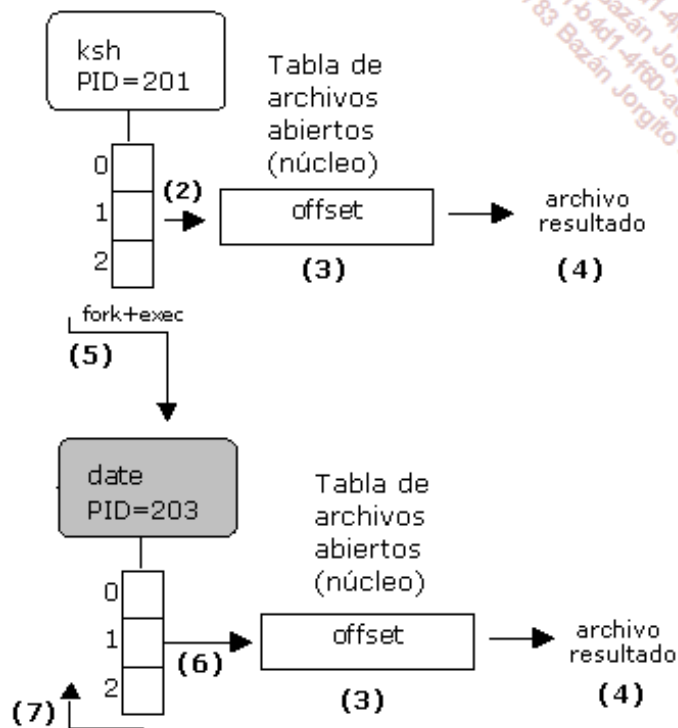


Figura 20: Primer ejemplo de agrupación con llaves - Segunda etapa



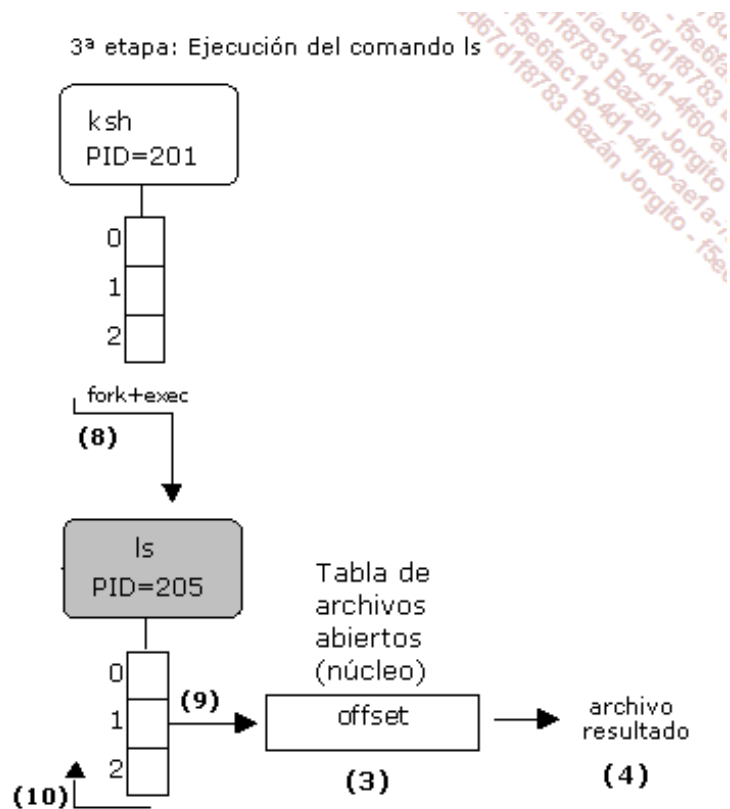


Figura 21: Primer ejemplo de agrupación con llaves - Tercera etapa

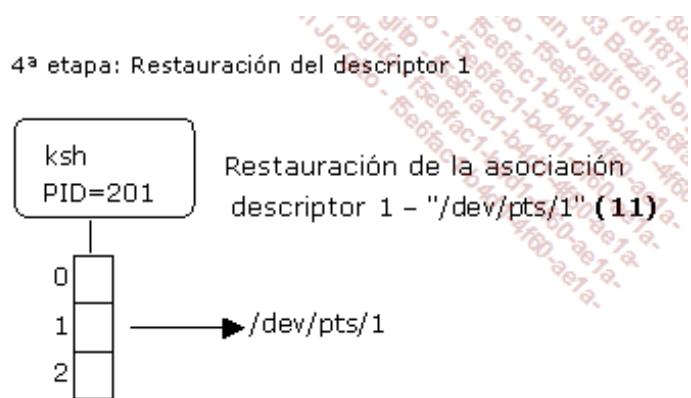


Figura 22: Primer ejemplo de agrupación con llaves - Cuarta etapa

Segundo ejemplo

En este caso, el entorno del shell de primer nivel se va a modificar, lo que no es necesariamente útil:

```

$ pwd
/home/cristina
$
$ { cd /tmp ; pwd ; ls ; } > listaarch
$
$ cat listaarch
/tmp
dcopNYSrKn
listatmp
$ pwd
/tmp
$

```

Las figuras 23 y 24 representan el mecanismo interno asociado. El shell de trabajo guarda una copia de sus asociaciones **descriptor-archivo** actuales y realiza él mismo la redirección solicitada. Ejecuta a continuación el comando interno **cd** (1) (su directorio actual cambia (2)), después el comando interno **pwd** (3) (que muestra por lo tanto **/tmp**) y finalmente se duplica (4) para la ejecución del comando externo **ls**, que hereda del directorio **/tmp** (5). Cuando **ls** ha terminado, el shell de trabajo retoma el control y restaura su entorno **descriptor-archivo**. En cambio, su directorio actual sigue siendo **/tmp**. El comando **pwd** lo confirma (6).

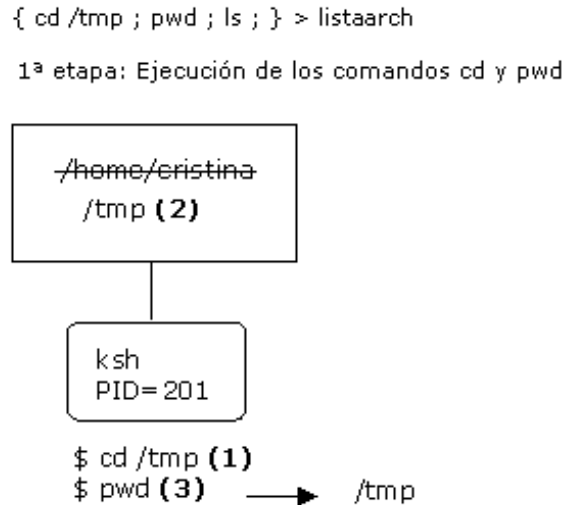


Figura 23: Segundo ejemplo de agrupación con llaves - Primera etapa

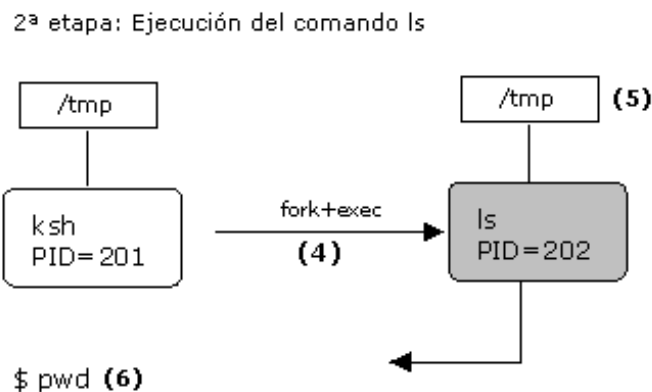


Figura 24: Segundo ejemplo de agrupación con llaves - Segunda etapa

### 3. Conclusión

Los paréntesis se utilizan más que las llaves por las dos razones siguientes:

- Su sintaxis es más sencilla de usar.
- Sea cual sea el conjunto de comandos, se está siempre seguro de encontrar el entorno de trabajo inicial.

La utilización de llaves se justificaría en el caso de una búsqueda de eficiencia.

## Procesos en segundo plano

Los conceptos de segundo plano y primer plano son gestionados por el shell.

Por defecto, los comandos se ejecutan en primer plano. En este modo, el shell padre "duerme" a la espera del final del comando. Retomará el control únicamente cuando el comando haya terminado.

El carácter `&` es un carácter especial del shell que permite ejecutar el comando en segundo plano. El shell ejecuta el comando y rescribe inmediatamente su prompt a la espera de un nuevo comando. Como el shell y el comando se ejecutan en **paralelo** y ambos están vinculados al mismo terminal, es aconsejable redirigir las salidas del comando.

### Ejemplo

*El shell muestra el PID del comando (8247), así como su índice ([1]) en la lista de tareas en segundo plano ejecutadas a partir de este shell:*

```
$ find / -size +2000 1>/tmp/resu 2 >/dev/null&  
[1] 8247  
$
```

# Ejercicios

## 1. Funcionalidades varias

### a. Ejercicio 1: comandos internos y externos

¿Son los comandos `umask` y `chmod` comandos internos?

### b. Ejercicio 2: generación de nombres de archivo

Sea la siguiente lista de archivos:

```
$ ls
bd.class.php   header.inc.php  install.txt     readme.txt
prueba        index.php      mail.class.php
```

1. Muestre los nombres de archivo que terminan en `.php`.
2. Muestre los nombres de archivo que tengan la letra `e` en segunda posición.
3. Muestre los nombres de archivo cuya primera letra esté comprendida entre `a` y `e`.
4. Muestre los nombres de archivo que no comienzan por una vocal.

### Expresiones complejas (ksh, bash)

5. Muestre los nombres de archivo que no terminan en `.php`.
6. Muestre los nombres de archivo que no terminan ni con `.txt` ni con `.php`.

### c. Ejercicio 3: separador de comandos

¿Cómo se escriben los dos comandos siguientes en la misma línea?

```
$ cd /tmp
$ ls -l
```

## 2. Redirecciones

### a. Ejercicio 1

Liste todos los procesos del sistema y redirija el resultado a un archivo.

### b. Ejercicio 2

Sea el comando `who -A`, que genera un mensaje de error:

```
$ who -A
who : opción inválida -- 'A'
```

1. Relance este comando y redirija los errores a un archivo.
2. Relance este comando y haga desaparecer los errores sin redirigir a un archivo en disco.

### c. Ejercicio 3

Ejecute los comandos siguientes:

```
$ touch fic_existe
$ chmod 600 fic_existe fic_noexiste
chmod: no se puede acceder a "noexiste": No existe el archivo o
el directorio
```

1. Redirija el resultado del comando **chmod** a un archivo, los errores a otro.
2. Redirija los resultados y los errores del comando a un mismo archivo.

### d. Ejercicio 4

¿Qué hace el comando siguiente?

```
$ ls > resu -l
```

### e. Ejercicio 5

Ejecute los comandos **date**, **who** y **ls** y guarde el resultado de los tres comandos en un archivo (una sola línea de comando).

### f. Ejercicio 6

Ejecute los comandos **date** y **who -A** y almacene el resultado de los dos comandos en un archivo **resu** (una sola línea de comando). Recuerde: el comando **who -A** genera un mensaje de error.

## 3. Tuberías de comunicación

### a. Ejercicio 1

Muestre la lista de procesos paginando el resultado.

### b. Ejercicio 2

Combinando los comandos **ps** y **grep**, muestre la lista de los procesos **httpd** que funcionan en el sistema.

### c. Ejercicio 3

Combinando los comandos **tail** y **head**, muestre la sexta línea del archivo **/etc/passwd**.

### d. Ejercicio 4

Cree los archivos siguientes:

```
$ touch f2 f1 fic1.txt FIC.c Fic.doc fIc.PDF fic
```

Cuente el número de archivos cuyo nombre contenga la palabra **fic**. La búsqueda no deberá discriminar entre mayúsculas y minúsculas.

# VARIABLES DE ENTORNO

Los temas abordados en este capítulo permitirán al usuario configurar su entorno de trabajo teniendo en cuenta el shell utilizado.

Se definen una serie de variables en el entorno del shell. Estas contienen la información necesaria para el funcionamiento del intérprete o de los comandos ejecutados por este.

## 1. Listado de variables

El comando **set** devuelve la lista de las variables definidas en el shell actual.

### Ejemplo

```
$ set
HOME=/home/cristina
LOGNAME=cristina
PATH=/usr/bin:/bin
PS1='$ '
PS2='> '
TERM=vt100
...
```

## 2. Mostrar el valor de una variable

El carácter especial **\$** del shell permite acceder al contenido de una variable.

### Ejemplo

```
$ echo $HOME
/home/cristina
$
```

## 3. Modificación del valor de una variable

El shell permite inicializar y modificar variables.

### Ejemplo

```
$ variable=valor
$ echo $variable
valor
$
```

Si el valor contiene caracteres especiales del shell (**\$**, **>**, espacio...), hay que impedir que el shell los interprete poniendo el valor entre comillas simples.

- Utilizar comillas simples es una de las tres maneras posibles de enmascarar caracteres en shell. Este aspecto se detallará más adelante.

### Ejemplo

El símbolo **">"** (redirección) tiene que enmascararse, el espacio (separador de palabras en la línea de comandos) también:

```
$ variable='palabra1 palabra2 =>'
$ echo $variable
```

```
palabra1 palabra2 =>
$
```

- No hay que poner espacios alrededor del símbolo `=`. El shell no comprendería que se trata de una asignación.

## 4. Variables principales

Las variables presentadas a continuación tienen un valor definido en el momento de conexión. Otras variables pueden ser definidas posteriormente.

### a. HOME

Esta variable contiene el valor del directorio de inicio del usuario. No debe ser modificada.

### b. PATH

La variable PATH contiene una lista de directorios que el shell explora cuando este debe invocar un comando externo.

- En ningún caso, un comando se buscará en el directorio actual si este no figura en la variable PATH.

### Ejemplos

```
$ echo $PATH
/usr/bin:/bin
$
```

El comando **date** se conoce:

```
$ date
Tue Jan 28 17:51:23 MET 2014
$
```

En efecto, se encuentra en el directorio **/usr/bin**:

```
$ find / -name date 2 > /dev/null
/usr/bin/date
$
```

El comando **ping** no se conoce:

```
$ ping localhost
ksh: ping: not found
$
```

El comando se encuentra dentro del directorio **/usr/sbin**, que no está incluido en la variable PATH:

```
$ find / -name ping 2> /dev/null
/usr/sbin/ping
$
```

El directorio actual no se explora si no se cita en `PATH`:

```
$ cd /usr/sbin
$ ping localhost
ksh: ping: not found
$
```

Modificar el contenido de la variable `PATH`:

```
$ PATH=$PATH:/usr/sbin
$ echo $PATH
/usr/bin:/bin:/usr/sbin
$
```

El comando `ping` ahora se conoce:

```
$ ping localhost
localhost is alive
$
```

## **Buscar un comando en el directorio actual**

Para que un comando se busque en el directorio actual, hay que añadir al final de la variable `PATH` la cadena `":"` o simplemente el carácter `:`.

### Ejemplo

```
PATH=/usr/bin:/usr/local/bin:/home/cristina/bin:.
```

Es equivalente a:

```
PATH=/usr/bin:/usr/local/bin:/home/cristina/bin:
```

## **c. PWD**

ksh	bash
-----	------

Esta variable contiene el valor del directorio actual. Cada vez que el usuario cambia de directorio, el shell se encarga de actualizarla. Esta variable puede utilizarse en `ksh` para mostrar el valor del directorio actual en el prompt.

## **d. PS1**

Esta variable contiene la cadena de caracteres que representan el prompt principal.

### Ejemplo

```
$ echo $PS1
$
$ PS1='Entre un comando => '
Entre un comando => date
Thu Jan 30 17:27:51 MET 2014
Entre un comando =>
```

En `ksh` y en `bash`, es posible configurar el prompt de tal forma que contenga permanentemente el valor del directorio actual.



## Mostrar el directorio actual en el prompt en ksh

Se usa la variable PWD.

### Ejemplo

A continuación, el prompt se compone de dos caracteres: el símbolo "\$" seguido de un espacio (ver figura 1):

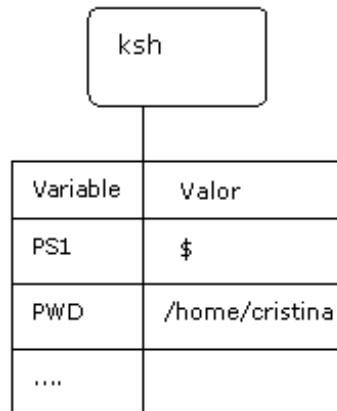


Figura 1: Inicialización de PS1 con el directorio actual (1)

```
$  
$ echo -$PS1-  
-$ -  
$
```

El directorio actual es **/home/cristina**:

```
$  
$ pwd  
/home/cristina  
$
```

Inicialización de PS1 con la cadena de caracteres '\$PWD\$'; es necesario impedir que el shell substituya \$PWD por su valor en el momento de la asignación; por lo tanto, hay que proteger la expresión con comillas (ver figura 2):

```
$  
$ PS1='$PWD$' $ _ '
```

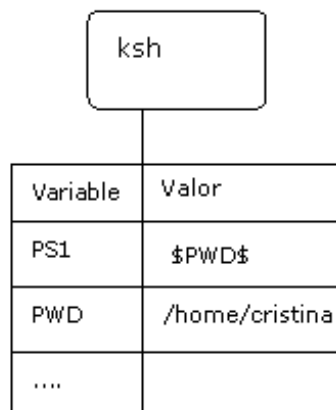


Figura 2: Inicialización de PS1 con el directorio actual (2)

El shell debe mostrar ahora su prompt. Va a buscar el valor de PS1 (\$PWD\$). La variable PWD se evalúa y se reemplaza por su valor (actualmente /home/cristina):

```
/home/cristina$
```

Cambio de directorio:

```
/home/cristina$ cd /tmp
```

El shell actualiza inmediatamente la variable PWD, que ahora vale "/tmp". Después tiene que mostrar su prompt. Vuelve a buscar el valor de PS1 (\$PWD\$) y lo evalúa. Por lo tanto, PS1 tiene por valor "/tmp" (ver figura 3):

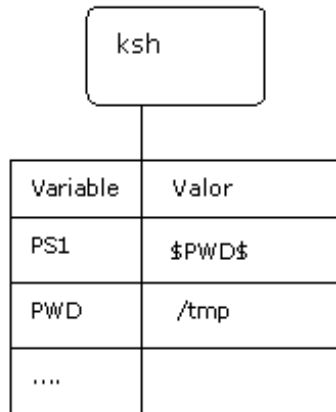


Figura 3: Inicialización de PS1 con el directorio actual (3)

```
/tmp $ pwd  
/tmp  
/tmp $
```

### **Mostrar el directorio actual en el prompt en bash**

Es posible emplear la misma técnica anterior. De todas formas, existen en bash secuencias de escape con un significado particular; resulta práctico utilizarlas para inicializar PS1. La tabla siguiente presenta las principales secuencias:

Secuencia de escape	Valor
\u	Nombre del usuario
\h	Nombre de la máquina
\w	Directorio actual
\W	Parte final del directorio actual

➤ Las secuencias de escape deben ir siempre entre comillas (simples o dobles).


### **Ejemplo**

Prompt de bash:

```
$
```

Modificar el prompt de tal manera que aparezca el nombre de la máquina seguido del directorio actual:

```
$ PS1='/ \h@\w$ '
rumba@~$ pwd
/home/cristina
rumba@~$ ls -l
drwxr-xr-x  2 cristina cristina   4096 dic  1 16:22 C
drwxr-xr-x  2 cristina cristina   4096 oct 23  2001 Desktop
rumba@~$ cd C
rumba@~/C$ cd /tmp
rumba@/tmp$
```

 El carácter ~ representa en bash y en ksh el directorio de inicio del usuario.

## e. PS2

Esta variable contiene la cadena de caracteres que representan el prompt secundario. Este aparece cuando los elementos de la sintaxis shell están incompletos.

### Primer ejemplo

Mientras el shell no encuentre las comillas de cierre, espera la continuación del comando:

```
$ echo 'Impresión del carácter *
> yo
> tengo que
> cerrar las comillas'          (Final del comando)
Impresión del carácter *      (Resultado de la ejecución)
yo
tengo que
cerrar las comillas
$
```

### Segundo ejemplo

Mientras el shell no encuentre de nuevo la etiqueta %, espera la continuación del mensaje:

```
$ mail cristina <<%
> Buenas
> ¿Cómo estás?
> %                          (Final del comando) $
```

## f. TMOUT

Esta variable contiene una espera expresada en segundos. Si alguna interacción con el teclado no ha tenido lugar durante este tiempo, el shell finaliza. Cuando su valor es 0, el contador está desactivado.

## g. TERM

Esta variable está en principio correctamente inicializada. Contiene el tipo de terminal del usuario. Los valores más comunes son: ansi, vt100, vt220, dtterm, xterm. Estos valores se envían a un archivo de configuración del sistema (base de datos terminfo).

### Ejemplo

```
$ echo $TERM
```

```
vt100
$ find / -name vt100 2> /dev/null (Búsqueda del archivo vt100)
/usr/share/terminfo/v/vt100
...
$
```

## h. LOGNAME

Esta variable contiene el nombre del usuario conectado.

```
$ echo $LOGNAME
crisrina
$
```

## i. Procesos y variables de entorno

No todas las variables de entorno son utilizadas por los mismos procesos. Se puede distinguir tres categorías de variables:

- Las que se utilizan únicamente por el shell (ej.: PS1, PS2).
- Las que se utilizan por múltiples comandos y si es preciso por el shell (ej.: PATH, TERM).
- Las que se utilizan por un comando determinado (ej.: EXINIT por **vi**).

## 5. Exportación de variables

Por defecto, las variables definidas a nivel del shell no se transmiten a los comandos ejecutados a partir de éste. Para que sean transmitidas, es necesario pedir al shell que las exporte.

### a. Listado de variables exportadas

El comando interno **env** muestra las variables declaradas del shell actual que se exportan.

#### Ejemplo

*Las variables PS1 y PS2 han sido redefinidas en el shell actual:*

```
$ set
HOME=/home/cristina
LOGNAME=cristina
PATH=/usr/bin:/bin
PS1=' comando> '
PS2=' continuación> '
TERM=vt100
...
```

*PS1 y PS2 no se exportan; por consiguiente, no aparecen en el resultado del comando **env**. Por lo tanto, retomarán su valor por defecto ('\$' y '>') en los shells descendientes:*

```
$ env
HOME=/home/cristina
LOGNAME=cristina
PATH=/usr/bin:/bin
TERM=vt100
...
```

### b. Variables que deben exportarse

Las variables utilizadas por otros procesos que no sean el shell deben exportarse obligatoriamente

para ser transmitidas.

Variable	¿Usada solo por el shell?	¿Exportación obligatoria?
PATH	no	sí
PS1	sí	no
PS2	sí	no
PWD	sí	sí
HOME	no	sí
LOGNAME	no	sí
TERM	no	sí
TMOUT	sí	no

### c. Exportar una variable

Una variable exportada a nivel de un shell será transmitida a todos los procesos descendientes, con independencia del nivel de descendencia.

#### Sintaxis

Exportar una variable ya definida:

```
$ export MIVARIABLE
```

Definir y exportar una variable:

```
$ MIVARIABLE=valor  
$ export MIVARIABLE
```

o

```
$ export MIVARIABLE=valor
```

ksh	bash
-----	------

#### Primer ejemplo

La variable utilizada en este ejemplo es EXINIT. Esta se consulta mediante el comando **vi** y contiene la lista de opciones que se deben configurar en el editor.

La figura 4 muestra cómo una variable no exportada no se transmite a los procesos descendientes del shell:

- La variable EXINIT se inicializa, pero no se exporta **(1)**.
- El comando **vi** se ejecuta. Este no recibe la variable EXINIT. Por lo tanto, los números de línea no se mostrarán **(2)**.

Conclusión: sin exportación, la definición de la variable EXINIT no sirve para nada.

## Definición de la variable EXINIT

(1) \$ EXINIT='set number'  
\$ vi prog.c

ksh  
PID=201

Variable	Valor	¿Se exporta?
HOME	/home/cristina	sí
EXINIT	set number	no
....		

(2) El proceso vi no recibe la variable EXINIT

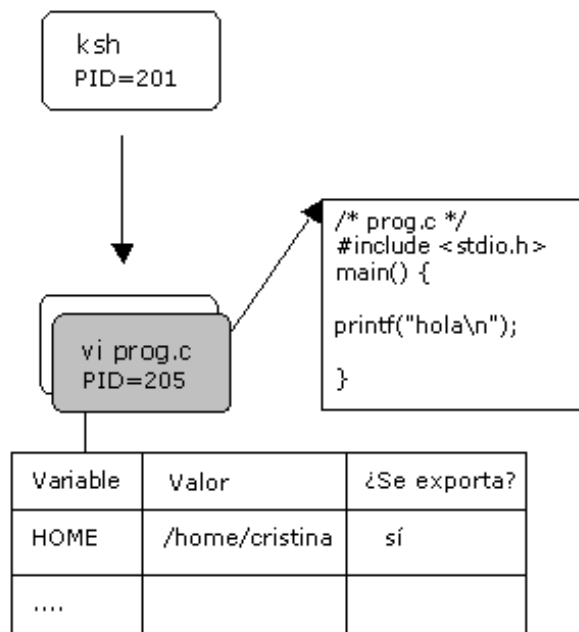


Figura 4: Definición de variable

### Segundo ejemplo

La figura 5 muestra cómo una variable exportada se transmite a los procesos descendientes del shell:

- La variable se define y se exporta **(1)**.
- Esta es, por lo tanto, recibida por el proceso **vi**, que se configura con las opciones requeridas **(2)**.

Definición y exportación de la variable EXINIT

```
$ EXINIT='set number'
$ export EXINIT
$ vi prog.c
```

ksh  
PID=201

(1)

Variable	Valor	¿Se exporta?
HOME	/home/cristina	sí
EXINIT	set number	sí
....		

El proceso vi recibe la variable EXINIT

ksh  
PID=201

```
1 /* prog.c */
2 #include <stdio.h>
3 main() {
4
5 printf("hola\n");
6
7 }
```

vi prog.c  
PID=205

Variable	Valor	¿Se exporta?
HOME	/home/cristina	sí
EXINIT	set number	sí
....		

(2)

Figura 5: Definición y exportación de una variable

# Las opciones del shell

ksh	bash
-----	------

El shell propone una serie de opciones que permiten la configuración un cierto número de funcionalidades.

## 1. Activar y desactivar una opción del shell

Las opciones `-o` y `+o` del comando interno `set` permiten activar y desactivar respectivamente una opción del shell.

### Sintaxis

```
set -o opción
set +o opción
```

## 2. Visualizar la lista de opciones

El comando `set` con la opción `-o` muestra la lista de las opciones del shell indicando si están actualmente activadas o no.

```
$ set -o
Current option settings
allexport      off
bgnice        on
emacs         on
errexit       off
gmacs         off
ignoreeof     off
interactive    on
keyword       off
markdirs      off
monitor       on
noexec        off
noclobber     off
noglob        off
nolog         off
nounset       off
privileged    off
restricted    off
trackall      off
verbose       off
vi            off
viraw         off
xtrace        off
notify        off
$
```

## 3. Opciones principales

### a. ignoreeof

Para salir de un shell, hay dos métodos:

- El comando `exit`.
- La secuencia de teclas `^d`.



Si la opción **ignoreeof** está activada, se vuelve imposible salir del shell pulsando `^d`.

La opción está desactivada por defecto.

### Ejemplo

```
$ set -o ignoreeof
$ ^d          (Se introduce ^d)
Use 'exit' to terminate this shell
```

## b. noclobber

Cuando una redirección se realiza hacia un archivo existente, este se sobrescribe sin advertencia previa (sujeto a los permisos). Para ser advertido de la existencia del archivo, hay que activar la opción **noclobber**.

### Ejemplo

La opción **noclobber** está desactivada:

```
$ set -o | grep noclobber
noclobber      off
$ echo Hola > resu
$ ls -l resu
-rw-r--r--  1 cristina cristina      8 ene 31 17:04 resu
$ cat resu
Hola
$
```

El archivo **resu** se sobrescribirá:

```
$ echo Adiós > resu
$ cat resu
Adiós
$
```

Activación de la opción **noclobber**:

```
$ set -o noclobber
```

Imposible sobrescribir **resu**:

```
$ echo Gracias noclobber > resu
ksh: resu: file already exists
$ cat resu
Adiós
$
```

Para forzar la sobrescritura, hay que usar la redirección **>|**:

```
$ echo Gracias noclobber >| resu
$ cat resu
Gracias noclobber
$
```

La opción **noclobber** está desactivada por defecto.

### c. emacs y vi

Estas opciones permiten configurar la llamada a comandos (ver Histórico de comandos, en este capítulo):

- En ksh, estas dos opciones están desactivadas por defecto.
- En bash, la opción **emacs** está activada por defecto.

### d. xtrace

Esta opción se usa en programación shell para debuguear los scripts (ver capítulo Las bases de la programación shell - Corrección de un script). Esta opción está desactivada por defecto.

# Los alias

ksh	bash
-----	------

El shell ofrece un comando interno **alias** que permite crear atajos a comandos. Un cierto número de alias existen por defecto.

## 1. Definir un alias

### Ejemplo

Creación de tres alias: **l**, **c**, y **rm** que serán respectivamente los equivalentes a **ls -l**, **clear** y **rm -i**:

```
$ alias l='ls -l'
$ l
total 30
-rw-r--r--  1 cristina  curso    11428 Ene 28 06:19 out
-rw-r--r--  1 cristina  curso      22 Ene 31 17:08 out2
-rwxr--r--  1 cristina  curso     18 Nov 15 20:08 primero
-rw-r--r--  1 cristina  curso     51 Ene 28 06:22 resu
$ alias c='clear'
$ alias rm='rm -i'
$ rm out
rm: remove out (y/n)? n
$
```

## 2. Visualizar la lista de alias

### a. Visualizar todos los alias

```
$ alias
autoload='typeset -fu'
c=clear
functions='typeset -f'
history='fc -l'
integer='typeset -i'
l='ls -l'
local=typeset
r='fc -e -'
rm='rm -i'
...
```

### b. Visualizar un alias en particular

```
$ alias l
l='ls -l'
$
```

## 3. Eliminar un alias

```
$ unalias l
$ l
ksh: l: not found
$
```

# Histórico de comandos

ksh	bash
-----	------

El shell almacena los comandos ejecutados en un archivo de texto localizado en el directorio de inicio del usuario. El nombre de este archivo difiere en función del shell utilizado.

Shell	Archivo histórico
ksh	.sh_history
bash	.bash_history

Para recuperar los comandos almacenados en este archivo, shell ofrece dos opciones: **emacs** y **vi**.

Estas dos opciones son mutuamente excluyentes: la activación de una desactiva la otra. En ksh, ambas están desactivadas por defecto. En bash, la opción **emacs** está activada por defecto.

*Configuración por defecto en ksh:*

```
$ set -o
Current option settings
...
emacs          off
...
vi             off
...
$
```

*Configuración por defecto en bash:*


```
$ set -o
Current option settings
...
emacs          on
...
vi             off
...
$
```

## 1. Configurar la recuperación de comandos en ksh

### a. Opción vi

El shell ofrece el uso de comandos idénticos a los que tiene el editor **vi** para recuperar y, si es preciso, modificar los comandos almacenados en el archivo **~/sh\_history**. Para ello es necesario activar la opción:

```
$ set -o vi
```

 El carácter **~** representa en bash y en ksh el directorio inicial del usuario.

A partir de este momento, hay que imaginarse que se está dentro del editor **vi**. Ciertas acciones se ejecutan en modo comando, otras en modo inserción. La tabla siguiente agrupa los comandos principales que permiten gestionar el histórico.


### Acciones en modo inserción

Este es el modo por defecto. El modo inserción permite introducir o modificar un comando.

Acción	Comando
Introducir un carácter	Carácter deseado
Ejecutar el comando	Intro
Pasar a modo comando	ESC
Quitar el significado especial del carácter siguiente	$\wedge v$
Quitar el significado especial de los caracteres "kill" ( $\wedge u$ ) y "erase" ( $\wedge h$ o $\wedge ?$ )	$\backslash$

### Acciones en modo comando


Acción	Comando
Ascender por el histórico	k
Descender por el histórico	j
Desplazarse hacia la derecha en el comando seleccionado	l
Desplazarse hacia la izquierda en el comando seleccionado	h
Borrar el carácter actual	x
Borrar el carácter anterior	parámetro "erase" de stty ( $\wedge h$ o $\wedge ?$ )
Borrar la línea actual	dd
Pasar a modo inserción (sin desplazamiento del cursor)	i
Pasar a modo inserción (desplazando el cursor al inicio de la línea)	I
Pasar a modo inserción (desplazando el cursor una posición a la derecha)	a
Pasar a modo inserción (desplazando el cursor al final de la línea)	A
Ejecutar el comando	Intro

  $\wedge h$  representa la tecla [Retrosceso] (backspace) y  $\wedge ?$  representa la tecla [Supr] (o [Del]).

### b. Opción emacs

El shell ofrece el uso de comandos idénticos a los del editor **emacs** para recuperar y, si es preciso, modificar los comandos almacenados en el archivo `~/.sh_history`. Para ello es necesario activar la opción:

```
$ set -o emacs
```

 El hecho de tener el editor **emacs** instalado o no en el sistema no afecta en ningún caso el funcionamiento de la opción **emacs**, ya que esta es gestionada por el shell.

La tabla siguiente agrupa los comandos principales que permiten gestionar el histórico.

Acción	Comando
--------	---------

Ascender por el histórico (previous)	<code>^p</code>
Descender por el histórico (next)	<code>^n</code>
Desplazarse hacia la derecha en el comando seleccionado (forward)	<code>^f</code>
Desplazarse hacia la izquierda en el comando seleccionado (backward)	<code>^b</code>
Borrar el carácter actual	<code>^d</code>
Borrar el carácter anterior	parámetro erase de stty ( <code>^h</code> o <code>^?</code> )
Borrar la línea actual	parámetro kill de stty ( <code>^u</code> )
Introducir un carácter	Carácter deseado
Ejecutar un comando	Intro
Mostrar la versión del shell	<code>^x^v</code>
Quitar el significado especial del carácter siguiente	<code>\</code>

### Ventajas de la opción **emacs**

Para un usuario poco familiarizado con los editores **vi** y **emacs**, todos estos comandos pueden parecer complicados.

En este caso, se recomienda usar el modo **emacs**, ya que, con una configuración suplementaria, las flechas del teclado pueden utilizarse para recorrer el histórico.

Representación de las flechas:

Símbolo representativo de las flechas en modo <b>emacs</b>	Flecha correspondiente
<code>__A</code>	[Flecha hacia arriba]
<code>__B</code>	[Flecha hacia abajo]
<code>__C</code>	[Flecha hacia la derecha]
<code>__D</code>	[Flecha hacia la izquierda]

El comando **alias** permite realizar la correspondencia entre las flechas del teclado y los comandos del modo **emacs**.

A continuación se muestra el modo de hacerlo:

Si la opción **emacs** no está previamente activada, hay que activarla:

```
$ set -o emacs
```

El siguiente ejemplo crea la correspondencia entre la flecha hacia arriba y el comando **emacs**, que asciende en el histórico: `^p`.

Algo importante que hay que tener en consideración:

A la derecha del símbolo de asignación, hay que insertar el código de la tecla `^p` manteniendo la tecla [Ctrl] apretada mientras se pulsa **p**. Dando por hecho que el modo **emacs** está activado, el hecho de pulsar `^p` no muestra `^p`, pero asciende por el histórico (ies normal!). Por lo tanto, es necesario enmascarar el significado especial de `^p` precediéndolo del carácter `\` (carácter de protección del modo **emacs**).

A continuación se muestra lo que tiene que aparecer por pantalla:

```
$ alias __A=^P
```

A continuación, los caracteres que será necesario introducir para obtener la línea anterior:

```
$ alias __A=\[Ctrl]p
```

Procediendo de la misma manera, se muestran los tres otros alias que hay que crear:

```
$ alias __B=^N
$ alias __C=^F
$ alias __D=^B
```

➤ Si estos alias se introducen en un archivo de configuración utilizando el editor **vi**, hay que usar **^v** y no **\** para quitar el significado especial del carácter siguiente.

Mostrar los alias:

Mostrar la lista de alias por pantalla envía los caracteres **^P**, **^N**, **^F**, **^B** al terminal y lo perturba. La figura 6 representa el efecto obtenido.

```

$ alias
__A=
__B=
__C=
__D=
[...]
```

Figura 6: Mostrar los alias de las flechas

Para evitar este fenómeno, hay que enviar la salida del comando **alias** hacia la entrada del comando **cat -v**.

### Ejemplo

La opción **-v** permite mostrar de manera visible los caracteres no imprimibles:

```
$ alias | cat -v
__A=^P
__B=^N
__C=^F
__D=^B
autoload='typeset -fu'
...
$
```

## 2. Configurar la recuperación de comandos en bash

En bash, la gestión del histórico es mucho más simple. Como en ksh, las opciones **emacs** y **vi** están disponibles, la opción **emacs** está activada por defecto. La correspondencia **comandos emacs** y **las flechas del teclado** es automática.

El usuario que prefiera la opción **vi** podrá activarla ejecutando el comando siguiente:

```
$ set -o vi
```

Los comandos se almacenan en el archivo `~/bash_history`.

### 3. Completar nombres de archivo

El completado es una funcionalidad ofrecida por ksh y bash. Apretando las teclas adecuadas, el shell puede completar automáticamente un nombre de archivo o proponer la lista de los archivos disponibles en un directorio determinado.

#### a. Completar en bash

##### Ejemplo

A continuación se muestra la lista de los archivos del directorio actual:

```
$ ls
f1 fic2.txt fic3.txt fic.txt prog.c
```

Tecleando la primera letra de un nombre de archivo seguida del carácter de tabulación ( `↵` ), el bash completa automáticamente la entrada con el nombre del archivo del directorio actual que comience por la letra "p", con la condición de que el directorio no contenga más de uno, que es el caso que nos ocupa:

```
$ ls p ↵      Completado de bash:      $ ls prog.c
```

Cuando hay múltiples posibilidades, el shell no completa:

```
$ ls fic ↵      Bash no hace nada: $ ls fic
```

Si el usuario aprieta dos veces al tabulador, el shell muestra las distintas posibilidades:

```
$ ls fic ↵ ↵
fic2.txt fic3.txt fic.txt
```

El usuario completa el nombre del archivo con un conjunto de caracteres discriminantes, seguidos de una tabulación:

```
$ ls fic2 ↵      Completado de bash:      $ ls fic2.txt
```

Cuando un archivo completado es un directorio, el shell añade una "/" al final de la cadena:

```
$ mkdir rep1
$ touch rep1/f1.c rep1/f2.c
$ cd r ↵      Completado de bash:      $ cd rep1/
```

Lo que permite, si es necesario, encadenar otra petición de completado:

```
$ ls r ↵      Completado de bash:      $ ls rep1/
```



```
$ ls rep1/f ↵ ↵
f1.c f2.c
```

## b. Completar en ksh

### Modo vi

Los ejemplos siguientes muestran las teclas que permiten gestionar el completado de los nombres de archivo cuando un usuario trabaja con ksh configurado en modo **vi** (set -o vi).

#### Ejemplo

La lista de los archivos del directorio actual es la siguiente:

```
$ ls
f1 fic2.txt fic3.txt fic.txt prog.c
```

Tecleando la primera letra de un nombre de archivo seguido de [Escape] \ , el ksh completa automáticamente la entrada con el nombre del archivo del directorio actual que comienza por la letra "p", con la condición de que el directorio no contenga más de uno, que es el caso que nos ocupa:

```
$ ls p[Escape]\ Completado de ksh: $ ls prog.c
```

Cuando hay múltiples posibilidades, apretar [Escape]= permite listarlas:

```
$ ls f[Escape]= Lista las distintas posibilidades
1) f1
2) fic2.txt
3) fic3.txt
4) fic.txt
```

### Modo emacs

Los ejemplos siguientes muestran las teclas que permiten gestionar el completado de los nombres de archivo cuando un usuario trabaja con ksh configurado en modo **emacs** (set -o emacs).

#### Ejemplo

La lista de los archivos del directorio actual es la siguiente:

```
$ ls
f1 fic2.txt fic3.txt fic.txt prog.c
```

Tecleando la primera letra de un nombre de archivo seguida de dos pulsaciones sobre [Escape], el ksh completa automáticamente la entrada con el nombre del archivo del directorio actual que comienza por la letra "p", con la condición de que el directorio no contenga más de uno, que es el caso que nos ocupa:

```
$ ls p[Escape][Escape] Completado de ksh: $ ls prog.c
```

Cuando hay múltiples posibilidades, la tecla [Escape]= permite listarlas:

```
$ ls f[Escape][Escape] Ninguna reacción por parte de ksh
$ ls f[Escape]= Lista las distintas posibilidades
1) f1
2) fic2.txt
3) fic3.txt
4) fic.txt
```

### c. Tabla resumen

	<b>bash</b>	<b>ksh modo emacs</b>	<b>ksh modo vi</b>
Completado simple de una "palabra"	↵ (tabulación)	[Escape] [Escape]	[Escape]\
Mostrar la lista de archivos que empiezan por "palabra"	↵ ↵	[Escape]=	[Escape]=

# Los archivos de entorno

## 1. Características de los archivos de entorno

Los archivos de entorno sirven para almacenar de manera permanente las definiciones vinculadas a la configuración del entorno de usuario.

Contienen comandos Unix y siempre se interpretan por un shell. Estos son los scripts shell de entorno.

Algunos scripts de entorno solo se ejecutan por el shell de conexión. Por tanto, es importante saber distinguir un shell de conexión de un shell normal.

### a. Shell de conexión

#### Entorno de texto

En un entorno de texto, el shell de conexión se ejecuta inmediatamente después de la identificación del usuario por el nombre de login y su contraseña. Este proceso es el ancestro común de todos los comandos que se ejecutarán durante la sesión. Todo shell ejecutado posteriormente no tendrá el estatus de shell de conexión.

#### Entorno gráfico

En un entorno gráfico, el shell de conexión se ejecuta entre la pantalla de conexión y la visualización del escritorio. Por lo tanto, este no es un shell interactivo. El escritorio y toda aplicación ejecutada posteriormente a partir de los iconos son procesos descendientes del shell de conexión. En la mayoría de los casos, un terminal abierto desde el entorno gráfico no se considera un shell de conexión.

Sin embargo, ciertas plataformas permiten ejecutar desde el escritorio o bien un shell de conexión (ej.: icono **Consola** en Solaris) o bien un shell ordinario (ej.: icono **Terminal/MyHost** de Solaris). Los archivos de entorno usados no serán los mismos en cada caso.

### b. Archivos de entorno leídos por el shell de conexión

El shell de conexión lee un script shell del sistema llamado **/etc/profile**. Este archivo se gestiona por el administrador del sistema y contiene los parámetros comunes a todos los usuarios.

A continuación busca en el directorio de inicio del usuario un script de entorno cuyo nombre depende del shell usado (ver tabla siguiente).

Shell	Nombre del archivo de entorno leído por el shell de conexión
sh (Bourne y POSIX) - ksh	.profile
bash	Uno de los tres archivos siguientes en el orden de búsqueda. -.bash_profile -.bash_login -.profile

#### ¿Qué poner en el archivo .profile (.bash\_profile)?

Este archivo contiene principalmente:

- La definición seguida de la exportación posible de una o varias variables: estas se transmiten a todos los procesos lanzados a partir del shell de conexión. Las variables

definidas pero no exportadas permanecerán locales a este último.

- La redefinición de parámetros del sistema tales como `umask`, características del terminal... que serán válidos durante toda la sesión.

### Ejemplo de archivo `.profile`

```
$ cat .profile
PATH=$PATH:/usr/local/bin:$HOME:$HOME/bin:.
PS1=' $PWD$ '
ENV=$HOME/.kshrc
EXINIT='set number autoindent'
export PATH PS1 ENV EXINIT
umask 077
$
```

- El valor del parámetro `umask` es un valor expresado en octal que repercute en los permisos asignados a los archivos y directorios en el momento de su creación.
- La variable `ENV` se trata, en este capítulo, en la sección Los archivos de entorno - Sesión utilizando un Korn Shell.

### Lo que no hay que añadir en el archivo `.profile` (`.bash_profile`)

No hay que añadir en el archivo las entidades que no se transmiten entre procesos, es decir:

- Las opciones del shell.
- Las definiciones de alias.
- En Korn Shell, un alias definido con la opción `-x` puede ser heredado por un script shell, pero no por un shell interactivo.

Las definiciones de los alias y de las opciones tienen que ser releídas por cada shell ejecutado, ya sea de conexión o no. Existe, en `ksh` y en `bash`, un archivo de entorno previsto para desempeñar este papel (en Bourne Shell, la cuestión no se plantea porque no hay ni opciones ni alias). El funcionamiento de `ksh` y de `bash` no es exactamente el mismo en este aspecto; cada uno de estos shells se analizará por separado.

### Provocar la relectura de un archivo de entorno

El archivo `.profile` se lee únicamente en el momento de la conexión por el mismo shell de conexión.

Si el usuario realiza una modificación en su archivo `.profile`, este debe ser releído por el shell. Para ello, hay dos métodos:

- Desconectarse y volverse a conectar, lo que tiene como efecto reiniciar un shell de conexión.
- Solicitar al shell actual (que no tiene por qué ser el de conexión) que relea el script de entorno. Para ello, hay que utilizar el comando interno `."` (el comando `."` se trata en el capítulo Las bases de la programación shell - Escritura y ejecución de un script en shell).

### Ejemplo

Contenido del archivo `.profile` que ha sido leído por el shell de conexión:

```
$ cat .profile
PATH=/usr/bin:/bin
```

```
$
```

Visualización del valor de las variables y del parámetro **umask** antes de la modificación del archivo **.profile**:

```
$ echo $PATH
/usr/bin:/bin
$ echo $PS1
$                               (Valor por defecto de PS1)
$ echo $EXINIT
$                               (Vacío por defecto)
$ echo $ENV
$                               (Vacío por defecto)
$ umask
022                             (Valor inicializado anteriormente en .profile)
```

Modificación del archivo **.profile**:

```
$ vi .profile
PATH=$PATH:/usr/local/bin:$HOME:$HOME/bin:.
PS1=' $PWD$'
ENV=$HOME/.kshrc
EXINIT='set number autoindent'
export PATH PS1 ENV EXINIT
umask 077
$
```

Pedir al shell actual que relea el archivo **.profile**:

```
$ . $HOME/.profile
```

Visualización del valor de las variables y del parámetro **umask**:

```
/home/cristina$ echo $PATH
/usr/bin:/bin:/usr/local/bin:/home/cristina:/home/cristina/bin:.
/home/cristina$ echo $PS1
$PWD$
/home/cristina$ echo $EXINIT
set number autoindent
/home/cristina$ echo $ENV
/home/cristina/.kshrc
/home/cristina$ umask
077
$
```

## 2. Sesión utilizando un Bourne Shell

La figura 7 representa el transcurso de una sesión de trabajo utilizando el Bourne Shell:

- Ejecución de un Bourne Shell de conexión **(1)**.
- Lectura del script en shell del sistema **/etc/profile**, seguidamente del script shell del usuario **\$HOME/.profile** si está presente: el shell de conexión va guardando la definición de las variables (exportadas o no) y de los parámetros del sistema **(2)**.
- Todos los descendientes del shell de conexión recibirán el valor de los parámetros del sistema y de las variables exportadas **(3)**.

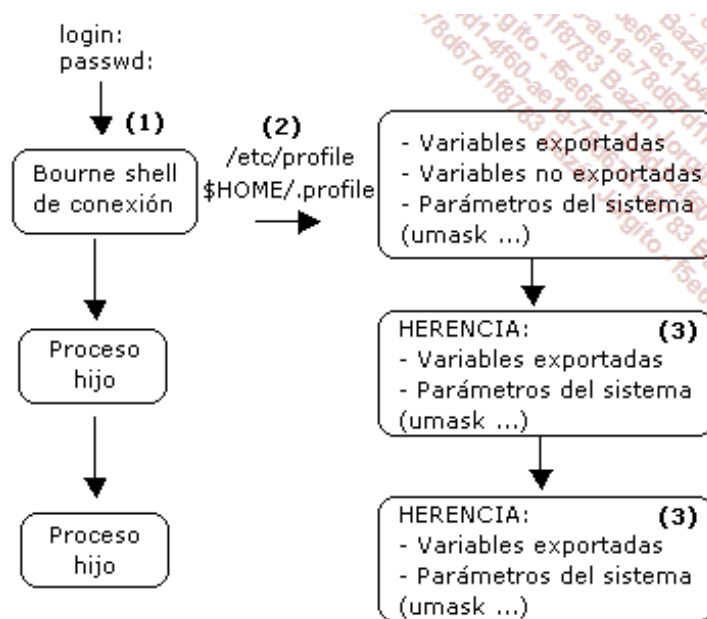


Figura 7: Sesión utilizando un Bourne Shell

### 3. Sesión utilizando un Korn Shell

Las definiciones de variables y parámetros de usuario deben realizarse en el archivo `~/profile`, las opciones y alias serán almacenados en otro script cuyo nombre se deja a elección del usuario (por convenio `~/kshrc`).

La figura 8 representa el transcurso de una sesión de trabajo usando Korn Shell.

#### Comportamiento de un Korn Shell de conexión

- Se ejecuta después de la identificación **(1)**.
- Lee el script de sistema `/etc/profile` **(2)**, seguidamente el script de usuario `~/profile` si está presente: el shell de conexión guarda la definición de las variables (exportadas o no) y de los parámetros del sistema.
- Si la variable ENV está definida **(3)**, ksh considera que su valor representa el nombre de otro script shell que debe leer. En general, se le da a este archivo el nombre `.kshrc`. Este último está cocebido para tener las definiciones de las opciones y alias **(4)**.

➤ La variable ENV no está definida por defecto.

El ksh muestra seguidamente su prompt (salvo en entorno gráfico, donde el escritorio se ejecuta rápidamente). Las variables de entorno, parámetros, opciones y alias se aplican a nivel del shell de conexión **(5)**.

#### Comportamiento de un Korn Shell ordinario (no conexión)

Un ksh lanzado posteriormente **(6)**:

- Recibirá automáticamente el valor de los parámetros de sistema y de las variables exportadas **(7)**.
- Si la variable ENV se ha recibido (para ello, es necesario que haya sido exportada a nivel del archivo `.profile`), el ksh leerá el archivo cuyo nombre está contenido en ella, en este caso `.kshrc` **(8)**.

Ksh muestra seguidamente su prompt. Las variables de entorno exportadas, parámetros, opciones y alias se aplican a nivel de este nuevo shell **(9)**.

➤ Un shell ordinario (de no conexión) no lee nunca el archivo **.profile**.

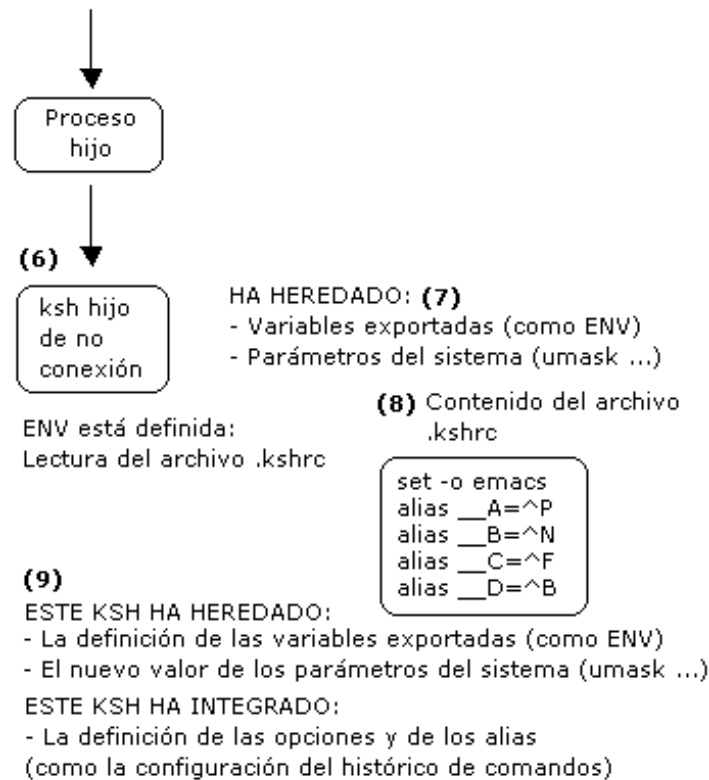
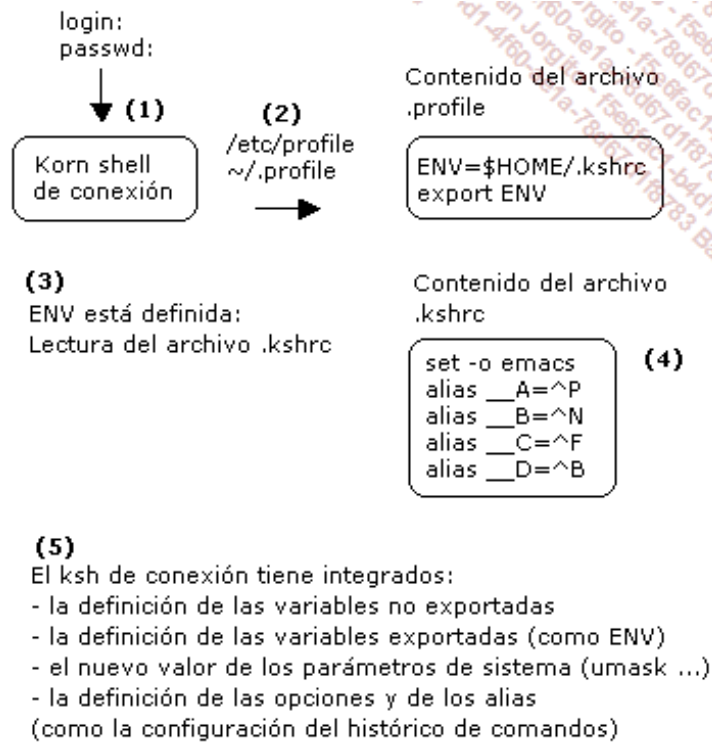


Figura 8: Sesión utilizando un Korn Shell

## 4. Sesión utilizando un Bourne Again Shell

Las definiciones de variables y de parámetros de usuario tienen que realizarse en el archivo `~/ .bash_profile`, las opciones y los alias estarán en otro script que el usuario llamará `~/ .bashrc`.

- El bash de conexión lee automáticamente el archivo `.bash_profile`, pero no el archivo `.bashrc`.
- Un bash ordinario (no de conexión) e interactivo lee automáticamente el archivo `~/ .bashrc`.
- Un bash ordinario y no interactivo (bash que va a ejecutar un script shell) lee el archivo cuyo nombre se cita en la variable `BASH_ENV` (que tiene que estar definida en el archivo `.bash_profile`). Esto permite inicializar las variables de entorno únicamente para el contexto de ejecución del script shell. Este caso no se tendrá en cuenta en el ejemplo siguiente.

La figura 9 representa la manera como el entorno tiene que configurarse para que las variables, parámetros, opciones y alias se tengan en cuenta en todos los bash interactivos.

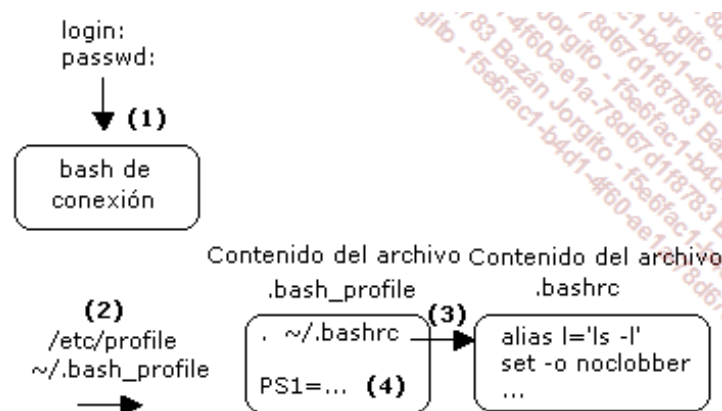
### Comportamiento del bash de conexión

- Se ejecuta justo después de la fase de identificación **(1)**.
- Lee el script de sistema `/etc/profile`, seguidamente abre el script de usuario `~/ .bash_profile` si existe **(2)**.
- El comando `. ~/ .bashrc` **(3)** se ha añadido para solicitar al bash de conexión la lectura del archivo `.bashrc` a fin de integrar la definición de los alias y de las opciones.
- Una vez el archivo `.bashrc` ha sido leído, el bash de conexión lee la continuación del archivo `.bash_profile` **(4)**.

El bash muestra seguidamente su prompt. Las variables de entorno, parámetros, opciones y alias han sido actualizados a nivel del bash de conexión **(5)**.

### Comportamiento de un bash interactivo ordinario

Un bash ejecutado con posterioridad recibirá automáticamente el valor de los parámetros del sistema y de las variables exportadas **(6)** y leerá automáticamente el archivo `~/ .bashrc` si este existe **(7)**. El bash muestra a continuación su prompt. Las variables de entorno, parámetros, opciones y alias han sido actualizados a nivel de este nuevo shell **(6, 8)**.

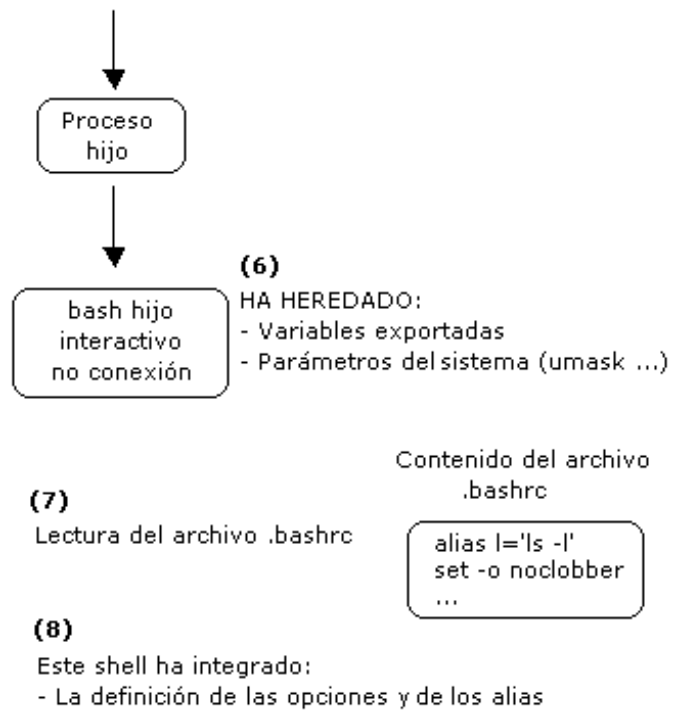


**(5)**

El bash de conexión ha integrado:

- la definición de las variables no exportadas
- la definición de las variables exportadas
- el nuevo valor de los parámetros del sistema (umask ...)
- la definición de las opciones y los alias





*Figura 9: Sesión utilizando un bash*

# Ejercicios

Los archivos proporcionados para los ejercicios están disponibles en la carpeta dedicada al capítulo en el directorio **Ejercicios/archivos**.

## 1. Variables de entorno

### a. Ejercicio 1

1. Haga que se muestre la lista de todas las variables de entorno.
2. Haga que se muestre la lista de las variables de entorno exportadas.
3. Haga que se muestre el valor de las variables de entorno **PATH** y **HOME**.

### b. Ejercicio 2

Cree un directorio **bin** en el directorio de inicio. En **bin**, cree o recupere el programa corto siguiente (script shell que muestra una frase en la pantalla):

```
$ pwd
/home/cristina/bin
$ vi micomando
echo "Ejecución de micomando"
$ chmod u+x micomando
```

1. Vuelva al directorio de inicio.
2. Modifique la variable **PATH** de forma que este comando funcione:

```
$ micomando
Ejecución de micomando
```

3. Haga que esta configuración sea permanente.

## 2. Alias de comando

### a. Ejercicio 1

1. Haga que se muestren los alias del shell actual (ksh, bash).
2. Cree un alias **p** que corresponda al comando **ps -ef | more**.
3. Pruebe el alias.
4. Destruya el alias.
5. Haga que este alias sea permanente.

### b. Ejercicio 2

Mientras se encontraba definiendo un alias, el usuario desafortunadamente ha tocado la tecla [Entrar] antes de haber podido cerrar las comillas. ¿A qué corresponde el carácter > y cómo salir de esta situación?

```
$ alias 'l=ls -l
Entrar
>
```



# Las variables de usuario

Este capítulo presenta las funcionalidades que componen las bases de la programación shell.

El shell permite definir o redefinir variables que condicionan el entorno de trabajo del usuario. También es posible definir otras variables, llamadas **variables de usuario**, que permitirán almacenar información que será necesaria durante la ejecución de un script.

## 1. Poner nombre a una variable

A continuación se muestran las reglas que hay que seguir para dar un nombre a las variables:

- El primer carácter pertenece al conjunto [a-zA-Z\_].
- Los caracteres siguientes pertenecen al conjunto [a-zA-Z0-9\_].

## 2. Definir una variable

Una variable está definida desde el momento en que ha sido inicializada. El contenido de una variable se considera por el shell como un conjunto de caracteres.

### a. Asignar un valor a una variable

#### *Ejemplo*

```
$ var1=palabra1
$ echo $var1
palabra1
$
```

➤ No hay que poner espacios alrededor del símbolo de la asignación: en el ejemplo siguiente, el shell interpreta **var1** como el comando que se ha de ejecutar, **=** y **palabra1** como los dos argumentos del comando **var1**. Dicho de otra manera, no interpreta el símbolo **=** como símbolo de asignación.

```
$ var1 = palabra1
ksh: var1: not found
$
```

### b. Asignar un valor con al menos un espacio

El carácter de espacio tiene que estar protegido, ya que es un carácter especial del shell (separador de palabras en la línea de comandos).

#### *Ejemplo*

```
$ var2='palabra1 palabra2 palabra3' #CORRECTO
$ echo $var2
palabra1 palabra2 palabra3
$ var2=palabra1 palabra2 palabra3 #INCORRECTO
ksh: palabra2: not found
$
```

### c. Variable indefinida

Una variable que nunca se ha inicializado está vacía.

#### Ejemplo

```
$ echo $vacia
$
```

### d. Borrar la definición de una variable

El comando interno **unset** permite borrar la definición de una variable.

#### Ejemplo

Definición de una variable **var**:

```
$ var=12
$ echo $var
12
```

Esta variable aparecerá en la lista de variables definidas a nivel de shell:

```
$ set | grep var
var=12
```

Se borra la definición de la variable:

```
$ unset var
```

La variable está indefinida:

```
$ echo $var

$ set | grep var
$
```

### e. Aislar el nombre de una variable

Hay que poner especial atención en la concatenación del contenido de una variable con una cadena de caracteres para que el shell interprete correctamente el nombre de la variable.

#### Ejemplo

Para el shell, el carácter **\_** forma parte del nombre de la primera variable:

```
$ arch=resu
$ fechadia=20140117
$ nuevoarch=$arch_ $fechadia
$ echo $nuevoarch
20140117
```

Para el shell, la primera variable se llama **arch\_** (iya que el carácter de guion bajo está permitido en los nombre de las variables). Por tanto, es sustituido por su valor (ique está vacío) y finalmente concatenado con el contenido de la variable **fechadia**.

Para especificar al shell cuáles son los caracteres que forman parte del nombre de una variable, hay que poner el nombre de este último entre **{ }**.

#### Ejemplo

```
$ arch=resu
$ fechadia=20140117
$ nuevoarch=${arch}_${fechadia}
$ echo $nuevoarch
resu_20140117
```

## f. Variables numéricas

### Declaración explícita de un nombre completo

```
ksh  bash
```

El comando interno **typeset** permite declarar explícitamente una variable como un entero. Esta declaración es opcional, pero permite tener controlado el valor almacenado y hace que los cálculos sean más rápidos.

#### Ejemplo

*Declaración, inicialización y visualización:*

```
$ typeset -i nb=1
$ echo $nb
1
```

*Control del valor asignado en ksh:*

```
$ nb=a
ksh: a: bad number
```

### Declaración explícita de un número flotante

```
ksh93
```

Solo ksh93 implementa la aritmética de números flotantes.

Declaración de un número flotante con una precisión de 3:

```
$ typeset -F3 iva=0.21
```

Esta funcionalidad se detalla en la sección dedicada a la aritmética de flotantes, en este capítulo.

## g. Variables complejas

```
ksh93
```

El ksh93 ofrece la posibilidad de crear variables complejas.

#### Ejemplo

La variable `user` tiene una propiedad **nombre** que contiene el valor `crisina` y una propiedad **uid** que contiene el valor `203`.

```
$ user=( nombre=crisina uid=203 )
```

Mostrar la variable `user`:

---

```
$ echo $user
Nombre : (
        nombre=cristina
        uid=203
)
```

Mostrar una propiedad en concreto:

```
$ echo ${user.nombre}
cristina
$ echo ${user.uid}
203
```

### 3. Sustitución de variables

El shell ofrece la posibilidad de atribuir un valor por defecto a las variables no inicializadas o, por el contrario, a las inicializadas.

#### Expresión `${variable:-valor}`

- Si la variable no está vacía, la expresión se sustituye por **`$variable`**.
- Si la variable está vacía, la expresión se sustituye por **`valor`**.

#### Ejemplo

```
$ arch=/tmp/cristina.log
$ echo "El archivo tratado será: ${arch:-/tmp/default.log}"
El archivo tratado será: /tmp/cristina.log
$ unset arch
$ echo "El archivo tratado será: ${arch:-/tmp/default.log}"
El archivo tratado será: /tmp/default.log
$ echo $arch
$
```

#### Expresión `${variable:=valor}`

- Si la variable no está vacía, la expresión se sustituye por **`$variable`**.
- Si la variable está vacía, a **`variable`** se le asigna **`valor`** y la expresión se sustituye por **`valor`**.

#### Ejemplo

```
$ arch=/tmp/cristina.log
$ echo "El archivo tratado será: ${arch:=/tmp/default.log}"
El archivo tratado será: /tmp/cristina.log
$ echo $arch
/tmp/cristina.log
$ unset arch
$ echo "El archivo tratado será: ${arch:=/tmp/default.log}"
El archivo tratado será: /tmp/default.log
$ echo $arch
/tmp/default.log
$
```

#### Expresión `${variable:+valor}`


- Si la variable no está vacía, la expresión se sustituye por **`valor`**.

- Si la variable está vacía, la expresión se sustituye por **\$variable**, luego estará vacía.

### Ejemplo

```
$ a=1
$ echo "Expresión: ${a:+99}"
Expresión: 99
$ unset a
$ echo "Expresión: ${a:+99}"
Expresión:
$
```

### **\${variable:?mensaje}**

- Si la variable no está vacía, la expresión se sustituye por **\$variable**.
  - Si la variable está vacía, el shell muestra el nombre de la variable seguido de la cadena de caracteres **mensaje**.
-  Si la variable está vacía y este comando está en un script de shell, este muestra el mensaje y finaliza inmediatamente.

### Ejemplo

La variable **var** está vacía:

```
$ echo $var
$ echo ${var:?no definida}
bash: var: no definida
```

Mensaje por defecto:

```
$ echo ${var:?}
bash: var: parameter null or not set
```

Definición de la variable **var**:

```
$ var=definida
$ echo ${var:?no definida}
definida
$
```



## Sustitución de comandos

Los caracteres de sustitución permiten reemplazar un comando por el resultado de su ejecución. Este mecanismo se utiliza para insertar en la línea de comandos Unix el resultado de otro comando.

### Sintaxis con las comillas invertidas (acento abierto)

```
comando argumentol `comando` ... argumenton
```

### Sintaxis equivalente

ksh	bash
-----	------

```
comando argumentol $(comando) ... argumenton
```

#### Ejemplos

Los comandos **uname -n** y **logname** son reemplazados por su resultado antes de la ejecución del comando **echo**:

```
$ echo Ud. está conectado actualmente a la máquina `uname -n` y
ud. es `logname`
Ud. está conectado actualmente a la máquina rumba y ud. es cristina
```

*Sintaxis específica de bash y ksh:*

*Inicialización de una variable **miuid** con el uid del usuario **cristina**:*

```
$ echo Ud. está conectado actualmente a la máquina $(uname -n) y
ud. es $(logname)
Ud. está conectado actualmente a la máquina rumba y
ud. es cristina
$
```


```
$ grep cristina /etc/passwd
cristina:x:2025:2000::/home/cristina:/bin/bash
$
$ grep cristina /etc/passwd | cut -d: -f3
2025
$
$ miuid=$(grep cristina /etc/passwd | cut -d: -f3)
$
$ echo $miuid
2025
$
```

# Caracteres de protección

Los caracteres de protección sirven para hacer perder el significado a los caracteres especiales del shell. Existen tres juegos de caracteres, cada uno con funcionalidades propias.

## 1. Las comillas simples

Las comillas simples (o apóstrofes) eliminan el significado a todos los caracteres especiales del shell. Las comillas deben ser un número par en la línea de comandos.

 Las comillas simples no se protegen a sí mismas.

### Ejemplos

La variable **\$HOME** se sustituye por su valor:

```
$ echo $HOME
/home/cristina
```

El carácter **\$** pierde su significado especial:

```
$ echo '$HOME'
$HOME
```

El carácter **\*** se sustituye por los nombres de archivo del directorio actual:

```
$ echo *
f1 f2 f3
```

El carácter **\*** pierde su significado especial:

```
$ echo '*'
*
```

El shell espera encontrar un nombre de archivo tras una redirección:

```
$ echo >
bash: syntax error near unexpected token `>'
```

El carácter **>** pierde su significado especial:

```
$ echo '>'
>
```

El shell ejecuta el comando **logname** y lo reemplaza por su resultado:

```
$ echo Hola $(logname)
Hola cristina
```

La secuencia de caracteres **\$( )** pierde su significado especial:

```
$ echo 'Hola $(logname)'
```

```
Hola $(logname)
$
```

*Protección de múltiples caracteres especiales:*

```
$ echo '* ? > < >> << | $HOME $(logname) &'
* ? >< >> << | $HOME $(logname) &
$
```

*Las comillas no se protegen a sí mismas. Para el shell, el comando no está finalizado. Mostrará el prompt secundario (PS2) mientras las comillas sigan siendo un número impar:*

```
$ echo 'las comillas ' no se protegen'
>
```

## 2. El carácter \

La barra invertida elimina el significado especial del carácter que le sigue.

### Ejemplos

*El asterisco se sustituye por los nombres de archivo del directorio actual; \$HOME se sustituye por su valor:*

```
$ echo A continuación un * y una variable $HOME.
A continuación un f1 f2 f3 y una variable /home/cristina.
$
```

*El asterisco y el dólar se vuelven caracteres normales:*

```
$ echo A continuación un \* y una variable \$HOME.
A continuación un * y una variable $HOME.
```

*La barra invertida se protege a sí misma:*

```
$ echo \\
\
$
```

*La barra invertida elimina el significado especial a las comillas:*

```
$ echo La barra invertida protege las \' (comillas)
La barra invertida protege la ' (comilla)
$
```

*La primera barra invertida protege a la segunda; la tercera barra invertida protege al símbolo de dólar:*

```
$ echo \\$HOME
\$HOME
$
```


## 3. Las comillas dobles

Las comillas dobles eliminan el significado de todos los caracteres especiales del shell, excepto \$, `` y \$(), \ y a sí mismas.

### Ejemplo

Los caracteres `>`, `|` y `'` se protegen con el uso de comillas dobles. Los `$`, `$( )` y `\` conservan su papel. Las penúltimas comillas dobles se protegen con la barra invertida:

```
$ echo "> y | se protegen, $HOME se sustituye, $(logname)
se ejecuta, la barra invertida protege al carácter siguiente,
lo que permite mostrar una \"." > y | se protegen, /home/cristina se sustituye, cristina
se ejecuta, la barra invertida protege el carácter siguiente,
lo que permite mostrar una ".
$
```

 En la práctica, es frecuente poner los argumentos del comando echo entre comillas dobles.

## Recapitulación

Esta tabla agrupa los caracteres especiales del shell vistos anteriormente:

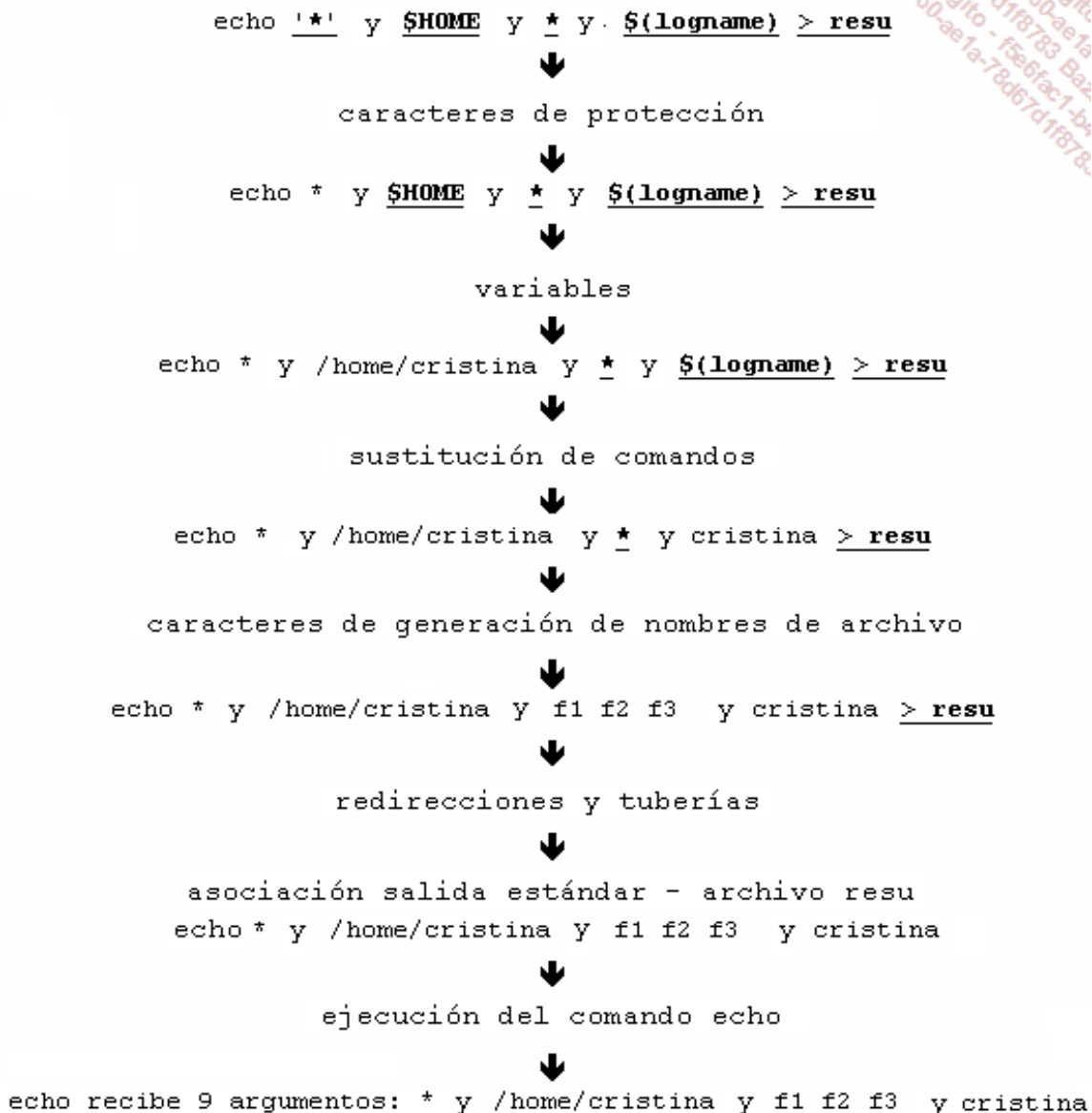
Caracteres	Significado
espacio - tabulación - salto de línea	Separadores de palabras en la línea de comandos
&	Segundo plano
<< < > >>	Tubería y redirecciones
() y {}	Agrupación de comandos
;	Separador de comandos
* ? [ ] ?() +() *() !() @()	Caracteres de generación de nombres de archivo
\$ y \${ }	Valor de una variable
`` \$()	Sustitución de comandos
' ' " " \	Caracteres de protección

# Interpretación de una línea de comandos

Los caracteres especiales del shell se interpretan en un orden determinado:

Secuencia de interpretación	Comando interno	Comando externo
1. Aislamiento de las palabras separadas por los caracteres espacio, tabulación y salto de línea 2. Tratamiento de los caracteres de protección ('', "", \) 3. Sustitución de las variables (\$) 4. Sustitución de los comandos (` ` \$()) 5. Sustitución de los caracteres de generación de nombres de archivo (*, ?, [] etc.)	Realizado por el shell actual	
6. Tratamiento de las tuberías y redirecciones 7. Ejecución del comando	shell actual	shell hijo

## Ejemplo



# Escritura y ejecución de un script en shell

## 1. Definición

Un script en shell es un archivo de texto que contiene los comandos Unix internos o externos, así como palabras clave del shell.

No hay ningún convenio establecido para el nombre de scripts de shell. El nombre de un archivo script de shell puede tener extensión, pero no es obligatorio. Sin embargo, generalmente se usa la extensión ".sh" (incluso si el script no se interpreta con el ejecutable llamado "sh").

### Ejemplo

A continuación se muestra el script **primero.sh**:

```
$ nl primero.sh
  1 pwd
  2 cd /tmp
  3 pwd
  4 ls
$
```

Ejecución del script:

```
$ ksh primero.sh
/home/cristina
/tmp
f1 f2 f3
$
```

Los comandos se ejecutan secuencialmente.

## 2. Ejecución de un script por un shell hijo

En la mayoría de los casos, los scripts tienen que ejecutarse a través de un shell hijo como intermediario. Esto tiene como ventaja la no modificación del entorno del shell actual. Para ejecutar un script en shell, existen tres métodos que producen un resultado equivalente.

### Primer método

```
$ ksh primero.sh
```

Este ha sido el método usado anteriormente. Se invoca al comando **ksh** pidiéndole la interpretación del script **primero.sh**.

En este caso, el permiso de lectura sobre el archivo **primero.sh** es suficiente:

```
$ ls -l primero.sh
-rw-r--r--  1 cristina curso      19 nov 15 19:16 primero.sh
```

### Segundo método

```
$ ksh < primero.sh
```

El ksh es un programa que lee su entrada estándar; por tanto, es posible conectarle el archivo **primero.sh**. En este caso, el permiso de lectura sigue siendo suficiente para su correcta

ejecución. Esta sintaxis es poco utilizada.

### **Tercer método**

```
$ chmod u+x primero.sh
$ ls -l primero.sh
-rwxr--r-- 1 cristina curso 19 nov 15 19:16 primero.sh
$ primero.sh
```

Este método es el más usado. En este caso, el script se considera un comando y, por consiguiente, como todo comando externo, es necesario poseer los permisos de ejecución sobre el archivo.

Por defecto, el script se interpretará por un shell hijo idéntico al shell actual (si el script se ejecuta desde un Korn shell, se interpretará por un Korn shell; si se ejecuta desde un Bourne Shell, se interpretará por un Bourne Shell).

### **Elección del intérprete**


En la primera línea del archivo, la directiva **#!** permite imponer el intérprete del script. La ruta de este tendrá que expresarse como ruta absoluta.

#### Ejemplo

*En este caso, el script se interpretará por el proceso /usr/bin/ksh.*

```
$ nl primero.sh
 1 #! /usr/bin/ksh

 2 pwd
 3 cd /tmp
 4 pwd
 5 ls
$
```

 Los caracteres **#!** se escriben respectivamente en las columnas 1 y 2. El espacio entre la **!** y la ruta absoluta del intérprete es opcional. Los shells más usados se encuentran en /usr/bin o /bin.

A continuación se muestran tres casos de error que se producen frecuentemente en la ejecución de un script:

### **Primer mensaje clásico de error**

```
$ primero.sh
ksh: primero.sh: not found
```

El **ksh** no encuentra el comando **primero.sh** porque el directorio donde se halla el script no esté referenciado en la variable **PATH**.

Si no se desea modificar la variable **PATH**, el script se puede ejecutar expresando su emplazamiento de manera explícita (con ruta absoluta o relativa).

#### Ejemplo

*El lugar del script se expresa con ruta absoluta:*

```
$ /home/cristina/primero.sh
$
```

*El lugar del script se expresa con ruta relativa:*



```
$ pwd
/home/cristina
$ ls
primero.sh
$ ./primero.sh
$
```

Otra alternativa es la de modificar la definición de la variable PATH en el archivo **.profile** (ver la sección Escritura y ejecución de un script en shell - Ejecución de un script por el shell actual, en este capítulo).

## Segundo mensaje clásico de error

```
$ primero.sh
ksh: primero.sh: cannot execute - Permission denied
$ ls -l primero.sh
-rw-r----- 1 cristina curso 3 nov 16 17:33 primero.sh
```

El usuario ha olvidado asignar el permiso de ejecución.

## Tercer mensaje clásico de error

```
$ vi primero.sh
#!/bin/ksh
...
$
$ primero.sh
ksh: primero.sh: No such file or directory
```

La ruta del intérprete es incorrecta. En este caso, falta la / delante de **bin**.

## Mecanismo interno común en las tres sintaxis

Examinemos más de cerca la ejecución de este script:

```
$ pwd
/home/cristina

$ primero.sh
/home/cristina
/tmp
f1 f2 f3
$ pwd
/home/cristina
```

Cuando la ejecución del script ha terminado, el valor del directorio actual no ha cambiado. En efecto, un script en shell, al ser un comando externo, se ejecuta a través de un shell **hijo**. Las figuras 1, 2 y 3 representan el detalle de la ejecución del script **primero.sh**.

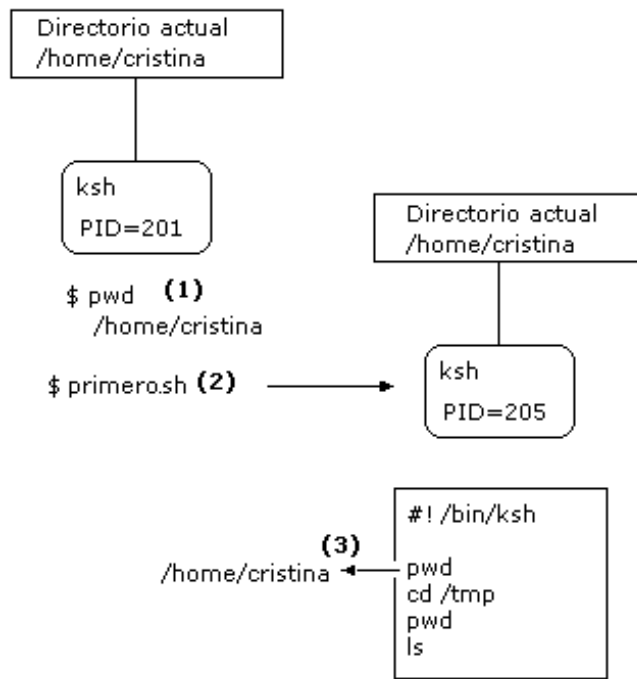


Figura 1: Mecanismo interno aplicado al ejecutar un script en shell - Primera parte

- (1) Antes de ejecutar el script, el usuario provoca la impresión por pantalla del directorio actual. El comando **pwd**, al ser un comando interno, se ejecuta por el shell actual (PID=201). Por tanto, el resultado del comando es **/home/cristina**.
- (2) Ejecución del script **primero.sh**. Un script en shell, al ser un comando externo, provoca la duplicación del shell actual. Por tanto, el script se interpretará por el shell hijo (PID=205). El shell hijo hereda el directorio actual de su padre.
- (3) El shell hijo (PID=205) ejecuta el comando interno **pwd**. Por tanto, el resultado del comando es **/home/cristina**.

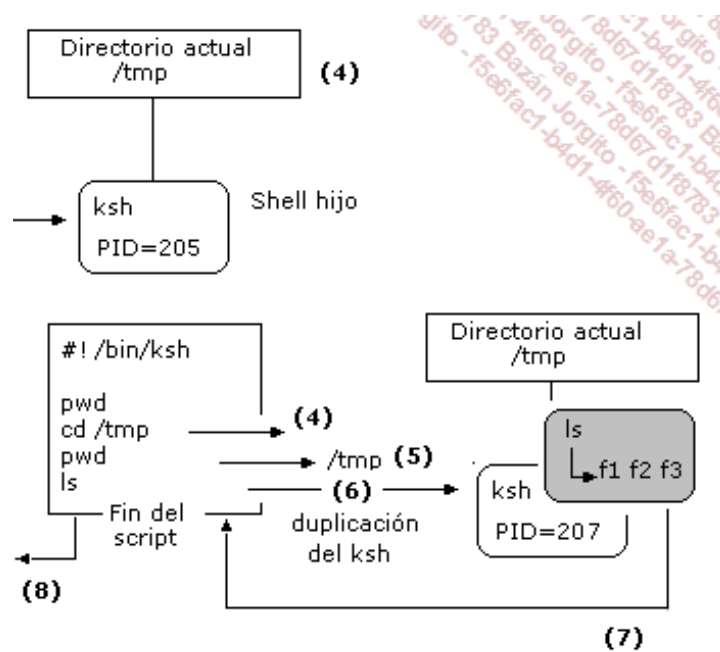


Figura 2: Mecanismo interno aplicado al ejecutar un script en shell - Segunda parte

- (4) El shell hijo (PID=205) ejecuta el comando interno **cd**. Por lo tanto, es él mismo el que realiza el cambio de directorio. El directorio actual del hijo ahora es **/tmp**.

- (5) Impresión por pantalla del directorio actual: **/tmp**
- (6) Ejecución del comando externo **ls**. El shell hijo se duplica a su vez. Por lo tanto, se produce la creación de un nuevo ksh (PID=207) que hereda el directorio actual **/tmp**. El nuevo ksh se reemplaza por el ejecutable **ls**. El proceso **ls** hereda el PID 207 y el directorio actual **/tmp**. **ls** muestra por pantalla la lista de archivos del directorio **/tmp**.
- (7) Cuando el proceso 207 finaliza, se devuelve el control al ksh de PID 205, que prosigue con la ejecución del script. No hay más comandos por ejecutar; por lo tanto, este ksh finaliza a su vez (8) y el shell de PID=201 retoma el control.

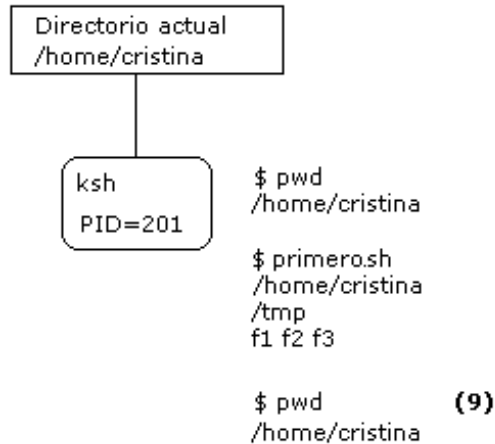


Figura 3: Mecanismo interno aplicado al ejecutar un script en shell - Tercera parte

- (9) El entorno del shell 201 no ha sido modificado, el resultado del comando **pwd** es, por consiguiente, **/home/cristina**.

### 3. Ejecución de un script por el shell actual

Este método se usa esencialmente en el caso de los scripts de entorno (.profile, .kshrc, .bash\_profile, .bashrc) y de forma más genérica cuando se desea modificar el entorno del shell actual. Para ejecutar un script con el shell actual, hay que usar el comando interno ".". No hace falta activar el permiso de ejecución en el script pasado como argumento.

#### Ejemplo

Mostrar el valor actual de la variable **EXINIT**:

```
$ env | grep EXINIT
EXINIT='set number showmode'
```

- La variable **EXINIT** se usa por **vi** y contiene la lista de las opciones que hay que activar o desactivar. Esta variable tiene que exportarse obligatoriamente.

El usuario modifica **EXINIT** en su archivo **.profile**:

```
$ vi .profile
EXINIT='set number showmode autoindent showmatch'
export EXINIT
:wq
```

Mientras el shell actual no haya releído el archivo **.profile**, la modificación no se tendrá en cuenta:

```
$ echo $EXINIT
```


```
set number showmode
```

Provocar la relectura (ejecución) del archivo **.profile** en el shell actual:

```
$ . .profile
```

La modificación ahora ya es considerada por el shell actual:

```
$ env | grep EXINIT
EXINIT='set number showmode autoindent showmatch'
```

 El argumento del comando `.` solamente puede ser un script en shell.

Mensaje clásico de error:

```
$ . .profile
ksh: .: .profile: not found
$ echo $PATH
/usr/bin:/bin
```

El ksh no encuentra el archivo **.profile**; sin embargo, está en su sitio y su nombre es el correcto. El problema yace en el hecho de que el argumento del comando `.` (en este caso, el archivo `.profile`) se busca en los directorios citados en la variable `PATH`. El directorio de inicio del usuario (lugar donde se encuentra el archivo `.profile`) tiene que estar referenciado para poder usar esta sintaxis.

Sin modificar la variable `PATH`, es posible provocar la relectura del archivo `.profile` expresando su ubicación de forma explícita (mediante la ruta absoluta o la ruta relativa).

Ejemplo


```
$ pwd
/home/cristina
$
```

Ruta absoluta al script:

```
$ . $HOME/.profile
```

Ruta relativa al script:

```
$ . ./profile
```

 ¡No confundir los `./`! El primer `.` representa el comando interno del shell y obligatoriamente debe tener un espacio a continuación. `./profile` representa un nombre de archivo expresado con ruta relativa; el primer `.` significa **directorio actual**; el segundo forma parte del nombre del archivo.

## 4. Comentarios

Un comentario comienza por el carácter `#` y termina al final de la línea. Una excepción a esta norma: si la primera línea del archivo empieza por `#!`, el shell espera encontrarse el nombre del intérprete del script justo detrás.

Las líneas en blanco, así como las sangrías (espacios, tabulaciones), se ignoran.

## Ejemplo

```
$ nl primero.sh
1 #! /bin/ksh

2 # Mi primer script
3 # con comentarios
4 pwd      # Mostrar el directorio actual
5 cd /tmp  # Cambio de directorio
6 pwd
7 ls      # Listado de los archivos del directorio actual
$
```

# Variables reservadas del shell

En un script, una serie de variables reservadas son accesibles en modo lectura. Estas variables se inicializan por parte del shell y ofrecen información de diversa índole.

## 1. Los parámetros posicionales

Los scripts en shell son capaces de recuperar los argumentos pasados por línea de comandos con la ayuda de variables especiales, llamadas **parámetros posicionales**.

- **\$#** representa el número de argumentos recibidos por el script.
- **\$0** representa el nombre del script.
- **\$1** representa el valor del primer argumento, **\$2** el valor del segundo y es así hasta **\$9**, que representa el valor del noveno argumento. Ksh y bash permiten usar las variables especiales **\${10}**, **\${11}**, etc. Las llaves son obligatorias cuando el nombre de la variable contiene más de una cifra.
- **\$\*** y **\$@** representan la lista de todos los argumentos (la diferencia entre **\$\*** y **\$@** se presenta en el capítulo Aspectos avanzados de la programación shell - Comparación de las variables **\$\*** y **\$@**).

### Ejemplo

A continuación, un script que muestra el valor de cada parámetro posicional:

```
$ vi miscript.sh
echo "Este script ha recibido $# argumentos"
echo "El nombre del script es: $0"
echo "Mi 1° argumento es      : $1"
echo "Mi 2° argumento es      : $2"
echo "Mi 3° argumento es      : $3"
echo "La lista de todos mis argumentos: $*"
$ chmod u+x miscript.sh
```

Llamada de **miscript.sh** con seis argumentos (ver figura 4):

```
$ miscript.sh f1 f2 f3 f4 /tmp/fic.txt 123
Este script ha recibido 6 argumentos
El nombre del script es: miscript.sh
Mi 1° argumento es      : f1
Mi 2° argumento es      : f2
Mi 3° argumento es      : f3
La lista de todos mis argumentos: f1 f2 f3 f4 /tmp/fic.txt 123
```

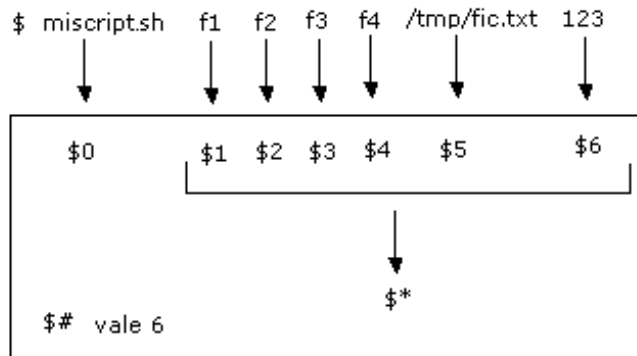


Figura 4: Pasar seis argumentos por línea de comandos

Llamada de **miscript.sh** con tres argumentos (ver figura 5):

```
$ miscript.sh 12 + 24
Este script ha recibido 3 argumentos
El nombre del script es: miscript.sh
Mi 1º argumento es      : 12
Mi 2º argumento es      : +
Mi 3º argumento es      : 24
La lista de todos mis argumentos: 12 + 24
```

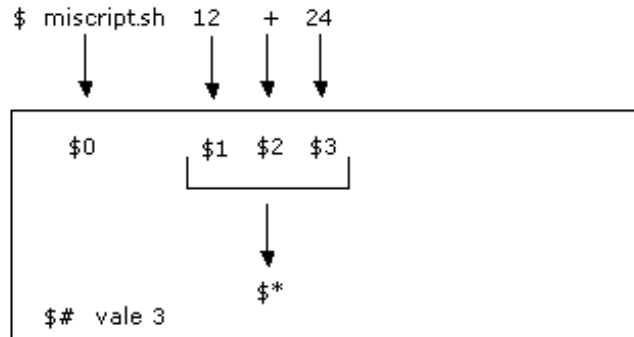


Figura 5: Pasar tres argumentos por línea de comandos

- Los argumentos de la línea de comandos tienen que separarse los unos de los otros con (al menos) un espacio o una tabulación.

## 2. El comando shift

El comando **shift** permite desplazar la lista de argumentos una o varias posiciones hacia la izquierda.

### a. Sintaxis

```
shift [n]
```

donde **n** representa el valor de desplazamiento (n vale 1 por defecto).

### b. Principio

La figura 6 representa el valor de los parámetros posicionales del shell antes (**1**) y después (**3**) de la ejecución del comando **shift** (**2**). Los valores **resultantes** se pierden; por lo tanto, tendrán que ser guardados antes de que se produzca la operación de desplazamiento si fuera necesario. La ejecución del comando **shift** actualiza las variables \$1, \$2 ... \$9, \${10} ..., \$#, \$\* y \$@.

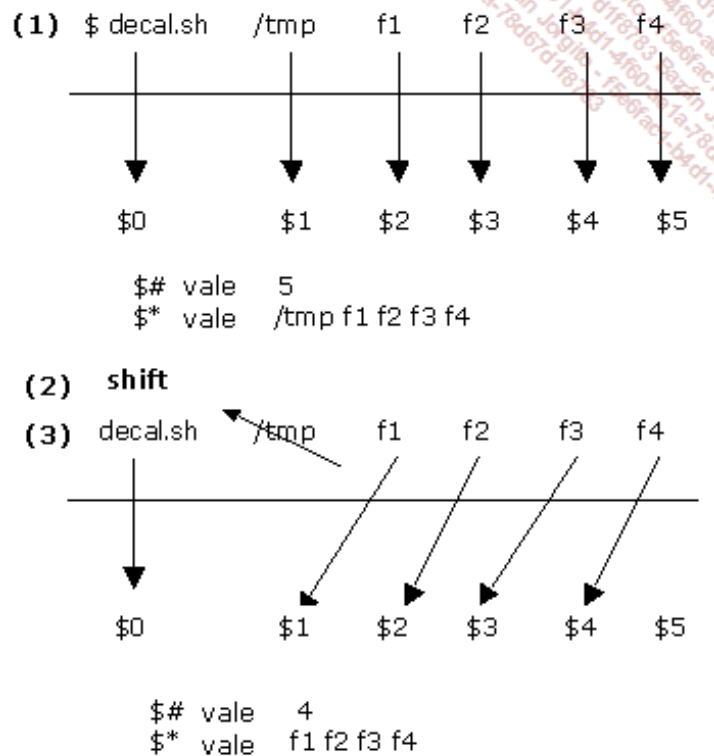


Figura 6: El comando shift

### Ejemplo

El script siguiente recibe como primer argumento el nombre de un directorio y después, a partir del segundo argumento, una serie de nombres de archivo. En este programa, el primer argumento no seguirá el mismo tratamiento que el resto. El nombre del directorio se guarda en una variable (línea 11); después, el comando **shift** expulsa el nombre del directorio (línea 13). Esto permite tener en la variable **\$\*** únicamente la lista de archivos, que se podrá tratar más adelante en un bucle for (línea 25).

```
$ nl decal.sh
1 #! /bin/bash
2 # Mostrar las variables antes de desplazamiento
3 echo "Antes de shift"
4 echo "1° argumento \$1: $1"
5 echo "2° argumento \$2: $2"
6 echo "3° argumento \$3: $3"
7 echo "4° argumento \$4: $4"
8 echo "Todos los argumentos \$*: $*"
9 echo -e "Número de argumentos \$#: $#\n"
10 # Guardar el primer argumento en la variable dir
11 dir=$1
12 # Desplazamiento de 1 posición a la izquierda
13 shift
14 # Mostrar las variables después de desplazamiento
15 echo -e "Después de shift\n"
16 echo "1° argumento \$1: $1"
17 echo "2° argumento \$2: $2"
18 echo "3° argumento \$3: $3"
19 echo "4° argumento \$4: $4"
20 echo "Todos los argumentos \$*: $*"
21 echo "Número de argumentos \$#: $#"
```



```
26 do
27     echo "Guardando $arch ..."
28     ...
29 done
$
```

*Asignación del permiso de ejecución:*

```
$ chmod u+x decal.sh
```

*Ejecución del script:*

```
$ decal.sh /tmp f1 f2 f3 f4 f5 f6
Antes de shift
1° argumento $1: /tmp
2° argumento $2: f1
3° argumento $3: f2
4° argumento $4: f3
Todos los argumentos $*: /tmp f1 f2 f3 f4 f5 f6
Número de argumentos $#: 7

Después de shift

1° argumento $1: f1
2° argumento $2: f2
3° argumento $3: f3
4° argumento $4: f4
Todos los argumentos $*: f1 f2 f3 f4 f5 f6
Número de argumentos $#: 6
Guardando f1 ...
Guardando f2 ...
Guardando f3 ...
Guardando f4 ...
Guardando f5 ...
Guardando f6 ...
$
```

### 3. Código de retorno de un comando

#### a. La variable \$?

Todos los comandos Unix retornan un código de error. Este es un entero comprendido entre 0 y 255. Para el shell, 0 representa el valor verdadero (comando ejecutado con éxito), todo valor mayor de 0 representa el valor falso (error en la ejecución del comando). El código de error del último comando ejecutado se encuentra en la variable especial \$?.

#### Ejemplo

El comando **grep** retorna verdadero cuando la cadena buscada es encontrada, falso en caso contrario:

```
$ grep cristina /etc/passwd
cristina:x:500:500:cristina Lago:/home/cristina:/bin/bash
$ echo $?
0
$ grep marcos /etc/passwd
$ echo $?
1
```

En el interior de un script en shell, la verificación del código de retorno de un comando permite orientar el flujo de ejecución. En este libro se presentarán múltiples ejemplos.

## b. El comando exit

Un script en shell es un comando; por lo tanto, también deberá retornar un código. El comando **exit** permite terminar un script retornando un código de error.

### Ejemplo

A continuación se muestra, en forma de pseudocódigo, un script que devuelve 0 o 1 en función del número de argumentos recibidos:

```
Si $# es diferente de 2
entonces
  Mostrar " Número de argumentos incorrecto "
  exit 1 # Finalización del script con devolución de código 1
(error)
finsi

# El script ha recibido 2 argumentos, puede efectuar el tratamiento
Tratamiento de los argumentos
...
exit 0 # Finalización del script con devolución de código 0 (éxito)
```

- El código de retorno de un script que no llama explícitamente a **exit** tiene por valor el código del último comando ejecutado en el interior del script.

## 4. Otras variables especiales

### a. PID del shell intérprete

**\$\$** representa el PID del shell que interpreta el script (ver figura 7). Esta variable guarda un valor constante durante toda la duración de la ejecución del script.

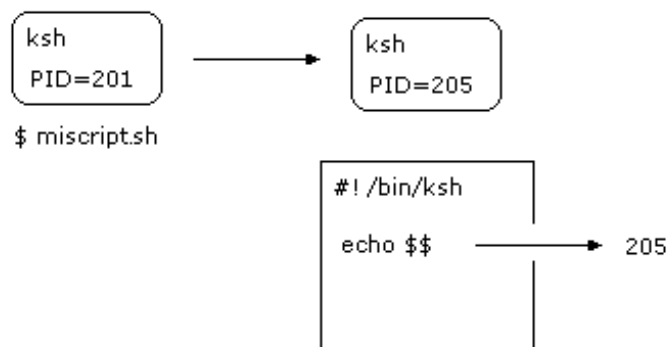


Figura 7: La variable \$\$

### Ejemplo

El script **encuentra.sh** crea un archivo de resultado que tiene un nombre diferente cada vez que se ejecuta:

```
$ nl encuentra.sh
1 archresu=/tmp/haencontrado.$$
2 find "$1" -name "$2" 2> /dev/null 1> $archresu
3 echo "Contenido del archivo $archresu: "
4 more $archresu
```

```
$
```

Este script recibe dos argumentos: un nombre de directorio y un nombre de archivo.

- Línea 1: la variable **archresu** se inicializa con un nombre de archivo que contiene el PID del shell intérprete.
- Línea 2: el comando **find** busca los archivos con nombre **\$2** a partir del directorio **\$1** y almacena los resultados en el archivo **\$archresu**.
- Línea 3: se muestra un mensaje por pantalla.
- Línea 4: el contenido del archivo **\$archresu** se muestra por pantalla.

*Primer ejemplo de ejecución del script*

```
$ encuentra.sh / passwd
Contenido del archivo /tmp/haencontrado.2343:
/etc/passwd
/etc/pam.d/passwd
/usr/bin/passwd
/usr/share/doc/nss_ldap-185/pam.d/passwd
/usr/share/doc/pam_krb5-1.55/krb5afs-pam.d/passwd
/usr/share/doc/pam_krb5-1.55/pam.d/passwd
```

*Segundo ejemplo de ejecución del script*

```
$ encuentra.sh /bin "g*"
Contenido del archivo /tmp/haencontrado.2346:
/bin/gawk-3.1.0
/bin/gawk
/bin/grep
/bin/gzip
/bin/gunzip
/bin/gtar
$
```

*Cada nueva ejecución del script genera, por tanto, un archivo de resultado con un nombre único*

```
$ ls -l /tmp/haencontrado*
-rw-rw-r-- 1 cristina cristina 68 abr 11 17:44 /tmp/haencontrado.2343
-rw-rw-r-- 1 cristina cristina 179 abr 11 17:44 /tmp/haencontrado.2346
$
```

## **b. PID del último proceso ejecutado en segundo plano**

Es posible ejecutar un comando en segundo plano a partir de un script. El PID de este comando se almacena en la variable especial **\$!**.

*Ejemplo*

*Ejecución de un comando en segundo plano a partir de un script en shell:*

```
$ nl segundoplano.sh
1 #! /usr/bin/ksh
2 find / -name $1 1> /tmp/res 2> /dev/null &
3 date
4 echo "*****"
5 echo "Resultado del comando ps: "
6 ps
7 echo "*****"
8 echo "PID de find: $!"
```

```
9 exit 0
$
$ segundoplano.sh passwd
mié feb  4 12:05:59 CET 2015
*****
Resultado del comando ps:
  PID TTY          TIME CMD
 1299 pts/0    00:00:00 bash
 1344 pts/0    00:00:00 ksh
 2240 pts/0    00:00:00 segundoplano.sh
 2241 pts/0    00:00:00 find
 2243 pts/0    00:00:00 ps
*****
PID de find: 2241
$
```

Línea 2: el comando **find** se ejecuta en segundo plano. El shell no espera a que termine; • pasa al comando siguiente. La variable **#!** se inicializa con el PID del proceso find (en este caso, 2241).

- Líneas 3 a 5: ejecución de los comandos **date** y **echo** en primer plano.
- Línea 6: ejecución del comando **ps** en primer plano. El proceso find, que todavía no ha finalizado, se identifica por el PID 2241.
- Línea 8: Se muestra la variable **#!**. Contiene efectivamente el valor 2241.

Un ejemplo de explotación de esta variable se verá posteriormente (en el capítulo Aspectos avanzados de la programación shell - Gestión de un proceso en segundo plano).

# El comando read

## 1. Sintaxis

### Primera sintaxis

```
read var1
read var1 var2 ...
```

### Segunda sintaxis

```
ksh
```

```
read var?"Mensaje de petición"
```

## 2. Lecturas del teclado

El comando **read** lee su entrada estándar y asigna las palabras leídas en la(s) variable(s) cuyo nombre se pasa como argumento. La lista de caracteres separadores de palabras usados por **read** se almacenan en la variable de entorno **IFS** (contiene por defecto los caracteres espacio, tabulación (\t) y salto de línea (\n)).

### Ejemplos

La palabra introducida se almacena en la variable **var1**:

```
$ read var1
hola
$ echo $var1
hola
```

Todas las palabras introducidas se almacenan en la variable **var1**:

La primera palabra se almacena en **var1**, la segunda en **var2**:

```
$ read var1
hola a todo el mundo
$ echo $var1
hola a todo el mundo
```

La primera palabra se almacena en **var1** y el resto de la línea en **var2**:

La palabra se almacena en **var1**, y **var2** se queda vacía:

```
$ read var1 var2
Hasta luego
$ echo $var1
Hasta
$ echo $var2
luego
```

Esta versión de **read** permite especificar un mensaje de petición (en **ksh** solamente):

```
$ read var1 var2
Hasta luego a todo el mundo
$ echo $var1
Hasta
$ echo $var2
luego a todo el mundo
```

```
$ read var1 var2
Gracias
$ echo $var1
Gracias
$ echo $var2
$
```

```
$ read var1?"Entre un valor: "
Entre un valor: Hola
$ echo $var1
Hola
$
```

➤ El comando **read** lee siempre una línea completa (excluyendo el salto de línea).

## 3. Código de retorno

El comando **read** devuelve un código verdadero si no recibe la información **fin de archivo** (materializada por **^d** en el teclado).

### Ejemplos

Un valor se ha introducido, `read` devuelve verdadero:

```
$ read var
Aquí mi entrada de datos
$ echo $?
0
$ echo $var
Aquí mi entrada de datos
```

Si el usuario presiona inmediatamente la tecla [Entrar], introduce la cadena vacía. La variable queda vacía, pero el código es verdadero.

Si el usuario pulsa las teclas `^d`, envía la información de **Fin de archivo** al comando. La variable queda vacía y el código es falso.

```
$ read var
[Entrar]
$ echo $?
0
$ echo $var
$
```

```
$ read var
^d
$ echo $?
1
$ echo $var
$
```

➤ La variable no contendrá nunca `^d`. Solamente la verificación del código puede decir si el usuario introdujo [Entrar] o `^d`.

➤ La introducción de `^d` tiene que realizarse al principio de la línea.

#### 4. La variable IFS

Esta variable contiene por defecto los caracteres siguientes: espacio, tabulación (`\t`) y salto de línea (`\n`).

##### Ejemplo

El comando `od` permite ver el valor de cada byte alojado en **IFS** (el carácter `\c` permite no recuperar en la tubería el salto de línea del comando `echo`):

```
$ echo "$IFS\c" | od -c
00000000      \t \n
00000003
$
```

El contenido de la variable IFS puede modificarse.

##### Ejemplo

Copia del valor actual de IFS para su restauración posterior:

```
$ OLDFIFS="$IFS"
```

Modificación de IFS:

El carácter "espacio" se vuelve un carácter normal, mientras que el carácter ":" se convierte en el separador de palabras:

```
$ IFS=":"
```

```
$ read var1 var2 var3
palabra1:palabra2 palabra3:palabra4
$ echo $var1
palabra1
$ echo $var2
palabra2 palabra3
$ echo $var3
palabra4
```

Restauración del valor inicial de IFS:

```
$ IFS="$OLDIFS"
```

➤ Las expresiones `$IFS` y `$OLDIFS` deben ir obligatoriamente entre comillas dobles para que los caracteres internos no sean interpretados por el shell.

# Ejecución de verificaciones

## 1. Introducción

Este párrafo presenta los dos comandos que permiten efectuar las pruebas:

- El comando `[ ]` también puede ser usado bajo el nombre `test`. Se trata del comando original, compatible con los shells Bourne, ksh y bash.
- El comando `[[ ]]`, que es un subconjunto del anterior (con algunas incompatibilidades, no obstante). Este es compatible con ksh y bash. Se recomienda utilizar este comando en el caso de que la compatibilidad con Bourne no sea necesaria.

## 2. El comando test

Este comando permite hacer verificaciones de archivos, de cadenas de caracteres y de números. Devuelve el código 0 o 1 (verdadero o falso), que el usuario puede consultar mostrando el valor de  `$?` . El comando test ofrece dos sintaxis equivalentes.

### a. Sintaxis

#### Primera sintaxis

```
test expresión
```

#### Segunda sintaxis: comando test

```
[ expresión ]
```

El par de corchetes representa el comando `test`. Los corchetes de apertura y clausura son respectivamente **seguidos** y **precedidos por un espacio**. Esta sintaxis es mucho más agradable de usar.

### b. Verificaciones de archivos

Expresión	¿Opción implementada en Bourne Shell?	Código de retorno
<b>Verificaciones sobre la existencia y el tamaño de un archivo</b>		
<code>-e nombrearch</code>	no	Verdadero si el archivo existe
<code>-s nombrearch</code>	sí	Verdadero si el archivo no está vacío
<b>Verificaciones sobre el tipo de archivo</b>		
<code>-f nombrearch</code>	sí	Verdadero si el archivo es de tipo regular
<code>-d nombrearch</code>	sí	Verdadero si el archivo es de tipo directorio
<code>-h nombrearch</code>	sí	Verdadero si el archivo es de tipo enlace simbólico

-L nombrearch	no	Verdadero si el archivo es de tipo enlace simbólico
-b nombrearch	sí	Verdadero si el archivo es de tipo bloque especial
-c nombrearch	sí	Verdadero si el archivo es de tipo carácter especial
-p nombrearch	sí	Verdadero si el archivo es de tipo tubería con nombre
-S nombrearch	no	Verdadero si el archivo es de tipo socket
<b>Verificaciones sobre los permisos de archivo</b>		
-r nombrearch	sí	Verdadero si el archivo es accesible en lectura
-w nombrearch	sí	Verdadero si el archivo es accesible en escritura
-x nombrearch	sí	Verdadero si el archivo posee el permiso de ejecución
-u nombrearch	sí	Verdadero si el archivo tiene activo el bit setuid
-g nombrearch	sí	Verdadero si el archivo tiene activo el bit setgid
-k nombrearch	sí	Verdadero si el archivo tiene activo el sticky bit
<b>Varios</b>		
nombrearch1 -nt nombrearch2	no	Verdadero si el archivo <b>nombrearch1</b> es más nuevo que el archivo <b>nombrearch2</b>
nombrearch1 -ot nombrearch2	no	Verdadero si el archivo <b>nombrearch1</b> es más viejo que el archivo <b>nombrearch2</b>
nombrearch1 -ef nombrearch2	no	Verdadero si los archivos <b>nombrearch1</b> y <b>nombrearch2</b> referencian el mismo inodo (enlaces físicos)
-O nombrearch	no	Verdadero si el usuario es el propietario del archivo
-G nombrearch	no	Verdadero si el usuario pertenece al grupo propietario del archivo
-t [ desc ]	sí	Verdadero si el descriptor (1 por defecto) está asociado a un terminal

### Ejemplos

**/etc/passwd** es un archivo regular:

```
$ test -f /etc/passwd
$ echo $?
0
```

Misma verificación, pero con la otra sintaxis:



```
$ [ -f /etc/passwd ]
$ echo $?
0
```

Verificación de un archivo que no existe:

```
$ ls
ensayo out out2 primero resu
$ [ -f out3 ]
$ echo $?
1
```

**/tmp** es un directorio:

```
$ [ -d /tmp ]
$ echo $?
0
```

Inicialización de una variable:

```
$ file1=/etc/passwd
```

El archivo contenido en **file1** (/etc/passwd) no es un directorio:

```
$ test -d $file1
$ echo $?
1
$
```

El usuario no tiene el permiso de escritura sobre **/etc/passwd**:

```
$ test -w $file1
$ echo $?
1
$
```

### c. Verificaciones de cadenas de caracteres

Una cadena puede estar compuesta de cualquier secuencia de caracteres (incluidas cifras).

Expresión	Código de retorno
-z cad1	Verdadero si la cadena es de longitud 0 (-z:cero)
-n cad1	Verdadero si la cadena no es de longitud 0 (-n:no cero)
cad1 = cad2	Verdadero si ambas cadenas son iguales
cad1 != cad2	Verdadero si ambas cadenas son diferentes
cad1	Verdadero si la cadena no está vacía

➤ Se recomienda poner el nombre de las variables entre comillas dobles. En efecto: si una variable está vacía, las comillas dobles permiten transformar la expresión en un cadena de longitud 0 (ver ejemplos siguientes). Si una variable vacía no está entre comillas dobles, su sustitución no devuelve nada (es como si no hubiera habido un nombre de variable). ¡En este caso, el comando **test** mostrará un error de sintaxis!

Ejemplos

Inicialización de dos cadenas:

```
$ cad1=root
$ cad2=crisrina
```

Las dos cadenas no son iguales:

```
$ [ "$cad1" = "$cad2" ]
$ echo $?
1
```

**cad1** no está vacía; la verificación devuelve, por lo tanto, verdadero:

```
$ test -n "$cad1"
$ echo $?
0
$
```

**cad1** no está vacía; la verificación devuelve, por lo tanto, falso:

```
$ test -z "$cad1"
$ echo $?
1
```

Las comillas dobles permiten transformar vacío en "cadena vacía":

```
$ test -n "$cad3"          # test -n ""
$ echo $?
1
```

Sin comillas dobles, la variable no se sustituye por vacío, lo que provoca un error de sintaxis (falta el argumento de la opción **-n**):

```
$ test -n $cad3          # test -n
test: argument expected
```

Sin opción, el comando **test** se comporta de la misma manera con una "cadena vacía" o un vacío:

```
$ test "$cad3"          # test ""
$ echo $?
1
$ test $cad3           # test
$ echo $?
1
$
```

#### d. Verificaciones de números

Expresión	Código de retorno
num1 -eq num2	Verdadero si <b>num1</b> es igual a <b>num2</b>
num1 -ne num2	Verdadero si <b>num1</b> es distinto de <b>num2</b>
num1 -lt num2	Verdadero si <b>num1</b> es estrictamente menor que <b>num2</b>
num1 -le num2	Verdadero si <b>num1</b> es menor o igual que <b>num2</b>

num1 -gt num2	Verdadero si <b>num1</b> es estrictamente mayor que <b>num2</b>
num1 -ge num2	Verdadero si <b>num1</b> es mayor o igual que <b>num2</b>

### Ejemplos

Introducción de dos números a través del teclado:

```
$ read num1
12
$ read num2
-3
$ echo $num1
12
$ echo $num2
-3
```

Comparaciones entre dos números:

```
$ [ $num1 -lt $num2 ]
$ echo $?
1
$ [ $num1 -ne $num2 ]
$ echo $?
0
$
```

- No es necesario poner entre comillas dobles las variables de números porque un número no puede tener longitud nula (al contrario que una cadena).

## e. Los operadores

Operador del comando test (por orden de prioridad decreciente)	Significado
!	Negación
-a	Y
-o	O

El orden de evaluación de los operadores se puede imponer mediante el uso de caracteres de agrupación: \( ... \).

### Ejemplos

El comando **test** devuelve verdadero si **\$archivo** no es un directorio:

```
$ archivo=/etc/passwd
$ [ ! -d $archivo ]
$ echo $?
0
$
```

El comando **test** devuelve verdadero si **\$dir** es un directorio y si el usuario tiene el permiso de recorrerlo:

```
$ dir=/tmp
$ echo $dir
```

```
/tmp
$ [ -d $dir -a -x $dir ]
$ echo $?
0
$
```

Es posible modificar la prioridad de los operadores usando los paréntesis:

```
$ [ -w $arch1 -a \( -e $dir1 -o -e $dir2 \) ]
```

➤ Los paréntesis no tienen que ser interpretados por el shell (agrupamiento de comandos), sino por el comando **test**. Por lo tanto, tienen que protegerse.

➤ Tiene que haber un espacio alrededor de los operadores **!**, **-a** y **-o**.

## f. Ejemplo concreto de uso

El comando **test** se usa con las estructuras de control. La estructura de control **if** se presenta a continuación con el fin de ilustrar el comando.

### La estructura de control if

```
if comando1
then
    comando2
    comando3
    ...
else
    comando4
    ...
fi
```

O

```
if comando1 ; then
    comando2
    comando3
    ...
else
    comando4
    ...
fi
```

El principio es el siguiente: el comando situado a la derecha del **if** (comando1) se ejecuta. Si el código de retorno del comando ( **\$?** vale 0) es verdadero, la primera parte del **if** se ejecuta (comando2 y comando3). En caso contrario ( **\$?** es mayor que 0), es la segunda parte la que se ejecuta (comando4). La parte **else** es opcional.

### Ejemplo

Un script en shell que verifica el número de argumentos recibidos:

```
$ nl testarg.sh
1  #! /usr/bin/ksh
2  if [ $# -ne 2 ]
3  then
```

```


4     echo "Uso: $0 arg1 arg2"
5     exit 1
6 fi
7     echo "El número de argumentos es correcto"
8     exit 0
$ testarg.sh arch1 arch2
El número de argumentos es correcto
$ echo $?      # Estado de retorno del script (exit 0)
0
$ testarg.sh arch1
Uso: testarg arg1 arg2
$ echo $?      # Estado de retorno del script (exit 1)
1
$

```

### 3. El comando [[ ]]

ksh	bash
-----	------

El comando `[[ ]]` es una versión enriquecida del comando `test`. Los operadores presentados anteriormente (ver sección El comando `test` de este capítulo) son también válidos (salvo `-a` y `-o`).

 Los operadores lógicos `-a` y `-o` han sido reemplazados por `&&` y `||`.

#### Funcionalidades suplementarias en referencia al comando test

##### Verificaciones sobre cadenas

Las comillas dobles alrededor de los nombres de variables ya no son obligatorias.

##### Ejemplo

*Sintaxis correctas para verificar si una cadena está vacía:*

```

$ echo $vacía
$
$ test -z "$vacía"
$ echo $?
0
$ [ -z "$vacía" ]
$ echo $?
0
$
$ [[ -z $vacía ]]
$ echo $?
0
$ [[ -z "$vacía" ]]
$ echo $?
0
$ [[ $vacía = "" ]]
$ echo $?
0

```

Se han añadido nuevos operadores.

Operadores suplementarios	Código de retorno
<code>cadena = patrón</code>	Verdadero si <b>cadena</b> coincide con el patrón

cadena != patrón	Verdadero si <b>cadena</b> no coincide con el patrón
cadena1 < cadena2	Verdadero si <b>cadena1</b> es lexicográficamente anterior a <b>cadena2</b>
cadena1 > cadena2	Verdadero si <b>cadena1</b> es lexicográficamente posterior a <b>cadena2</b>

Usando las expresiones vistas en el capítulo Mecanismos esenciales del shell - Sustitución de nombres de archivos, es posible comparar cadenas con relación a un patrón. La tabla siguiente recuerda los caracteres especiales que se pueden usar en los patrones de cadenas de caracteres.

Caracteres especiales para patrones de cadenas de caracteres	Significado
<b>Caracteres especiales válidos en todos los shells:</b>	
*	0 a n caracteres
?	1 carácter cualquiera
[abc]	1 carácter de los citados entre los corchetes
[!abc]	1 carácter que no forme parte de los citados entre los corchetes
<b>Caracteres especiales no válidos en Bourne Shell (en bash, activar la opción <code>extglob</code>: <code>shopt -s extglob</code>).</b>	
?(expresión)	La expresión aparece 0 o 1 veces
*(expresión)	La expresión aparece de 0 a n veces
+(expresión)	La expresión aparece de 1 a n veces
@ (expresión)	La expresión aparece 1 vez
!(expresión)	La expresión aparece 0 veces
?(expresión1 expresión2 ...) *(expresión1 expresión2 ...) +(expresión1 expresión2 ...) @(expresión1 expresión2 ...) !(expresión1 expresión2 ...)	Alternativas

### Ejemplo

El script **test\_num.sh** verifica si el valor introducido es un número:

```
$ nl test_num.sh
1  #! /usr/bin/ksh

2  echo "Entre un número: \c"
3  read numero

4  # Se verifica si el número se compone de un conjunto de
5  # cifras precedidas de un signo + o - si se da el caso

6  if [[ $numero = ?([+-])+([0-9]) ]]
7  then
8    echo "$numero es un número"
9    exit 0
10 fi

11 echo "$numero no es un número"
```

```
12 exit 1
$
$ test_num.sh
Entre un número: 456
456 es un número
$ test_num.sh
Entre un número: +456
+456 es un número
$ test_num.sh
Entre un número: -456
-456 es un número
$ test_num.sh

Entre un número: 78*
78* no es un número
$ test_num.sh
Entre un número: az78
az78 no es un número
$
```

### Verificaciones lógicas

- Los operadores `-a` y `-o` se reemplazan por `&&` y `||`.
- Los paréntesis no tienen necesidad de estar protegidos.

Comando test ( [ ] )	Comando [[ ]]	Significado
\( ... \)	( ... )	Agrupación de expresiones
!	!	Negación
-a	&&	Y lógico
-o		O lógico

#### Ejemplo

Con el comando *test*:

```
if [ -w $arch1 -a \( -e $dir1 -o -e $dir2 \) ]
then
...
```

Con el comando *[[ ]]*:

```
if [[ -w $arch1 && ( -e $dir1 || -e $dir2 ) ]]
then
...
```

# Los operadores del shell

Estos operadores permiten ejecutar o no un comando en función del código de retorno de otro comando. La evaluación se realiza de izquierda a derecha.

Operador	Significado
&&	Y lógico
	O lógico

## 1. Evaluación del operador &&

### Sintaxis

```
comando1 && comando2
```

- El segundo comando se ejecuta únicamente si el primer comando devuelve el código verdadero.
- La expresión global es verdadera si los dos comandos devuelven verdadero.

### Ejemplos

El directorio **/tmp/svg** no existe; por lo tanto, el comando **cd** no se ejecuta:

```
$ ls -d /tmp/svg
/tmp/svg: No such file or directory
$ pwd
/export/home/cristina
$ [[ -d /tmp/svg ]] && cd /tmp/svg

$ echo $?          # Código del comando [[ ]]
1
$ pwd
/export/home/cristina
```

El directorio **/tmp/svg** existe; por lo tanto, el comando **cd** se ejecuta:

```
$ mkdir /tmp/svg
$ [[ -d /tmp/svg ]] && cd /tmp/svg
$ pwd
/tmp/svg
$
```

Estas acciones también pueden implementarse con la estructura de control **if**.

```
$ pwd
/export/home/cristina
$ ls -d /tmp/svg
/tmp/svg
$ if [[ -d /tmp/svg ]]
> then      # Prompt PS2 del shell
> cd /tmp/svg
> fi       # Cierre del if: ejecución del comando
$ pwd
/tmp/svg
$
```

## 2. Evaluación del operador ||



## Sintaxis

comando1 || comando2

- El segundo comando se ejecuta únicamente si el primer comando devuelve un código falso.
- La expresión global es verdadera si al menos uno de los dos comandos devuelve verdadero.

### Ejemplo

El directorio **/tmp/svg** no existe; por tanto, el comando **echo** se ejecuta:

```
$ ls -d /tmp/svg
/tmp/svg: No such file or directory
$ pwd
/export/home/cristina
$ [[ -d /tmp/svg ]] || echo "El directorio /tmp/svg no existe"
El directorio /tmp/svg no existe
```

El directorio **/tmp/svg** existe; por tanto, el comando **echo** no se ejecuta.

```
$ mkdir /tmp/svg
$ [[ -d /tmp/svg ]] || echo "El directorio /tmp/svg no existe"
$
```

Estas acciones también pueden interpretarse con la estructura de control **if**.

```
$ ls -d /tmp/svg
/tmp/svg: No such file or directory
$ if [[ ! -d /tmp/svg ]]
> then
> echo "El directorio /tmp/svg no existe"
> fi
El directorio /tmp/svg no existe
$
```

- No confundir los operadores de shell (&& y ||), que efectúan una operación lógica entre dos comandos, con los operadores del comando [[ ]] (&& y ||), que son internos a esta.

# Aritmética

Los shells permiten realizar cálculos de forma nativa con números enteros. La puesta en práctica de aritmética de punto flotante se trata en la sección Aritmética de punto flotante, en este capítulo.

## 1. El comando `expr`

### a. Sintaxis

```
expr num1 operador num2
expr cadena: expresión_regular
```

### b. Operadores

La tabla siguiente presenta los operadores del comando `expr`. Ciertos operadores se construyen con caracteres que tienen un significado especial para el shell. Por lo tanto, es necesario evitar su interpretación; por esta razón, ciertos símbolos tienen que estar precedidos por una barra invertida.

Operadores	Significado
<b>Operadores aritméticos</b>	
num1 + num2	Suma
num1 - num2	Resta
num1 \* num2	Multiplicación
num1 / num2	División
num1 % num2	Módulo
<b>Operadores de comparación</b>	
num1 \> num2	Verdadero si <b>num1</b> es estrictamente mayor que <b>num2</b>
num1 \>= num2	Verdadero si <b>num1</b> es mayor o igual que <b>num2</b>
num1 \< num2	Verdadero si <b>num1</b> es estrictamente menor que <b>num2</b>
num1 \<= num2	Verdadero si <b>num1</b> es menor o igual que <b>num2</b>
num1 = num2	Verdadero si <b>num1</b> es igual a <b>num2</b>
num1 != num2	Verdadero si <b>num1</b> es distinto de <b>num2</b>
<b>Operadores lógicos</b>	
cadena1 \& cadena2	Verdadero si ambas cadenas son verdaderas (valor distinto de cadena nula y 0)
cadena1 \   cadena2	Verdadero si una de las cadenas es verdadera (valor distinto de cadena nula y 0)
<b>Operadores varios</b>	
-num1	Opuesto de num1
\( expresión \)	Agrupación
cadena: expresión_regular	Compara la cadena con la expresión regular (se

mostrarán ejemplos en el capítulo Expresiones regulares).

- Los argumentos del comando **expr** tienen que estar separados por lo menos por un espacio o una tabulación.

### Ejemplos

Suma:

```
$ x=1
$ expr $x + 2
3
```

*Sintaxis incorrecta: el comando recibe un solo argumento que se interpreta como una cadena de caracteres:*

```
$ expr $x+2
1+2
$
```

*El carácter \* se interpreta por el shell (reemplazado por la lista de archivos del directorio actual) antes de la ejecución del comando **expr**:*

```
$ expr $x * 2
expr: syntax error
```

*El carácter \* tiene que protegerse:*

```
$ expr $x \* 2
2
$
```

*Recuperar el resultado del comando dentro de una variable:*

```
$ expr $x - -6
7
$ res=`expr $x - -6`
$ echo $res
7
```

*Agrupación de expresiones:*

```
$ y=2
$ res=`expr $y \* 3 + 1` # Equivalente a res=`expr \( $y \* 3 \) + 1`
$ echo $res
7
$ res=`expr $y \* \( 3 + 1 \)`
$ echo $res
8
$
```

- ¡Los ejemplos presentados a continuación muestran que no hay que confundir la salida del comando con su código de retorno!

*El valor mostrado por una comparación no representa el valor verdadero de la verificación:*

```

$ x=1
$ expr $x \>= 0
1          # Valor mostrado
$ echo $?
0          # Código de retorno

```

En este ejemplo, la impresión por pantalla del comando **expr** no se explota; por lo tanto, la salida estándar se redirige hacia **/dev/null**:

```

$ nl mayorque.sh
 1  #!/usr/bin/ksh
 2  # Verificación del número de argumentos
 3  if [[ $# -ne 2 ]] ; then
 4    echo "Número de argumentos incorrecto"
 5    echo "Uso: $0 num1 num2"
 6    exit 1
 7  fi
 8
 9  # Comparar num1 y num2 con expr
10  if expr $1 \> $2 > /dev/null
11  then
12    echo "expr: $1 es mayor que $2"
13  fi
14  # La verificación también puede realizarse con [[ ]]
(o [ ])
15  if [[ $1 -gt $2 ]]
16  then
17    echo "[[ ]]: $1 es mayor que $2"
18  fi
$
$ mayorque.sh 4 2
expr: 4 es mayor que 2
[[ ]]: 4 es mayor que 2
$

```

## 2. El comando (( ))

ksh	bash
-----	------

### a. Sintaxis

```
((expresión_aritmética))
```

### b. Uso

Este comando presenta mejoras respecto al comando **expr**:

- Tiene multitud de operadores.
- Los argumentos no tienen que estar separados por espacios.
- Las variables no requieren estar prefijadas por \$.
- Los caracteres especiales de shell no tienen que protegerse.
- Las asignaciones se realizan dentro del comando.
- Su ejecución es más rápida.

El comando recupera una gran parte de los operadores del lenguaje C.

Operadores	Significado
<b>Operadores aritméticos</b>	
num1 + num2	Suma
num1 - num2	Resta
num1 * num2	Multiplicación
num1 / num2	División
num1 % num2	Módulo
num1++	Incrementa num1 en 1 (bash/ksh93)
num1--	Decrementa num1 en 1 (bash/ksh93)
<b>Operadores a nivel de bit</b>	
~num1	Complemento a 1
num1 >> num2	Desplazamiento de <b>num1</b> con <b>num2</b> bits hacia la derecha
num1 << num2	Desplazamiento de <b>num1</b> con <b>num2</b> bits hacia la izquierda
num1 & num2	Y bit a bit
num1   num2	O bit a bit
num1 ^ num2	O exclusivo bit a bit
<b>Operadores de comparación</b>	
num1 > num2	Verdadero si <b>num1</b> es estrictamente mayor que <b>num2</b>
num1 >= num2	Verdadero si <b>num1</b> es mayor o igual que <b>num2</b>
num1 < num2	Verdadero si <b>num1</b> es estrictamente menor que <b>num2</b>
num1 <= num2	Verdadero si <b>num1</b> es menor o igual que <b>num2</b>
num1 == num2	Verdadero si <b>num1</b> es igual a <b>num2</b>
num1 != num2	Verdadero si <b>num1</b> es distinto de <b>num2</b>
<b>Operadores lógicos</b>	
!num1	Inversión del valor lógico de <b>num1</b>
&&	Y
	O
<b>Operadores varios</b>	
-num1	Opuesto de <b>num1</b>
num1 = expresión	Asignación
(expresión)	Agrupación
num1 <i>binop</i> = num2	<i>binop</i> representa uno de los operadores siguientes: +, -, *, /, %, >>, <<, &,  , ^. Esta escritura es equivalente a: num1 = num1 <i>binop</i> num2

### Ejemplos

Añadir 10 a la variable x:

```
$ x=10
$ (( x = $x + 10 ))
$ echo $x
20
```

Los espacios no son obligatorios:

```
$ x=10
$ ((x=$x+10))
$ echo $x
20
```

El símbolo \$ puede omitirse:

```
$ x=10
$ ((x=x+10))
$ echo $x
20
$
```

De otra forma más:

```
$ x=10
$ ((x+=10))          # Equivalente a ((x=x+10))
$ echo $x
20
$
```

El script **igual.sh** indica si dos números son iguales:

```
$ nl igual.sh
 1  #! /usr/bin/ksh

 2  # Comparar $1 y $2 con (( ))
 3  if (( $1 == $2 )) ; then
 4    echo "$1 y $2 son iguales"
 5  else
 6    echo "$1 y $2 son diferentes"
 7  fi

$
$ igual.sh 3 2
3 y 2 son diferentes
$ igual.sh 3 3
3 y 3 son iguales
```

Agrupación de verificaciones lógicas:

```
$ if (( ( x > 0 ) && ( y > x ) ))
>> then
>> ...
```

### 3. El comando let

ksh	bash
-----	------

El comando **let** **expresión** es equivalente a **((expresión))**.

Ejemplo

Estos dos comandos son equivalentes:

```
$ let "x = x * 10"  
$ ((x = x * 10))
```

## 4. Aritmética de punto flotante

Solo el **ksh93** ofrece una funcionalidad nativa para trabajar con números de punto flotante.

### a. ksh93

El comando **typeset** permite declarar un número de punto flotante con la precisión deseada.

#### Syntaxis

```
typeset -Fprecisión var1[=val1] [ var2=[val2] ... ]
```

#### Ejemplo

```
$ typeset -F3 iva=0.21  
$ typeset -F2 noimp=153  
$ typeset -F2 monto Iva  
$ (( monto Iva = $noimp * $iva ))  
$ echo $monto Iva  
32.2 # redondeo de 32.13
```

### b. Otros shells

El comando **bc** permite realizar cálculos y se comporta como un filtro. También es posible utilizar el comando **awk** (**nawk** en algunas plataformas), que puede efectuar cálculos sobre números de punto flotante.

#### Ejemplo

Envío de una expresión aritmética al comando **bc**:


```
$ noimp=153  
$ iva=0.21  
$ echo "$noimp * $iva" | bc  
32.13
```

Envío de dos parámetros al comando **awk** (ver capítulo El lenguaje de programación awk):

```
$ echo "$noimp $iva" | awk '{ print $1 * $2 }'  
32.13
```

La misma operación realizada con formato:

```
$ echo "$noimp $iva" | awk '{ printf("%.2f\n", $1 * $2) }'  
32.2
```

 Preste atención a los redondeos de los valores límite, tanto con **typeset** como con **printf**: por ejemplo, el número 2,255 podrá ser redondeado a 2,25, y no a 2,26.

# Sustitución de expresiones aritméticas

ksh	bash
-----	------

Los caracteres de sustitución de comandos se han presentado en la sección Sustitución de comandos, de este capítulo. De igual modo, existen unos caracteres especiales de shell que permiten sustituir una expresión aritmética por su resultado.

## Sintaxis

```
comando argumento1 $((expresión-aritmética)) ... argumenton
```

### Ejemplo

*Recordatorio de la sustitución de comandos:*

```
$ echo "Número de usuarios conectados: `who | wc -l`"
Número de usuarios conectados:      5
```

*Escritura equivalente:*

```
$ echo "Número de usuarios conectados: $(who | wc -l)"
Número de usuarios conectados:      5
```

*El comando (( )) no muestra nada:*


```
$ x=2
$ ((x+1))          # La expresión vale: 3
$ ((x=x+1))       # Se le asigna a x el valor 3 y
la expresión vale 3
$ echo $x
3
```

*Por lo tanto, el comando (( )) no se sustituye por vacío:*

```
$ echo "Después del incremento, x vale: `((x=x+1))`"
Después del incremento, x vale:
```

*Para sustituir la expresión por su valor, hay que usar los caracteres especiales del shell \$(( )).*

```
$ x=2
$ echo "Después del incremento, x vale: $((x=x+1))"
Después del incremento, x vale: 3
$ echo $x
3
```

 No confundir (( )) y \$(( )). (( )) es un comando interno de shell y \$(( )) son los caracteres especiales de shell del mismo tipo que `` o \$().



# Corrección de un script

El shell ofrece algunas opciones que permiten corregir scripts de shell.

## 1. Opción -x

La opción **-x** permite visualizar los comandos que son ejecutados, es decir, después del tratamiento de los caracteres especiales del shell.

### Primera sintaxis

Activar la opción:

```
set -x
```

Desactivar la opción:

```
set +x
```

### Segunda sintaxis

ksh	bash
-----	------

Activar la opción:

```
set -o xtrace
```

Desactivar la opción:

```
set +o xtrace
```

### Tercera sintaxis

Invocar el shell intérprete con la opción **-x**:

```
$ ksh -x script
```

### Ejemplo

A continuación, el script **muestra.sh**, en el cual se ha introducido un error. El desarrollador del script ha escrito, por descuido, **arch** en vez de **\$arch** (línea 4):

```
$ nl muestra.sh
 1  #! /usr/bin/ksh
 2  echo "Nombre del archivo que se visualizará: \c"
 3  read arch
 4  if [[ -f arch ]] ; then
 5    cat $arch
 6  else
 7    echo "Archivo inexistente"
 8  fi
```

Ejecución del script sin corrección. ¡Parece sorprendente (¿o perturbador?) que el archivo **/etc/passwd** se haya encontrado!

```
$ muestra.sh
Nombre del archivo que se visualizará: /etc/passwd
Archivo inexistente
```

Ejecución del script activando la opción **-x**. En este caso, la opción se pasa como argumento al shell intérprete del script. Las líneas mostradas por la corrección se preceden con un signo "+". Se constata que la variable **arch** (`[[ -f arch ]]`) no se está sustituyendo por su valor:

```
$ ksh -x muestra.sh
+ echo Nombre del archivo que se visualizará: \c
Nombre del archivo que se visualizará: + read arch
/etc/passwd
+ [[ -f arch ]]
+ echo Archivo inexistente
Archivo inexistente
```

La activación de la opción también puede hacerse en el interior del script (línea 2):

```
$ nl muestra.sh
 1  #! /usr/bin/ksh

 2  set -x

 3  echo "Nombre del archivo que se visualizará: \c"
 4  ...
```

Corrección del error y ejecución con la modificación. Esta vez, la variable se sustituye por su valor:

```
$ nl muestra.sh
 1  #! /usr/bin/ksh

 2  set -x

 3  echo "Nombre del archivo que se visualizará: \c"
 4  read arch
 5  if [[ -f "$arch" ]] ; then
 6    cat $arch
 7  else
 8    echo "Archivo inexistente"
 9  fi

$ muestra.sh
+ echo Nombre del archivo que se visualizará: \c
Nombre del archivo que se visualizará: + read arch
/etc/passwd
+ [ -f /etc/passwd ]
+ cat /etc/passwd
root:x:0:1:Super-User:/:usr/bin/ksh
daemon:x:1:1:/:
bin:x:2:2:/:usr/bin:
sys:x:3:3:/:
adm:x:4:4:Admin:/var/adm:
...
$
```

## 2. Otras opciones

Función	Bourne Shell, ksh, bash	ksh, bash
Lectura de comandos sin ejecutarlos. Detección de errores de sintaxis del shell.	set -n set +n	set -o noexec set +o noexec
Escritura de los comandos antes de la	set -v	set -o verbose

Ejemplo

Detección de un error de sintaxis del shell en la línea 4 (faltan las comillas dobles), sin lanzar la ejecución del script:

```
$ nl muestra.sh
  1  #! /usr/bin/ksh
    2  echo "Nombre del archivo que se visualizará: \c"
    3  read arch
    4  if [[ -f "$arch ]] ; then
    5      cat $arch
    6  else
    7      echo "Archivo inexistente"
    8  fi

$ ksh -n muestra.sh
muestra.sh[9]: syntax error at line 9: `"' unmatched
$
```

# Las estructuras de control

## 1. if

La estructura de control **if** permite realizar verificaciones. El comando situado tras la palabra **if** se ejecuta. En función del código devuelto por este, el shell orienta el flujo de ejecución en la parte **then** si el comando ha devuelto verdadero ( $\$? \text{ vale } 0$ ) y en la parte **else** si el comando ha devuelto falso ( $\$? > 0$ ). Si el comando ha devuelto falso y no tiene parte **else**, el flujo de ejecución continúa con el primer comando situado bajo **fi**.

### Primera sintaxis

```
if comando1
then
  comando2
  comando3
  ...
else
  comando4
  ...
fi
```

### Segunda sintaxis

La parte **else** es opcional.

```
if comando1
then
  comando2
  comando3
  ...
fi
```

### Tercera sintaxis

Es posible usar la palabra clave **elif**, que significa **sino si**.

```
if comando1
then
  comando2
  ...
elif comando3
then
  comando4
  ...
elif comando5
then
  comando6
  ...
else
  comando7
  ...
fi
```



La palabra clave **fi** representa el cierre del **if**. La palabra clave **elif** no tiene cierre.

### Otras sintaxis

La palabra clave **then** puede colocarse en la primera línea con la condición de usar un **;** para poder separar el comando.

```
if comando1 ; then
  comando2
  ...
fi
```

También es posible anidar estructuras de control. La sintaxis mostrada a continuación es equivalente a la sintaxis **elif** presentada con anterioridad.

```
if comando1
then
  comando2
  ...
else
  if comando3
  then
    comando4
    ...
  else
    if comando5
    then
      comando6
      ...
    else
      comando7
      ...
    fi
  fi
fi
```

### Primer ejemplo

El script **existe\_usuario.sh** recibe como argumento un nombre de usuario. El script muestra por pantalla si el usuario existe o no en el sistema:

```
$ nl existe_usuario.sh
1  #! /bin/bash

2  if [[ $# -ne 1 ]] ; then
3    echo "Número de argumentos incorrecto"
4    echo "Uso: $0 nombre_usuario"
5    exit 1
6  fi

7  if grep -q "^$1:" /etc/passwd
8  then
9    echo "El usuario $1 existe en el sistema"
10 else
11   echo "El usuario $1 no existe en el sistema"
12 fi

13 exit 0

$ existe_usuario.sh cristina
El usuario cristina existe en el sistema
$ existe_usuario.sh marta
El usuario marta no existe en el sistema
```

Este script analiza el número de argumentos introducidos (línea 2) con el comando de test **[[ ]]**. Si

este número es distinto de 1, no se puede realizar el tratamiento, ya que el script termina devolviendo un estado falso (línea 5).

El comando **grep** permite buscar, en modo silencioso (-q), el nombre del usuario en el archivo **/etc/passwd** (línea 7). El carácter **^** usado en la cadena que se está buscando tiene un significado especial para el comando **grep**: significa "comienzo de línea"; es decir, que **grep** debe buscar el nombre "\$1" seguido del carácter ":" solamente al comienzo de la línea. El comando **grep** devuelve un código de retorno con valor verdadero cuando ha encontrado al menos una línea, y falso en caso contrario. La estructura de control "if" orienta el flujo de ejecución en función de este código.

### Segundo ejemplo

El script **codigopos.sh** solicita al usuario la introducción de un código postal y comprueba que sea válido:

```
$ nl codigopos.sh
 1  #! /usr/bin/ksh

 2  echo "Intoduzca un código postal: \c"
 3  read cp
 4  if [[ $cp = 280@([0-4][0-9]|5[0-4]) ]]
 5  then
 6    echo "$cp es un código postal de Madrid"
 7  elif [[ $cp = 28@([0-9][0-9][0-9]) ]]
 8  then
 9    echo "$cp es un código postal de la provincia de Madrid"
10  elif [[ $cp = @(@([0-4][0-9]|5[0-2])[0-9][0-9][0-9]) ]]
11  then
12    echo "$cp es un código postal de España"
13  else
14    echo "$cp no es un código postal de España"
15  fi

$ codigopos.sh
Introduzca un código postal: 28007
28007 es un código postal de Madrid
$ codigopos.sh
Introduzca un código postal: 28220
28220 es un código postal de la provincia de Madrid
$ codigopos.sh
Introduzca un código postal: 08025
08025 es un código postal de España
$ codigopos.sh
Introduzca un código postal: 789651
789651 no es un código postal de España
$ codigopos.sh
Introduzca un código postal: 78a89
78a89 no es un código postal de España
```

Este script será interpretado por un Korn Shell (línea 1). Por tanto, se permite el uso del comando **[[ ]]** y de la comparación con patrones de cadenas de caracteres (líneas 4, 7 y 10).



Para implementar este script en Bourne Shell, la estructura **case** se adapta mejor.

## 2. case

La estructura de control **case** también sirve para realizar verificaciones. Permite comparar una variable con diferentes valores o patrones. Cuando la verificación se presta, reemplaza con ventajas

a **if-elif** debido a que es más leíble.

## a. Sintaxis

```
case $variable in
patrón1) comando
...
;;
patrón2) comando
...
;;
patrón3 | patrón4 | patrón5 ) comando
...
;;

esac
```

## b. Principio

El shell compara el valor de la variable con cada uno de los patrones escritos (la evaluación se realiza de arriba abajo). Cuando un valor se corresponde con uno de los patrones, los comandos relativos a este se ejecutan. Los caracteres `;;` representan el final del tratamiento y permiten salir del **case** (el siguiente comando ejecutado es el situado después de **esac**). Los patrones son cadenas de caracteres que pueden incluir los caracteres especiales presentados en el capítulo Mecanismos esenciales del shell - Sustitución de nombres de archivos. El carácter `|` permite expresar la alternativa entre múltiples patrones.



El olvido de los caracteres `;;` genera un error de sintaxis.

La tabla siguiente es un recordatorio de los caracteres especiales que hemos nombrado anteriormente.

Caracteres especiales para patrones de cadenas de caracteres	Significado
<b>Caracteres especiales que se pueden usar en los patrones (válidos en todos los shells):</b>	
*	De 0 a n caracteres cualesquiera.
?	1 carácter cualquiera.
[abc]	1 carácter de los citados entre los corchetes.
[!abc]	1 carácter que no sea ninguno de los citados entre los corchetes.
<b>Caracteres especiales no válidos en Bourne Shell (en bash, activar la opción <code>nextglob: shopt -s extglob</code>):</b>	
?(expresión)	0 o 1 veces la expresión.
*(expresión)	De 0 a n veces la expresión.
+(expresión)	De 1 a n veces la expresión.
@(expresión)	1 vez la expresión.
!(expresión)	0 veces la expresión.

```
?(expresión1|expresión2|...)  
*(expresión1|expresión2|...)  
+(expresión1|expresión2|...)  
@(expresión1|expresión2|...)  
!(expresión1|expresión2|...)
```

Alternativas.

### c. Uso

#### Primer ejemplo

El script **menu.sh** muestra un menú, lee la opción introducida por el usuario y verifica con la ayuda de una estructura **case** la validez de la elección:

```
$ nl menu.sh  
1  #! /usr/bin/ksh  
  
2  echo "- 1 - Copia de seguridad "  
3  echo "- 2 - Restaurar "  
4  echo "- 3 - Fin"  
  
5  read respuesta?"Su elección: " # Sintaxis ksh  
  
6  case "$respuesta" in  
7    1) echo "Su elección ha sido copia de seguridad"  
8      # Ejecución de la copia de seguridad  
9      ;;  
10   2) echo "Su elección ha sido restaurar"  
11     # Ejecución de la restauración  
12     ;;  
13   3) echo "Fin del tratamiento"  
14     echo "Hasta luego ..."  
15     exit 0  
16     ;;  
17   *) echo "Opción incorrecta"  
18     echo "Adiós ..."  
19     exit 1  
20     ;;  
21  esac  
$ menu.sh  
- 1 - Copia de seguridad  
- 2 - Restaurar  
- 3 - Fin  
Su elección: 1  
Su elección ha sido copia de seguridad  
$ menu.sh  
- 1 - Copia de seguridad  
- 2 - Restaurar  
- 3 - Fin  
Su elección: 2  
Su elección ha sido restaurar  
$ menu.sh  
- 1 - Copia de seguridad  
- 2 - Restaurar  
- 3 - Fin  
Su elección: 3  
Fin del tratamiento  
Hasta luego ...  
$ menu.sh  
- 1 - Copia de seguridad  
- 2 - Restaurar  
- 3 - Fin  
Su elección: 4  
Opción incorrecta  
Adiós ...
```



```
$
```

En la línea 17, el carácter \* significa **Cualquier otra cadena de caracteres**.

### Segundo ejemplo

El script **codigopos.sh** solicita la entrada de un código postal y verifica la validez de este:

```
$ nl codigopos.sh
 1  #! /usr/bin/sh
 2  echo "Introduzca un código postal: \c"
 3  read cp
 4  case "$cp" in
 5      280[0-4][0-9] | 2805[0-4] )
 6      echo "$cp es un código postal de Madrid"
 7      ;;
 8      28[0-9][0-9][0-9] )
 9      echo "$cp es un código postal de la provincia de Madrid"
10      ;;
11      [0-4][0-9][0-9][0-9][0-9] | 5[0-2][0-9][0-9][0-9] )
12      echo "$cp es un código postal de España"
13      ;;
14      *)
15      echo "$cp no es un código postal de España"
16      ;;
17  esac
```

El script **codigopos.sh** es compatible con Bourne Shell.

### Tercer ejemplo

El script **testnum.sh** solicita la entrada de un número y verifica su validez:

```
$ nl testnum.sh
 1  #! /usr/bin/ksh
 2  echo "Introduzca un número: \c"
 3  read num
 4  case "$num" in
 5      ?(+)([0-9]) )
 6      echo "$num es un número entero positivo"
 7      ;;
 8      -+([0-9]) )
 9      echo "$num es un número entero negativo"
10      ;;
11
12      ?(+)([0-9]).+([0-9]) )
13      echo "$num es un número real positivo"
14      ;;
15      -+([0-9]).+([0-9]) )
16      echo "$num es un número real negativo"
17      ;;
18      *)
19      echo "$num no es un número"
20      exit 1
```

```

21     ;;
22
23     esac
24     exit 0
$ testnum.sh
Introduzca un número: 4
4 es un número entero positivo
$ testnum.sh
Introduzca un número: +4
+4 es un número entero positivo
$ testnum.sh
Introduzca un número: -4
-4 es un número entero negativo
$ testnum.sh
Introduzca un número: +4.5
+4.5 es un número real positivo
$ testnum.sh
Introduzca un número: 4.5
4.5 es un número real positivo
$ testnum.sh
Introduzca un número: -45.689
-45.689 es un número real negativo
$ testnum.sh
Introduzca un número: 789621
789621 es un número entero positivo
$ testnum.sh
Introduzca un número: 7.
7. no es un número
$ testnum.sh
Introduzca un número: az45
az45 no es un número
$

```

El script **testnum.sh** usa expresiones complejas, por lo que no es compatible con Bourne Shell.

### 3. Bucle for

#### Sintaxis

La estructura de control **for** permite tratar una lista de valores representados por **val1 val2 val3 ... valn**. A cada iteración del bucle, la variable **var** se inicializa con uno de los valores de la lista (los valores se tratan en el orden de su enumeración). La lista de valores puede citarse directamente o generarse por sustitución de caracteres especiales de shell.

#### Lista de valores citadas directamente

```

for var in val1 val2 val3 ... valn
do
    comando
    ...
done

```

#### Lista de valores generada por sustitución de variable

```

for var in $variable
do
    comando
    ...
done

```

#### Lista de valores generada por sustitución de comando

```
for var in `comando`
do
    comando
    ...
done
```

## Lista de valores generada por sustitución de caracteres de generación de nombres de archivo

```
for var in *.c
do
    comando
    ...
done
```

## Lista por defecto: argumentos de la línea de comandos

```
for var
do
    comando
    ...
done
```

## Equivalente a

```
for var in $*
do
    comando
    ...
done
```

### Primer ejemplo

La lista de valores del script **despegue.sh** se cita directamente:

```
$ nl despegue.sh
 1  #! /usr/bin/ksh

 2  for despegue in 4 3 2 1 "Fuego  !!"
 3  do
 4      echo "$despegue"
 5  done
$ despegue.sh
4
3
2
1
Fuego  !!
```

### Segundo ejemplo

El script **testarch.sh** recibe una lista de nombres de archivo como argumento y da una indicación al usuario sobre el tipo de archivo para cada elemento. La lista de valores se genera por sustitución de una variable (\$\*).

```
$ nl testarch.sh
 1  #! /usr/bin/ksh

 2  echo "Lista de los argumentos recibidos: \n$*"

 3  for arch in $*          # Equivalente a "for arch"
```

```

4 do
5   if [[ -f "$arch" ]]
6   then
7     echo "$arch es un archivo regular"
8   elif [[ -d "$arch" ]]
9   then
10    echo "$arch es un directorio"
11  elif [[ -e "$arch" ]]
12  then
13    echo "$arch no es ni un archivo regular ni un
directorio"
14  else
15    echo "$arch no existe"
16  fi
17 done
$ ls -lId /etc
drwxrwxr-x 26 root sys 4096 Nov 5 12:16 /etc
$ ls -lId /etc/passwd
-rw-r--r-- 1 root other 3983 Nov 4 10:29 /etc/passwd
$ ls -lId /dev/console
crw--w---- 1 root tty 0, 0 Mar 10 03:14 /dev/console
$ ls -lId /backup
/backup: No such file or directory
$
$ testarch.sh /etc /etc/passwd /dev/console /backup
Lista de los argumentos recibidos:
/etc /etc/passwd /dev/console /backup
/etc es un directorio
/etc/passwd es un archivo regular
/dev/console no es ni un archivo regular ni un directorio
/backup no existe

```

### Tercer ejemplo

El script **tratausuario.sh** trata todos los usuarios definidos en el archivo **/etc/passwd** cuyo nombre empiece por "cuenta". La lista de valores se genera a partir de una sustitución de comando:

```

$ nl tratausuario.sh
1  #! /usr/bin/ksh

2  for user in `cut -d":" -f1 /etc/passwd | grep "^cuenta"`
3  do
4    echo "Tratamiento del usuario: $user"
5    # ...
6  done

$
$ tratausuario.sh
Tratamiento del usuario: cuenta01
Tratamiento del usuario: cuenta02
Tratamiento del usuario: cuenta03
Tratamiento del usuario: cuenta04

```



Puede consultar el capítulo Expresiones regulares para saber el significado del carácter ^ usado con **grep**.

## 4. Bucle while

### a. Sintaxis

```
while comando1
do
    comando2
    ...
done
```

La estructura de control **while** permite iterar mientras un comando devuelva el código verdadero. A cada iteración del bucle, el comando especificado tras la palabra **while** se ejecuta. Cuando este devuelve verdadero, el shell ejecuta los comandos internos situados entre **do** y **done**, después vuelve a subir hasta **while** para ejecutar de nuevo el comando. Si este devuelve falso, el shell sale del **while** y ejecuta el comando situado después de **done**.

## b. Uso

### Ejemplo

El script **suma.sh** muestra por pantalla la suma de los números introducidos:

```
$ nl suma.sh
 1  #! /usr/bin/ksh
 2  suma=0
 3  echo "Introduzca un número por línea, ^d para mostrar
la suma:"
 4  while read num
 5  do
 6      # La verificación siguiente no es compatible
 7      # con Bourne shell
 8      if [[ "$num" != ?([+-])+([0-9]) ]]
 9      then
10          echo "El valor introducido no es un número"
11          continue
12      fi
13      # La sintaxis siguiente es equivalente a ((suma=suma+num))
14      # y a ((suma=$suma+$num))
15      # En Bourne Shell: suma=`expr $suma + $num`
16      ((suma+=num))
17  done
18  echo "Suma: $suma"
19  exit 0
```

El comando **read** se ejecuta a cada iteración del bucle (línea 4). Mientras el usuario no introduzca **^d** (EOF), el comando devuelve verdadero, que es lo que provoca entrar en el cuerpo del **while** (línea 5). El valor introducido se verifica (línea 8): si este valor corresponde a un número, este se añade al valor actual de **suma.sh** (línea 16). En caso contrario, el comando interno **continue** (línea 11) permite subir inmediatamente a **while** (línea 4) para leer un nuevo valor (el comando **continue** se detalla en este capítulo, sección Las estructuras de control - break y continue).

```
$ suma.sh
Introduzca un número por línea, ^d para mostrar la suma:
1
3
a
El valor introducido no es un número
4
^d
```

```
Suma: 8
$
```

### c. Bucle infinito

El shell ofrece el comando interno `:`, cuyo interés reside en el hecho de que devuelve siempre verdadero. Colocado tras un `while`, permite construir un bucle infinito.

#### Ejemplo

El comando `:` es interno al shell y devuelve siempre verdadero:

```
$ type :
: is a shell builtin
$ :
$ echo $?
0
```

➤ Bash y ksh también ofrecen el comando interno `true` equivalente a `:`, cuya ventaja es la de tener un nombre más explícito. Las plataformas Unix ofrecen el comando externo `/usr/bin>true` (o `/bin>true`).

#### Ejemplos

*bucle infinito usando el comando `::`*

```
while :
do
...
done
```

*Bucle infinito usando el comando `true`:*

```
while true
do
...
done
```

El script `menubucle.sh` retoma el programa `menu.sh` visto en la sección Las estructuras de control - case, de este capítulo. La impresión por pantalla y el tratamiento se incluyen dentro de un bucle infinito (línea 2), lo que permite volver a mostrar por pantalla el menú después de cada tratamiento (excepto para el caso de la salida).

```
$ nl menubucle.sh
1  #! /usr/bin/ksh
2  while :
3  do
4      echo "- 1 - Copia de seguridad "
5      echo "- 2 - Restaurar "
6      echo "- 3 - Fin"
7
8      read respuesta?"Su elección: "
9
10     case "$respuesta" in
11         1) echo "Su elección ha sido copia de seguridad
<Confirmar>\c"
12             # Ejecución de la copia de seguridad
13             read x
14             ;;
```

```

15     2) echo "Su elección ha sido restaurar <Confirmar>\c"
16     # Ejecución de la restauración
17     read x
18     ;;
19     3) echo "Fin del tratamiento"
20     echo "Hasta luego ..."
21     exit 0
22     ;;
23     *) echo "Opción incorrecta <Confirmar>\c"
24     read x
25     ;;
26     esac
27     done

```

Para que el usuario tenga tiempo de visualizar las impresiones por pantalla, el script usa el comando **read x** (líneas 13, 17 y 24): el comando provoca una pausa en pantalla esperando que el usuario pulse [Entrar]. El contenido de la variable **x** no se utilizará.

### Resultado de la ejecución

```

$ menubucle.sh
- 1 - Copia de seguridad
- 2 - Restaurar
- 3 - Fin
Su elección: 1
Su elección ha sido copia de seguridad <Confirmar>
[Enter]
- 1 - Copia de seguridad
- 2 - Restaurar
- 3 - Fin
Su elección: 2
Su elección ha sido restaurar <Confirmar>
[Entrar]
- 1 - Copia de seguridad
- 2 - Restaurar
- 3 - Fin
Su elección: 4
Opción incorrecta <Confirmar>
[Enter]
- 1 - Copia de seguridad
- 2 - Restaurar
- 3 - Fin
Su elección: 3
Fin del tratamiento
Hasta luego ...
$

```

## 5. until

### a. Sintaxis

```

until comando1
do
    comando2
    ...
done

```

La estructura de control **until** permite iterar hasta que un comando devuelva el código verdadero. A cada iteración del bucle, el comando especificado detrás de la palabra **until** se ejecuta. Cuando este devuelve un código falso, el shell ejecuta los comandos internos situados entre **do** y **done**; después vuelve a **until** para ejecutar de nuevo el comando. Tan pronto como este último

devuelve verdadero, el shell sale de **until** y ejecuta el comando situado justo después de **done**.

## b. Uso

### Ejemplo


El script **espera.sh** realiza iteraciones hasta que el archivo cuyo nombre está en **\$2** llega al directorio representado por **\$1**: el archivo **\$2** llegará por red.

```
$ nl espera.sh
 1 #!/bin/bash
 2 # Verificación del número de argumentos
 3 if [[ $# -ne 2 ]]
 4 then
 5     echo "Uso: $0 directorio archivo"
 6     exit 1
 7 fi
 8 # El 2º argumento tiene que ser un directorio
 9 if [[ ! -d $1 ]]
10 then
11     echo "$1 no es un directorio"
12     exit 2
13 fi
14 # Los argumentos son correctos
15 # Hasta que el archivo exista, el script duerme 2 segundos
16 until [[ -e $1/$2 ]]
17 do
18     sleep 2
19 done
20 # El archivo ha llegado
21 date=$(date '+%d%m%y_%H%M')
22 mv $1/$2 $HOME/$2.$date
23 mail $LOGNAME <<FIN
24 El archivo $HOME/$2.$date ha llegado.
25 FIN
26 echo "$0: Ha recibido correo."
27 exit 0
$
```

El script comprueba inicialmente la validez de los argumentos:

Si el número de argumentos es diferente de 2 (línea 3), el script termina devolviendo un código de error.

Si **\$1** no contiene el nombre de un directorio válido (línea 9), el script termina devolviendo un código de error.

 La variable **\$2** no se verifica porque este archivo tiene que llegar por red. Por lo tanto, hay muchas posibilidades de que no exista en el sistema de archivos en el momento de ejecutar el script.

El script comprueba cada 2 segundos si el archivo ha llegado. Esta acción se ejecuta asociando una estructura de control **until** y el comando **[[ ]]** (línea 16). Una vez que el comando de test **[[ ]]** devuelve verdadero, el flujo de ejecución pasa a la línea 21.

Una variable **date** se inicializa con una cadena de caracteres que contiene la fecha y hora de



llegada del archivo (línea 21). El archivo se mueve y se renombra hacia el directorio de inicio del usuario que ha lanzado el script (\$HOME). El contenido de la variable **date** se concatena al nombre actual (línea 22).

Se envía un correo al usuario conectado (\$LOGNAME). El script usa la doble redirección en lectura (línea 23).

Un mensaje se muestra por pantalla (línea 26) y el script finaliza devolviendo un código verdadero (línea 27).

### Resultado de la ejecución

El archivo **/tmp/resu.log** no existe aún:

```
$ ls -l /tmp/resu.log
/tmp/resu.log: No such file or directory
```

El script se ejecuta en segundo plano:

```
$ espera.sh /tmp resu.log &
[1] 4481
```

Creación manual del archivo esperado:

```
$ > /tmp/resu.log
```

Mensaje generado por la línea 26:

```
espera.sh: Usted ha recibido un correo
```

Lectura del buzón de correo:

```
$ mail
Message 1:
From cristina   Mié feb 43 13:05:21 2015
To: cristina
Date: Sat, 4 Feb 2015 13:05:21 +0200 (CEST)

El archivo /home/cristina/resu.log.030215_1305 ha llegado.
&
```

## **Variante del script espera.sh**

Al usuario le gustaría asegurarse de que, en el momento en que el archivo esperado aparece en el sistema de archivos, este esté presente en el disco en su totalidad (transferencia del archivo finalizada).

El ejemplo siguiente propone una variante del script **espera.sh** que va a verificar la presencia de un archivo testigo. El transmisor envía el archivo de datos primero y a continuación el archivo testigo, lo que significa para la máquina receptora que, si el archivo testigo está en el disco, entonces el archivo de datos ya se ha recibido completamente.

### Ejemplo

```
$ nl espera.sh
 1  #!/bin/bash
   2  # Verificación del número de argumentos
   3  # Es necesario al menos 2 argumentos
```

```

4  if [[ $# -lt 2 ]]
5  then
6    echo "Uso : $0 directorio archivo [ testigo ] "
7    exit 1
8  fi

9  # El 2º argumento tiene que ser un directorio
10 if [[ ! -d $1 ]]
11 then
12   echo "$1 no es un directorio"
13   exit 2
14 fi

15 # Nombre del archivo testigo por defecto : "testigo"
16 archTestigo=${3:-testigo}

17 # Los argumentos son correctos
18 # Hasta que llegue el archivo testigo, el script duerme 2 segundos
19 until [[ -e $1/$archTestigo ]]
20 do
21   sleep 2
22 done

23 # Si el archivo testigo ha llegado, el archivo de datos
ha llegado también
24 # Por seguridad, se verifica
25 if [[ ! -e $1/$2 ]] ; then
26   echo "El archivo $archTestigo ha llegado pero el archivo
de datos no existe"
27   exit 1
28 fi

29 # Tratamiento del archivo de datos
30 date=$(date '+%d%m%y_%H%M')
31 mv $1/$2 $HOME/$2.$date

32 mail $LOGNAME <<FIN
33 El archivo $HOME/$2.$date ha llegado.
34 FIN

35 echo "$0 : Ha recibido correo"
36 exit 0

```

Comentario:

Línea 4: el script espera como mínimo dos argumentos: el nombre del archivo testigo es opcional, ya que se tiene previsto un nombre por defecto (el archivo por defecto se llama "testigo").

Línea 16: la variable **archTestigo** se inicializa con "\$3" si la variable no está vacía y con el nombre de archivo "testigo" en caso contrario.

Línea 19: al contrario de la versión anterior, el script está iterando mientras el archivo testigo esté ausente.

Línea 25: una vez que el archivo testigo está presente, significa que el archivo de datos está presente también, íntegramente. Se ejecuta al menos una verificación de la existencia del archivo de datos. La parte de tratamiento es idéntica.

## 6. break y continue

Los comandos internos **break** y **continue** pueden usarse en el interior de los bucles **for**, **while**, **until** y **select** (ver capítulo Aspectos avanzados de la programación shell - Gestión de menús con select).

El comando **break** permite salir de un bucle, mientras que el comando **continue** permite subir a la condición del bucle.

## Sintaxis

Salir del bucle de primer nivel:

```
break
```

Salir del bucle de nivel n:

```
break n
```

Subir a la condición del bucle de primer nivel:

```
continue
```

Subir a la condición del bucle de nivel n:

```
continue n
```

## Ejemplo

El script **suma2.sh** es una variante del script **suma.sh**. Esta versión usa un bucle infinito y muestra el mensaje de petición en cada iteración del bucle:

```
$ nl suma2.sh
 1  #!/usr/bin/ksh
 2  suma=0
 3  # Bucle infinito
 4  while true
 5  do
 6    echo "Introduzca un número: \c"
 7    # Entrada de un número
 8    if read num
 9    then
10      # Si la entrada es incorrecta, se sube a la condición
del bucle de nuevo
11      if [[ "$num" != ?([+-])+([0-9]) ]]
12      then
13        echo "El valor entrado no es un número"
14        continue
15      fi
16      # La entrada es correcta, se realiza la suma
17      (( suma = $suma + $num ))
18    else
19      # El usuario ha entrado ^d: Salida del bucle
20      break
21    fi
22  done
23  # Impresión del resultado
24  echo "\nLa suma es: $suma\c"
25  exit 0
```

El script **suma2.sh** utiliza un bucle infinito (línea 4). En cada iteración del bucle se muestra un mensaje de petición de datos (línea 6) y se solicita la entrada de un número (línea 8). Si el comando **read** devuelve verdadero (el usuario ha introducido cualquier elemento excepto ^d): el valor de entrada se comprueba (línea 11). Si no se corresponde con un número entero, se vuelve a la línea 4 (gracias a la línea 14) para realizar una nueva introducción de datos. Si el usuario ha introducido un número, se efectúa la suma (línea 17).

Si **read** devuelve falso (el usuario ha introducido ^d), se ejecuta el comando **break** (línea 20) y el flujo de ejecución prosigue en la línea 24.

```
$ suma2.sh
Introduzca un número: 4
Introduzca un número: a*
El valor introducido no es un número
Introduzca un número: 3
Introduzca un número: Enter
El valor introducido no es un número
Introduzca un número: ^d
La suma es: 7
```

# Ejercicios

## 1. Variables, caracteres especiales

### a. Ejercicio 1: variables

1. Defina una variable que contenga su nombre. Muestre esta variable.
2. Defina una variable que contenga su nombre seguida de su apellido. Muestre esta variable.
3. Elimine las dos variables (dejándolas indefinidas).

### b. Ejercicio 2: variables

Defina una variable que contenga su apellido, y otra que contenga su nombre. Utilizando un solo echo, muestre las dos variables, separadas por un carácter de subrayado (*apellido\_nombre*).

### c. Ejercicio 3: sustitución de comando

1. En un solo comando, muestre la fecha actual:

```
Hoy es mié 4 feb 14:32:22 CET 2015
```

2. Igual pero aplique a la fecha el formato siguiente:

```
Hoy es 04/02/2015
```

### d. Ejercicio 4: caracteres de protección

El directorio actual contiene los archivos **f1**, **f2** y **f3**:

```
$ ls  
f1 f2 f3
```

¿Qué obtendrá con los comandos siguientes?:

1. 

```
$ echo *
```
2. 

```
$ echo \*
```
3. 

```
$ echo ""
```
4. 

```
$ echo '*'
```
5. 

```
$ edad=20  
$ echo $edad
```
6. 

```
$ echo \$edad
```

7. `$ echo "$edad"`
8. `$ echo '$edad'`
9. `$ echo "Tú eres $(logname) y tienes -> $edad años"`
10. `$ echo Tú eres $(logname) y tienes -> $edad años`

## 2. Variables, visualización y lectura del teclado

### a. Ejercicio 1: variables

Escriba un script **primer.sh** y realice las operaciones siguientes:

- Inicialice una variable **nombre**.
- Inicialice una variable **miFecha** que contendrá la fecha actual.
- Muestre las dos variables.

Ejecute este script.

### b. Ejercicio 2: parámetros posicionales

Comandos filtro utilizados: **wc** (ver capítulo Los comandos filtro).

Escriba un shell script **wcount.sh** que produzca el resultado siguiente:

```
$ bash wcount.sh oso pájaro
El nombre del script es: wcount.sh
El primer argumento es: oso
El segundo argumento es: pájaro
Todos los argumentos: oso pájaro
Número total de argumentos: 2
El primer argumento contiene: 3 caracteres
El segundo argumento contiene: 6 caracteres
```

### c. Ejercicio 3: lectura de teclado

Escriba un script **hello.sh** que:

1. Solicite al usuario la introducción de su nombre y lo almacene en una variable.
2. Solicite al usuario la introducción de su apellido y lo almacene en otra variable.
3. Muestre un mensaje de bienvenida al usuario.
4. Muestre el PID del shell que ejecuta el script.

Ejemplo

```
$ hello.sh
Introduzca su nombre: Cristina
Introduzca sus apellidos: Perez Lopez
Buenos días Cristina Perez Lopez
El PID del shell es 2569
```

## 3. Tests y aritmética

Estos ejercicios no necesitan el empleo de la estructura de control **if**. Solo deben emplearse los comandos `[ ]`, `[[ ]]`, `expr`, `(( ))` y `$(( ))`. Muestre el estado de retorno de los comandos para saber si el comando ha retornado verdadero (0) o falso (>0).

### a. Ejercicio 1: tests a los archivos

1. Verifique si el archivo (o directorio) `/etc` existe.
2. Verifique si es posible acceder al archivo `/etc/hosts` en lectura.
3. Verifique si el archivo `/etc/hosts` es ejecutable.
4. Verifique si el archivo `/usr` es un directorio y si se puede atravesar.
5. Verifique si el archivo `/dev/null` es un archivo especial de dispositivo.

### b. Ejercicio 2: tests de cadenas de caracteres

Definir las siguientes variables:

```
$ s1=si
$ s2=no
$ vacia=""
$ arch1=informe.pdf
```

1. Pruebe si `$s1` es igual a `$s2`.
2. Pruebe si `$s1` es diferente de `$s2`.
3. Pruebe si `$vacía` está vacía.
4. Pruebe si `$vacía` no está vacía.
5. Pruebe si `$arch1` termina en `.doc` (bash/ksh solamente).
6. Pruebe si `$arch2` termina en `.doc` o en `.pdf`.

### c. Ejercicio 3: tests numéricos

Definir las variables `num1` y `num2` con los valores siguientes:

```
$ num1=2
$ num2=100
```

Verifique si `$num1` es mayor que `$num2` empleando los comandos `[ ]`, `[[ ]]` y `(( ))`.

### d. Ejercicio 4: aritmética

En la línea de comandos, inicialice dos variables numéricas `num1` y `num2`:

```
$ num1=3
$ num2=5
```

1. Inicialice una variable `res` con la suma de `num1` y `num2`. Muestre `res`. Proporcione una solución compatible con Bourne y una solución específica bash/ksh.
2. Sin inicializar la variable `res`, muestre por pantalla la suma de dos números. Proporcione a su vez las dos soluciones.
3. Inicialice una variable `res` con el resultado de la multiplicación de `num1` y `num2`. Muestre `res`. Proporcione una solución compatible con Bourne y una solución

específica bash/ksh.

## e. Ejercicio 5: operadores lógicos de los comandos [ ], [[ ]] y operadores lógicos del shell

Ejecute los comandos siguientes:

```
$ > arch
$ chmod 444 arch
$ ls -l arch
-r--r--r-- 1 cristina perez 0 2 feb 17:23 arch
```

1. Si el archivo **\$arch** no es ejecutable, muestre "Permiso x no indicado".
2. Si el archivo **\$arch** no es ni ejecutable ni accesible para escritura, muestre "Permisos wx no indicados".

## 4. Estructuras de control if, case, bucle for

### a. Ejercicio 1: los comandos [ ] y [[ ]], la estructura de control if

Comandos filtro útiles: **awk** (ver capítulo Los comandos filtro).

Escriba un script **compare.sh**:

- Número de argumentos recibidos: dos nombres de archivo ordinarios.
- Verificar que el número de argumentos es igual a 2 y que los archivos son de tipo ordinario.
- Si los argumentos son correctos, el script deberá decir si los dos archivos son del mismo tamaño; en caso contrario, deberá decir cuál es el mayor de los dos.

*Ejemplo*

```
$ compare.sh /etc/hosts
Uso: ./compare.sh archivo1 archivo2

$ compare.sh /etc/hosts /etc/passwd
El archivo /etc/passwd es el mayor de los dos. Tamaño:
1910 bytes ...
```

### b. Ejercicio 2: estructura de control case, bucle for

Escriba un script **tipoarch.sh** que tome los nombres de archivo por argumento. Si el archivo termina en **.doc** o **.pdf**, muestre un mensaje específico. En caso contrario, muestre "Ni DOC, ni PDF".

*Ejemplo*

```
$ tipoarch.sh f1.doc f2.pdf f3.txt
f1.doc: Archivo DOC
f2.pdf: Archivo PDF
f3.txt: Ni DOC, ni PDF
```

## 5. Bucles

### a. Ejercicio 1: bucle for, comando tr

Comandos filtro útiles: **tr** (ver capítulo Los comandos filtro). Otros comandos útiles: **mv**.

Escriba un script **may\_min.sh**:



- Argumento opcional: un nombre de directorio. El valor por defecto será el directorio actual.
- Verificar que el posible argumento es un directorio.
- El script renombrará los archivos del directorio: los nombres de archivo en mayúsculas serán convertidos a minúsculas.

## b. Ejercicio 2: bucle for, aritmética

Comandos filtro útiles: **grep**, **wc** (ver capítulo Los comandos filtro). Otros comandos útiles: **ps**.

Escribir un script **proc\_users.sh**:

- Argumentos: uno o más nombres de usuario.
- Verificar el número de argumentos recibido: debe haber al menos un argumento.
- Para cada usuario recibido como argumento, muestre en la pantalla el número de procesos que le pertenezcan. Si el usuario no está definido en el sistema, este será ignorado.

### Ejemplos

```
$ proc_users.sh
Uso: ./proc_users.sh user1 user2 ... usern

$ proc_users.sh cristina olivier daniel
El usuario cristina tiene 8 procesos en ejecución
El usuario olivier no tiene procesos activos
daniel no es un usuario válido
```

## c. Ejercicio 3: bucles for, while

Comandos filtro útiles: **grep** (ver capítulo Los comandos filtro). Otros comandos útiles: **file**, **find**.

Escriba un script **consulta.sh**:

- Argumento opcional: un nombre de directorio. Verifique que el argumento recibido es un directorio.
- Si el script no recibe un nombre de directorio, trate el directorio actual.
- Busque todos los archivos ordinarios que se encuentren bajo este directorio (incluyendo subniveles).
- Para cada archivo de contenido texto accesible en lectura, pregunte al usuario si desea consultar el archivo. Los archivos que no sean de texto se ignorarán.
- El usuario podrá introducir 's', 'S', 'si', 'SI' para consultar el archivo, 'n', 'N', 'no', 'NO' para no consultarlo, o 'q' para salir del script. Cualquier otra respuesta generará una nueva pregunta al usuario.
  - Si el usuario desea consultar el archivo, muestre el contenido paginado de este en la pantalla.
  - Si el usuario no desea consultar el archivo, pase al archivo siguiente.

# Comparación de las variables \$\* y @\$

## 1. Uso de \$\* y de @\$

Este capítulo presenta otras funcionalidades utilizadas en la programación shell que completan las abordadas en el capítulo Las bases de la programación shell.

Las variables \$\* y @\$ contienen la lista de los argumentos de un script shell. Cuando no están entre comillas dobles, son equivalentes.

### Ejemplo

El script **test\_var1.sh** muestra el valor de cada argumento de la línea de comandos:

```
$ nl test_var1.sh
 1  #!/usr/bin/ksh
 2  contador=1 ;
 3  for arg in $*      # Equivalente a @$
 4  do
 5      echo "Argumento $contador: $arg"
 6      ((contador+=1))
 7  done
$
```

A continuación, un ejemplo de llamada al script:

```
$ test_var1.sh a b c "d e f" g
```

Primera etapa: el **shell actual** trata los caracteres de protección antes de ejecutar el script. A este nivel, los espacios internos en "d e f" se protegen y no son vistos como separadores de palabras, sino como caracteres cualesquiera.

Segunda etapa: el **shell hijo** interpreta el script sustituido \$\* (o @\$) por a b c d e f g.

```
for arg in "$@"
o for arg in $*
```



```
for arg in a b c d e f g
```

Los espacios que estaban protegidos a nivel de shell de trabajo no estarán a nivel del shell hijo. A continuación, el resultado de la ejecución del script:

```
$ test_var1.sh a b c "d e f" g
Argumento 1: a
Argumento 2: b
Argumento 3: c
Argumento 4: d
Argumento 5: e
Argumento 6: f
Argumento 7: g
```

## 2. Uso de "\$\*"

Las comillas dobles eliminan el significado especial de los espacios que están dentro de \$\*.

### Ejemplo

En el script **test\_var2.sh**, la variable **\$\*** se encuentra entre comillas dobles.

```
$ nl test_var2.sh
1  #!/usr/bin/ksh

2  contador=1 ;
3  for arg in "$*"
4  do
5      echo "Argumento $contador: $arg"
6      ((contador+=1))
7  done
```

A continuación, un ejemplo de llamada al script:

```
$ test_var2.sh a b c "d e f" g
```

Primera etapa: el **shell actual** trata los caracteres de protección antes de ejecutar el script. A este nivel los espacios internos en **"d e f"** se protegen y no son vistos como separadores de palabras, sino como caracteres cualesquiera.

Segunda etapa: el **shell hijo** interpreta el script sustituido **"\$\*"** por **"a b c d e f g"**. Las comillas dobles alrededor de **\$\*** protegen todos los espacios internos. Por tanto, estos son vistos como caracteres regulares:

```
for arg in "$*"
↓
```

```
for arg in "a b c d e f g"
```

Para el shell hijo, hay un solo argumento. A continuación se muestra el resultado de la ejecución del script:

```
$ test_var2 a b c "d e f" g
Argumento 1: a b c d e f g
$
```

### 3. Uso de "\$@"

Poner la variable **\$@** entre comillas dobles permite conservar la protección realizada a nivel del shell de trabajo.

#### Ejemplo

En el script **test\_var3.sh**, la variable **\$@** se pone entre comillas dobles.

```
$ nl test_var3.sh
1  #!/usr/bin/ksh

2  contador=1 ;
3  for arg in "$@"
4  do
5      echo "Argumento $contador: $arg"
6      ((contador+=contador))
7  done
```

A continuación se muestra un ejemplo de llamada al script.

```
$ test_var3.sh a b c "d e f" g
```

Primera etapa: el **shell en ejecución** trata los caracteres de protección antes de ejecutar el script. A este

nivel, los espacios internos en "**d e f**" se protegen y no son vistos como separadores de palabras, sino como caracteres cualesquiera.

Segunda etapa: el **shell hijo** interpreta el script ya sustituido "\$@" por **a b c "d e f" g**. La protección de los espacios internos en **d e f** se conserva por el shell hijo.

```
for arg in "$@"
```



```
for arg in a b c "d e f" g
```

Para el shell hijo, hay cinco argumentos. A continuación, el resultado de la ejecución del script:

```
$ test_var3.sh a b c "d e f" g
Argumento 1: a
Argumento 2: b
Argumento 3: c
Argumento 4: d e f
Argumento 5: g
```

# Sustitución de variables

ksh	bash
-----	------

La sustitución de variables ha sido tratada en el capítulo Las bases de la programación shell - Las variables de usuario. Esta sección presenta nuevas funcionalidades disponibles en los shells bash y ksh.

## 1. Longitud del valor contenido en una variable

### Sintaxis


```
${#variable}
```

### Ejemplo

```
$ var="mi cadena"  
$ echo ${#var}  
9  
$
```

## 2. Manipulación de cadenas de caracteres

Las funcionalidades que se presentan a continuación permiten manipular el valor contenido en una variable.

 El contenido de la variable nunca se modifica.

### a. Eliminar el fragmento más pequeño de la izquierda

#### Sintaxis

```
${variable#patrón}
```

donde **patrón** es una cadena de caracteres que puede incluir los caracteres especiales \*, ?, [], ? (expresión), +(expresión), \*(expresión), @(expresión), !(expresión) (ver capítulo Mecanismos esenciales del shell - Sustitución de nombres de archivos).

El carácter # significa "Cadena lo más corta posible empezando por la izquierda".

### Ejemplo

Mostrar la variable **linea** sin su primer campo:

```
$ linea="campo1:campo2:campo3"  
$ echo ${linea#*:*}  
campo2:campo3
```

### b. Eliminar el fragmento más grande de la izquierda

#### Sintaxis

```
${variable##patrón}
```

Los caracteres ## significan "Cadena lo más larga posible empezando por la izquierda".

### Ejemplo

Mostrar el último campo de la variable **linea**:

```
$ linea="campo1:campo2:campo3"
$ echo ${linea##*:}
campo3
```

## c. Eliminar el fragmento más pequeño de la derecha

### Sintaxis

```
${variable%patrón}
```

El carácter % significa "Cadena lo más corta posible empezando por la derecha".

### Ejemplo

Mostrar la variable **linea** sin su último campo:

```
$ linea="campo1:campo2:campo3"
$ echo ${linea%:*}
campo1:campo2
```

## d. Eliminar el fragmento más grande de la derecha

### Sintaxis

```
${variable%%patrón}
```

Los caracteres %% significan "Cadena lo más larga posible empezando por la derecha".

### Ejemplo

Mostrar el primer campo de la variable **linea**:

```
$ linea="campo1:campo2:campo3"
$ echo ${linea%%:*}
campo1
```



En bash, no hay que olvidarse de activar la opción **extglob** con el comando **shopt -s extglob** para que el shell interprete los patrones usando expresiones complejas.

# Tablas

ksh	bash
-----	------

Los shells recientes permiten trabajar con tablas de una dimensión. Los elementos de una tabla se indexan a partir del número 0.

## 1. Asignar un elemento

### Sintaxis

```
nombretabla[indice]=valor
```

### Ejemplo

```
$ tab[0]=10  
$ tab[2]=12
```

Una casilla de la tabla no inicializada es vacío.

## 2. Referenciar un elemento

### Sintaxis

```
${nombretabla[indice]}
```

### Ejemplo

*Mostrar el elemento de índice 0:*

```
$ echo ${tab[0]}  
10
```

*Mostrar el elemento de índice 2:*

```
$ echo ${tab[2]}  
12
```

*Mostrar el elemento de índice 1. Como nunca ha sido inicializado, la expresión se sustituye por vacío:*

```
$ echo ${tab[1]}  
$
```



Las llaves son obligatorias.

## 3. Asignación global de una tabla

ksh93	bash
-------	------

### Sintaxis

```
nombretabla=(val1 val2 ... valn)
```

### Ejemplo

```
$ tab=(10 11 12 13 12)
```


```
ksh
```

### **Sintaxis**

```
set -A nombretabla val1 val2 ... valn
```

### Ejemplo

```
$ set -A tab 10 11 12 13 12
```

 Ambas sintaxis reinician completamente la tabla si esta contenía ya valores.

## 4. Referenciar todos los elementos de una tabla

### **Sintaxis**

```
${nombretabla[*]}
```

### Ejemplo

```
$ echo ${tab[*]}  
10 11 12 13 12  
$
```

## 5. Obtener el número de elementos de una tabla

### **Sintaxis**

```
${#nombretabla[*]}
```

### Ejemplo

```
$ echo ${#tab[*]}  
5  
$
```

## 6. Obtener la longitud de un elemento de una tabla

### **Sintaxis**

```
${#nombretabla[indice]}
```

### Ejemplo

Mostrar el elemento de índice 0:

```
$ echo ${tab[0]}
```



Mostrar el número de caracteres del elemento de índice 0:

```
$ echo ${#tab[0]}
2
```

## 7. Tablas asociativas

ksh93	bash4
-------	-------

Las tablas asociativas son tablas donde los elementos son índices para una cadena de caracteres (clave del elemento).

Definir e inicializar una tabla asociativa:

```
$ typeset -A tabAssoc
$ tabAssoc=([apellidos]="Perez Lopez" [nombre]=Cristina)
```

Mostrar el valor asociado a una clave:

```
$ echo ${tabAssoc[apellidos]}
Perez Lopez
```

Mostrar la lista de las claves:

```
$ echo ${!tabAssoc[*]}
apellidos nombre
```

Mostrar la lista de valores:

```
$ echo ${tabAssoc[*]}
Perez Lopez Cristina
```

Bucle sobre una tabla asociativa:

```
$ for cle in ${!tabAssoc[*]}
> do
>   echo "Clave : $clave , Valor : ${tabAssoc[$clave]}"
> done
Clave : apellidos , Valor : Perez Lopez
Clave : nombre , Valor : Cristina
```

## Inicialización de parámetros posicionales con set

El comando **set** llamado sin ninguna opción pero seguido de argumentos asigna estos últimos a los parámetros posicionales (**\$1**, **\$2**, ..., **\$\***, **\$@**, **\$#**). Esto permite manipular fácilmente el resultado de sustituciones diversas.

### Ejemplo

Ejecución del comando **date**:

```
$ date
Tue Mar 18 23:57:43 MET 2014
```

El resultado del comando **date** se asigna a los parámetros posicionales:

```
$ set `date`
$ echo $1
Tue
$ echo $2
Mar
$ echo $4
23:57:53
$ echo $*
Tue Mar 18 23:57:53 MET 2014
$ echo $#
6
$
```

# Funciones

Las funciones sirven para agrupar comandos que tienen que ejecutarse en varios sitios en el transcurso de la ejecución de un script.

## 1. Definición de una función

La definición de una función tiene que hacerse antes de su primera llamada.

### Primera sintaxis

Los paréntesis indican al shell que `mifuncion` es una función.

Definición de la función:

```
mifuncion() {  
  comando1  
  comando2  
  ...  
}
```

Llamada a la función:

```
mifuncion
```

### Segunda sintaxis

ksh	bash
-----	------

La palabra clave `function` reemplaza los paréntesis usados en la primera sintaxis.

Definición de la función:

```
function mifuncion {  
  comando1  
  comando2  
  ...  
}
```

Llamada a la función:

```
mifuncion
```

En un script que contenga funciones, los comandos situados fuera del cuerpo de las funciones se ejecutan secuencialmente.

Para que los comandos localizados en una función se ejecuten, hay que realizar una **llamada a una función**. Una función puede llamarse tanto desde del programa principal como desde otra función.

### Ejemplos

*Uso de la primera sintaxis:*

```
$ nl func2.sh  
1  f1() {                               # Definición de la función  
2    echo "En f1"  
3  }  
4  echo "1° comando"  
5  echo "2° comando"  
6  f1                                   # Llamada a la función  
7  echo "3° comando"
```

```
$ func2.sh
1° comando
2° comando
En f1
3° comando
$
```

*El mismo script usando la segunda sintaxis:*

```
$ nl func3.sh
 1 function f1 {                # Definición de la función
 2     echo "En f1"
 3 }

 4 echo "1° comando"
 5 echo "2° comando"
 6 f1                          # Llamada a la función
 7 echo "3° comando"
```

- Una vez que una función se ha definido, es considerada por el shell como un comando interno. Las sintaxis usadas para llamar a una función o ejecutar un comando Unix son exactamente las mismas.

## 2. Código de retorno de una función

Como todo comando Unix, una función devuelve un código de error. Por defecto, el valor devuelto corresponde al código del último comando ejecutado en la función. El comando **return** permite a una función finalizar la ejecución de sí misma y, si fuera necesario, devolver un código de error explícito.

- **return** permite a una función devolver un valor numérico entero comprendido entre 0 y 255. No hay que usar **return** para devolver un valor que no hará el papel de código de error (por ejemplo, no es necesario devolver el resultado de un cálculo).

El código de error devuelto por la función se guarda en la variable **\$?**.

### Ejemplo

*El script **existe\_usuario.sh** solicita la entrada de un nombre de usuario y comprueba si este existe o no en el sistema:*

```
$ nl existe_usuario.sh
 1  #!/usr/bin/ksh

 2  # Provocar una pausa en pantalla
 3  function pausa {
 4      echo "Pulse una tecla para continuar"
 5      read x
 6  }

 7  # Saber si un usuario existe en el sistema
 8  function existe_usuario {
 9      echo "Introduzca el nombre de un usuario: \c"
10      read user
11      if grep -q "^$user:" /etc/passwd ; then
12          return 0
```

```

13  fi
14  return 1
15  }
16  # Programa principal
17  while true
18  do
19  clear
20  echo "- 1 - Saber si un usuario existe en el sistema"
21  echo "- 2 - Saber el uid de un usuario"
22  echo "- 3 - Fin"
23
24  echo "Su elección: \c"
25  read eleccion
26
27  case $eleccion in
28
29      1) # Llamada a la función
30         # Verificar el código de retorno de la función ($?)
31         if existe_usuario
32         then
33             echo "El usuario $user existe en el sistema"
34         else
35             echo "El usuario $user no existe en el sistema"
36         fi
37         pausa          # Llamada a la función pausa
38         ;;
39      2)
40         echo "Elección no implementada" ;;
41
42      3) exit 0 ;;
43
44  esac
45  done

```

El script **existe\_usuario.sh** contiene dos definiciones de función:

- La función **pausa** (línea 3) permite realizar una pausa en pantalla (mediante el comando **read**). Se invoca en la línea 37 para que el usuario del script pueda visualizar los mensajes por pantalla antes de hacer la llamada al comando **clear** (línea 19). El código de retorno de esta función (que tiene por valor el código del comando **read x**) no se utiliza.
- La función **existe\_usuario** (línea 8) solicita la introducción del nombre de usuario y verifica con la ayuda del comando **grep** (línea 11) si este existe en el archivo `/etc/passwd`. Si **grep** devuelve verdadero, la función finaliza y devuelve a su vez un código verdadero (línea 12). En caso contrario, finaliza devolviendo un código falso (línea 14).

La llamada a la función **existe\_usuario** se considera como un comando Unix y, por tanto, se puede colocar detrás de un **if** (línea 31). En función del valor devuelto por la función (0 o 1), la estructura de control orientará el flujo de ejecución hacia la línea 33 (si el código es verdadero) o hacia la línea 35 (si el código es falso).

El shell ejecutará a continuación la función **pausa**. El código de retorno de esta última no se utilizará.

### 3. Ámbito de las variables

Las variables de usuario son globales. En el script **existe\_usuario.sh**, la variable **user** se inicializa en la función (línea 10) y se reutiliza en el programa principal (líneas 33 y 35). Cada referencia al nombre de la variable **user** hace referencia al mismo espacio de memoria (ver figura 1).

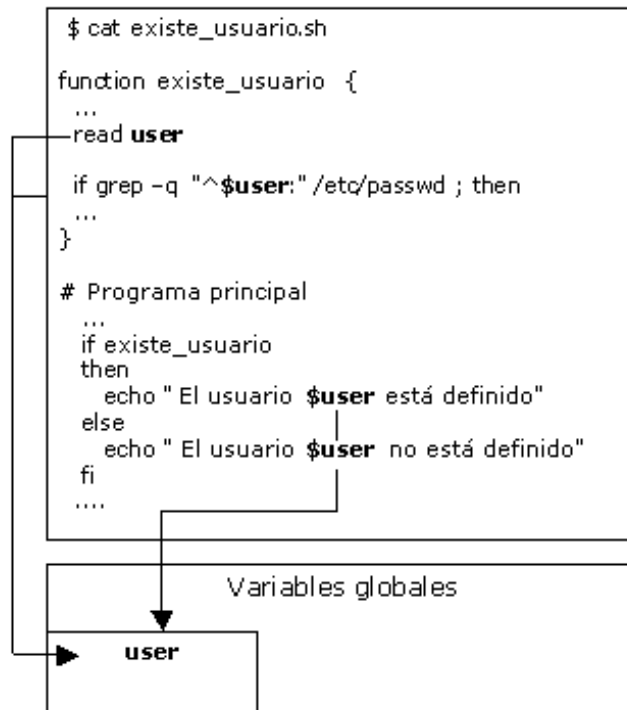


Figura 1: Las variables de usuario son globales por defecto.

#### 4. Definición de variables locales

ksh	bash
-----	------

El comando **typeset** permite definir variables locales en una función.

##### Sintaxis

```
typeset variable
typeset variable=valor
```

##### Ejemplo

La función **f1** del script **locales.sh** define una variable local:

```

$ nl locales.sh
1  #! /usr/bin/ksh

2  function f1 {
3    # var1 es una variable local
4    typeset var1

5    echo "En f1 => var1 antes: $var1"
6    var1=100
7    echo "En f1 => var1 después: $var1"

8    echo "En f1 => var2 antes: $var2"
9    var2=200
10   echo "En f1 => var2 después: $var2"
11  }

12  # var1 y var2 son 2 variables globales
13  var1=1
14  var2=2

```

```

15 echo "En el programa principal => var1 antes de invocar
a f1: $var1"
16 echo "En el programa principal => var2 antes de invocar
a f1: $var2"
17 f1
18 echo "En el programa principal => var1 después de invocar
a f1: $var1"
19 echo "En el programa principal => var2 después de invocar
a f1: $var2"

$ locales.sh
En el programa principal => var1 antes de invocar a f1: 1
En el programa principal => var2 antes de invocar a f1: 2
En f1 => var1 antes:
En f1 => var1 después: 100
En f1 => var2 antes: 2
En f1 => var2 después: 200
En el programa principal => var1 después de invocar a f1: 1
En el programa principal => var2 después de invocar a f1: 200
$

```

El programa principal define (líneas 13 y 14) dos variables globales, **var1** y **var2**, inicializadas respectivamente con los valores **1** y **2**.

La función **f1** define una variable local **var1** (línea 4). Toda modificación de **var1** en **f1** no afectará de modo alguno a la variable global del mismo nombre (la variable global **var1** es inaccesible en la función). No sucede lo mismo con la variable global **var2**, cuyo contenido se modifica en la función.

## 5. Paso de parámetros

El shell ofrece un mecanismo que permite enviar valores a una función. Los valores recibidos por la función serán asignados **automáticamente** a las **variables especiales de shell locales a la función**.

Estas variables locales se llaman \$1, \$2, ..., \$9, \${10} ..., \$\*, \$@, \$#. La variable \$0 contiene permanentemente el nombre del script.

### Ejemplo

El script **param.sh** contiene tres funciones. Cada una muestra sus parámetros recibidos:

```

$ nl param.sh
1  #!/usr/bin/ksh

2  function f1 {
3    echo "Parámetros de la función f1:"
4    echo "\$0 => $0"
5    echo "\$1 => $1"
6    echo "\$2 => $2"
7    echo "\$3 => $3"
8    echo "\$* => $*"
9    echo "\$# => $#"
10 }

11 function f2 {
12  echo "Parámetros de la función f2:"
13  echo "\$0 => $0"
14  echo "\$1 => $1"
15  echo "\$2 => $2"
16  echo "\$3 => $3"
17  echo "\$* => $*"
18  echo "\$# => $#"
19 }

```

```

20 function f3 {
21     echo "Parámetros de la función f3:"
22     echo "\$0 => $0"
23     echo "\$1 => $1"
24     echo "\$2 => $2"
25     echo "\$3 => $3"
26     echo "\$* => $*"
27     echo "\$# => $#"
```

```

28 }

29 echo "Parámetros del programa principal: "
30 echo "\$0 => $0"
31 echo "\$1 => $1"
32 echo "\$2 => $2"
33 echo "\$3 => $3"
34 echo "\$* => $*"
35 echo "\$# => $#"
```

```

36 f1 1 2 3
37 f2 100 200 fic.c
38 f3 $2 $3
```

Invocación del script **param.sh**:

```

$ param.sh f1 f2 f3 f4 f5
Parámetros del programa principal:
$0 => param.sh
$1 => f1
$2 => f2
$3 => f3
$* => f1 f2 f3 f4 f5
$# => 5
Parámetros de la función f1:
$0 => param.sh
$1 => 1
$2 => 2
$3 => 3
$* => 1 2 3
$# => 3
Parámetros de la función f2:
$0 => param.sh
$1 => 100
$2 => 200
$3 => fic.c
$* => 100 200 fic.c
$# => 3
Parámetros de la función f3:
$0 => param.sh
$1 => f2
$2 => f3
$3 =>
$* => f2 f3
$# => 2
```

El script **param.sh** recibe cinco argumentos (**f1**, **f2**, **f3**, **f4** y **f5**). Estos argumentos son accesibles en el programa principal (líneas de la 29 a la 38) mediante las variables `$1`, `$2`, `$3`, `$4`, `$5`, `$*`, `$@`, `$#`.

La función **f1** se invoca (línea 36) con los valores **1**, **2** y **3**. En el cuerpo de la función (líneas de la 3 a la 9), estos valores son accesibles en las variables locales `$1`, `$2`, `$3`, `$*`, `$@`, `$#`. Por tanto, los argumentos del programa principal son inaccesibles durante la ejecución de la función.

La función **f2** se invoca (línea 37) con los valores **100**, **200** y **fic.c**. En el cuerpo de la función (líneas de la 12 a la 18), estos valores son accesibles mediante las variables locales `$1`, `$2`, `$3`, `$*`, `$@`, `$#`. Por tanto, los argumentos del programa principal son inaccesibles durante la ejecución de la función.



La función **f3** muestra como una función puede trabajar con los argumentos recibidos en el programa principal. La función se invoca (línea 38) con los valores \$2 y \$3, es decir, **f2** y **f3**. En el cuerpo de la función (líneas de la 21 a la 27), los valores recibidos son accesibles mediante las variables locales \$1, \$2, \$\*, \$@, \$#. Los argumentos segundo y tercero del programa principal son, por tanto, pasados por parámetro a la función f3, que los recupera como primer y segundo parámetro.

## 6. Utilizar la salida de una función

Una llamada a una función puede, como sucede con los comandos, estar colocada dentro de los caracteres de sustitución de comandos ('' o \$()).

### Ejemplo

El script **get\_uid.sh** recibe como argumento un nombre de usuario y busca su uid:

```
$ nl get_uid.sh
 1  #!/usr/bin/ksh
 2
 3  function get_uid
 4  {
 5      grep "^$1:" /etc/passwd | cut -d':' -f3
 6  }
 7
 8  # -----
 9  # Definición de variables globales
10  # -----
11  UID=""          # UID de un usuario
12
13 # -----
14 # Programa principal
15 # -----
16
17 # La función muestra un UID o vacío
18 get_uid $1
19
20 # La variable UID se inicializa con el resultado (=salida)
21 # de la función (=comando)  get_uid
22
23 UID=$(get_uid $1)
24 if [[ $UID != "" ]] ; then
25     echo "El usuario $1 tiene por uid: $UID"
26 else
27     echo "El usuario $1 no existe"
28 fi
29
30 $ get_uid.sh cristina
31 2025
32 El usuario cristina tiene por uid: 2025
33 $
```

La línea 14 muestra que la función **get\_uid** imprime el uid del usuario del que ha recibido el nombre por parámetro (no escribe nada si el usuario no existe). Esta línea se ha puesto por necesidades de la demostración; en un caso real, es poco útil.

En la línea 17, la función se llama de nuevo. Mediante el mecanismo de sustitución de comandos, el shell reemplaza esta llamada por el valor mostrado por la función. Este valor se asigna a continuación en la variable global UID.

## 7. Programa completo del ejemplo

Aquí se muestra el script **gestusuario.sh**, que agrupa las funciones **pausa**, **existe\_usuario** y **get\_uid**:

```
$ nl gestusuario.sh
 1  #!/usr/bin/ksh
 2
 3  # Provocar una pausa por pantalla
 4  function pausa {
 5      echo "Pulse una tecla para continuar"
 6      read x
 7  }
 8
 9  # Saber si un usuario existe en el sistema
10  function existe_usuario {
11      grep -qi "^$1:" /etc/passwd && return 0
12      return 1
13  }
14
15  # Encontrar el uid de un usuario
16  function get_uid
17  {
18      grep -i "^$1:" /etc/passwd | cut -d':' -f3
19  }
20
21  # -----
22  # Programa principal
23  # -----
24
25  Uid=""
26  Usuario=""
27  Eleccion=""
28  while true
29  do
30      clear
31      echo "- 1 - Saber si un usuario existe en el sistema"
32      echo "- 2 - Conocer el uid de un usuario"
33      echo "- 3 - Fin"
34
35      echo "Su elección: \c"
36      read Eleccion
37
38      if [[ $Eleccion = @(1|2) ]] ; then
39          echo "Introduzca el nombre de un usuario: \c"
40          read Usuario
41      fi
42
43      case $Eleccion in
44
45          1) # Llamada a la función
46             # Comprobación del código de retorno de la función ($)
47             if existe_usuario $Usuario ; then
48                 echo "El usuario $Usuario existe en el sistema"
49             else
50                 echo "El usuario $Usuario no existe en el sistema"
51             fi
52             ;;
53
54          2)
55             if existe_usuario $Usuario ; then
56                 Uid=$(get_uid $Usuario)
57                 echo "El UID del usuario $Usuario es: $Uid"
58             else
59                 echo "El usuario $Usuario no existe"
60             fi
61             ;;
62
63          3) exit 0 ;;
64
65      esac
66  done
```

```
56
57     esac
58     pausa          # Llamada a la función pausa
59 done
```

```
$
```

# Comandos de salida

## 1. El comando print

```
ksh
```

Este comando aporta funcionalidades que no existen con **echo**.

### a. Uso simple

#### Ejemplo

```
$ print Error de impresión
Error de impresión
$
```

### b. Supresión del salto de línea natural de print

Hay que usar la opción **-n**.

#### Ejemplo


```
$ print -n Error de impresión
Error de impresión$
```

### c. Mostrar argumentos que comienzan por el carácter "-"

#### Ejemplo

En el ejemplo siguiente, la cadena de caracteres **-i** forma parte del mensaje. Por desgracia, **print** interpreta **-i** como una opción y no como un argumento:

```
$ print -i: Opción inválida
ksh: print: bad option(s)
$ print "-i: Opción inválida"
ksh: print: bad option(s)
```

 Es inútil poner protecciones alrededor de los argumentos de **print**. En efecto, **-** no es un carácter especial de shell; por tanto, no sirve protegerlo. No se interpreta por el shell, sino por el comando **print**.

Con la opción **-** del comando **print**, los caracteres siguientes se interpretarán como argumentos, sea cual sea su valor.

#### Ejemplo

```
$ print - "-i: Opción inválida"
-i: Opción inválida
$
```

### d. Escritura hacia un descriptor determinado

La opción **-u** permite enviar un mensaje hacia un descriptor determinado.

```
print -udesc mensaje
```

donde `desc` represente el descriptor de archivo.

### Ejemplo

Enviar un mensaje hacia la salida de error estándar con `print`:

```
$ print -u2 "Mensaje de error"
```

Enviar un mensaje hacia la salida de error estándar con `echo`:

```
$ echo "Mensaje de error" 1>&2
```

Comparación de ambos comandos:

- La opción `-u2` del comando **print** le indica que debe enviar el mensaje hacia la salida de error estándar.
- El comando **echo** escribe siempre hacia su descriptor 1. Por tanto, será necesario que la salida estándar haya sido redirigida hacia la salida de error estándar (`1>&2`) antes de que el comando **echo** se ejecute.

## 2. El comando `printf`

```
bash
```

Este comando retoma la función `printf` del lenguaje C. Permite formatear las impresiones. Como el comando es interno, no está disponible en todos los shells. Sin embargo, puede ofrecerse bajo la forma de comando externo (`/usr/bin/printf`).

```
printf cadena expr1 expr2 ... exprn
```

`cadena` representa la cadena que será impresa por pantalla. Puede contener los formatos que se sustituirán por el valor de las expresiones citadas a continuación. Tiene que haber el mismo número de formatos que de expresiones.

### Ejemplos de formatos de uso común

<code>%20s</code>	Muestra una cadena (string) de 20 posiciones (alineada a la derecha por defecto).
<code>%-20s</code>	Muestra una cadena (string) de 20 posiciones con alineación a la izquierda.
<code>%3d</code>	Muestra un entero (decimal) de 3 posiciones (alineado a la derecha).
<code>%03d</code>	Muestra un entero (decimal) de 3 posiciones (alineado a la derecha) completado por 0 a la izquierda.
<code>%-3d</code>	Muestra un entero (decimal) de 3 posiciones (alineado a la izquierda).
<code> %+3d</code>	Muestra un entero (decimal) de 3 posiciones (alineado a la derecha) con impresión sistemática de su signo (un número negativo siempre se muestra con su signo).
<code>%10.2f</code>	Muestra un número en coma flotante de 10 posiciones, de las cuales 2 son decimales.
<code> %+010.2f</code>	Muestra un número en coma flotante de 10 posiciones, de las cuales 2

son decimales, completado por 0 a la izquierda, alineado a la derecha e impresión sistemática del signo.

Ejemplo

```
$ articulo="Suministros" ; cantidad=3 ; precio=45.2
$ printf "%-20s***%03d***%+10.2f\n" $articulo $cantidad $precio
Suministros      ***003***      +45.20
$
```

# Gestión de entradas/salidas de un script

## 1. Redirección de entradas/salidas estándar

El comando interno **exec** permite manipular los descriptores de archivo del shell en ejecución. Usado en el interior de un script, permite redirigir de manera global las entradas/salidas de este.

### Redirigir la entrada estándar de un script

```
exec 0< archivo 1
```

Todos los comandos del script situados después de esta directiva y que leen de su entrada estándar extraerán sus datos desde **archivo1**. Por tanto, no habrá más interacción con el teclado.

### Redirigir la salida estándar y la salida de error estándar de un script

```
exec 1> archivo1 2> archivo2
```

Todos los comandos del script situados después de esta directiva y que escriben en su salida estándar enviarán sus resultados a **archivo1**. Los que escriban en su salida de error estándar enviarán sus errores a **archivo2**.

### Redirigir la salida estándar y la salida de error estándar de un script al mismo archivo

```
exec 1> archivo1 2>&1
```

Todos los comandos del script situados después de esta directiva enviarán sus resultados y sus errores a **archivo1** (ver capítulo Mecanismos esenciales de shell - Redirecciones).

#### Primer ejemplo

El script **batch1.sh** envía su salida estándar a **/tmp/resu** y su salida de error estándar a **/tmp/log**:

```
$ nl batch1.sh
1  #! /usr/bin/ksh
2
3  exec 1> /tmp/resu 2> /tmp/log
4  print "Inicio del tratamiento: $(date)"
5  ls
6  cp *.c /tmp
7  rm *.c
8  sleep 2
9  print "Fin del tratamiento: $(date)"
$
```

No hay archivos que terminen por ".c" en el directorio actual:

Ejecución del script:

Contenido del archivo **/tmp/resu**:

Contenido del archivo **/tmp/log**:

```
$ ls
Shell          resu.log.140303_1008
```

```
$ batch1.sh
```

```
$ nl /tmp/resu
1  Inicio del tratamiento: Thu Mar 13 20:04:47 MET 2014
2  Shell
3  resu.log.140303_1008
4  Fin del tratamiento: Thu Mar 13 20:04:49 MET 2014
```

```
$ nl /tmp/log
1  cp: cannot access *.c 7
2  *.c: No such file or directory
```

➤ No es práctico tener los mensajes de resultado y los mensajes de error en archivos distintos, ya que la cronología de los eventos resulta difícil de reconstruir.

#### Segundo ejemplo

El script **batch2.sh** envía su salida estándar y su salida de error estándar a **/tmp/log**:

```
$ nl batch2.sh
1  #! /usr/bin/ksh
2
3  exec 1> /tmp/log 2>&1
4  print "Inicio del tratamiento: $(date)"
5  ls
6  cp *.c /tmp
7  rm *.c
8  sleep 2
9  print "Fin del tratamiento: $(date)"
```

Ejecución del script:

Contenido del archivo **/tmp/log**:

```
$ batch2.sh
```

El archivo **/tmp/log** agrupa todas las impresiones

```
$ nl /tmp/log
1 Inicio del tratamiento: Thu Mar 13 20:12:31 MET 2014
2 Shell
3 resu.log.140303_1008
4 cp: cannot access *.c
5 *.c: No such file or directory
6 Fin del tratamiento: Thu Mar 13 20:12:33 MET 2014
```

### Tercer ejemplo

A continuación se muestra el archivo de datos **tel.txt** que será explotado por el script **lectura.sh**:



```
nl tel.txt
1 Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915351478
2 Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|
932282177
3 Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de
Calatrava|926443602
4 Expósito Heredia, Pedro|calle del castillo|38870|La
Calera|984122369
```

El script **lectura.sh** lee las dos primeras líneas del archivo:

Cada llamada al comando read (líneas 4 y 7) provoca la lectura de una línea del archivo **tel.txt**, ya que la entrada estándar del script se ha conectado a este archivo (línea 1).

Resultado de la ejecución:

```
$ nl lectura.sh
1 exec 0<tel.txt

2 # El comando read lee su entrada estándar
3 echo "Lectura de la 1ª línea"
4 read línea
5 echo $línea
6 echo "Lectura de la 2ª línea"
7 read línea
8 echo $línea
```

```
$ lectura.sh
Lectura de la 1ª línea
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915351478
Lectura de la 2ª línea
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|932282177
```

## 2. Gestión de archivos

ksh	bash
-----	------

Los shells recientes ofrecen funcionalidades adicionales a las descritas anteriormente. Permiten abrir y manipular archivos usando

descriptores comprendidos entre 3 y 9 (además de 0, 1 y 2). La ventaja es poder manipular archivos conservando los descriptores 0, 1 y 2 conectados al terminal.

### a. Apertura de archivo

#### En lectura

```
exec desc<archivo
```

#### En escritura

```
exec desc>archivo
```

### b. Lectura a partir de un archivo

```
read variable1 variable2 ... variablen <&desc
```

o

```
read -udesc variable1 variable2 ... variablen
```

### c. Escritura en un archivo

```
echo variable1 variable2 ... variablen >&desc
```

o

ksh
-----

```
print -udesc variable1 variable2 ... variablen
```

### d. Cierre de un archivo

#### Sintaxis

```
exec desc<&-
exec desc>&-
```

#### Ejemplo

El script **lectura2.sh** lee las dos primeras líneas del archivo **tel.txt**, las muestra por pantalla y las escribe en el archivo **out.txt**:

```
$ nl lectura2.sh
1 #! /usr/bin/ksh
```

Ejecución del script:

Contenido del archivo **out.txt**:



```

2 # Apertura de los archivos
3 exec 3<tel.txt 4>out.txt

4 # Lectura de la 1ª línea de tel.txt
5 read -u3 línea
6 # Escritura de la 1ª línea por pantalla y en out.txt
7 print $línea
8 print -u4 $línea

9 # Lectura de la 2ª línea de tel.txt
10 read -u3 línea

11 # Escritura de la 2ª línea por pantalla y en out.txt
12 print $línea
13 print -u4 $línea

14 # Cierre de los archivos
15 exec 3<&-
16 exec 4>&-

```

```

$ lectura2.sh
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915351478
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|932282177

```

```

$ nl out.txt
1 Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915351478
2 Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|
932282177
$

```

### 3. Tratamiento de un archivo

#### a. Información previa

##### Recordatorio de las características de read

La lectura de una línea de un archivo se realiza con el comando **read**. A continuación, un recordatorio de las características de este comando:

- Lee una línea de la entrada estándar (descriptor 0).
- Devuelve un código verdadero si ha leído una línea (incluso si la línea está vacía).
- Devuelve un código falso cuando encuentra el final del archivo.

La lectura de todas las líneas de un archivo se realiza asociando el comando **read** y el bucle **while**.

##### El concepto de final de archivo

Cuando la entrada estándar (descriptor 0) de **read** está conectada al terminal (caso por defecto), el concepto de final de archivo tiene que ser enviado por el usuario mediante las teclas **^d**.

Si la entrada estándar del comando está conectada a un archivo regular, el núcleo de Unix previene a **read** cuando se llega al final del archivo. Para el desarrollador, solamente el test de código de retorno de **read** indica que el comando ha llegado al final de archivo.

#### b. Las diferentes formas de explotar un archivo

##### Redirigir la ejecución del script

Las redirecciones pueden realizarse en el momento de invocar el script.

##### Ejemplos

El script **leearch1.sh** usa los comandos **read** y **print** y no realiza ninguna redirección. El archivo de entrada es el teclado y el archivo de salida es el terminal:

```

$ nl leearch1.sh
1 #!/usr/bin/ksh

2 # A cada iteración del bucle, se lee una línea
3 numero=0

4 while read línea
5 do
6     ((numero+=1))
7     print "Línea nº $numero => $línea" # Usar echo en bash
8 done
9 print "Fin de archivo"

$ leearch1.sh
1ª línea # entrada de datos
Línea nº 1 => 1ª línea
2ª línea # entrada de datos
Línea nº 2 => 2ª línea
3ª línea # entrada de datos
Línea nº 3 => 3ª línea
^d # entrada de datos
Fin de archivo

```

A continuación se muestra una versión de este script compatible con Bourne Shell:

La entrada y la salida estándar del shell (lo que interpreta el script) se asocian respectivamente a los archivos **tel.txt** y **out.txt** antes de que el script **leearch1.sh** comience a ejecutarse. Todos los comandos del script que leen de su entrada estándar (en este caso, **read**) extraen sus datos del archivo **tel.txt**; todos los que escriben en su salida estándar (en este caso, **print**) envían sus impresiones al archivo **out.txt**:

##### Redirecciones internas al script

Las redirecciones de la entrada (0) y de salida (1) estándares pueden realizarse dentro del script.

##### Ejemplo

Las redirecciones de la entrada y de la salida

estándares se realizan dentro del script **leearch2.sh** utilizando el comando **exec**:

```
$ nl leearch1_sh.sh
1  #!/usr/bin/sh

2  numero=0

3  while read linea
4  do
5      numero=`expr $numero + 1`
6      echo "Línea n° $numero => $linea"
7  done
8  echo "Fin de archivo"

$
```

A continuación se muestra una versión de este script compatible con Bourne Shell:

### Uso de otros descriptores

En ksh y bash, es posible usar descriptores de archivo comprendidos entre 3 y 9, lo que permite mantener asociados los descriptores 0, 1 y 2 del terminal.

#### Ejemplo

```
$ leearch1.sh <tel.txt >out.txt
$ nl out.txt
1  Línea n° 1 => Méndez Roca, Gisela|calle Ruiseñor|28023|
Madrid|915351478
2  Línea n° 2 => Ruiz del Castillo, Marcos|calle Balmes|08020|
Barcelona|932282177
3  Línea n° 3 => Gómez Bádenas, Josefina|calle Sagasta|13190|
Corral de Calatrava|926443602
4  Línea n° 4 => Expósito Heredia, Pedro|calle del castillo|
38870|La Calera|984122369
5  Fin de archivo

$
```

En el script **leearch3.sh**, el archivo **tel.txt** se abre en modo lectura con el descriptor 3 y el archivo **out.txt** se abre en modo escritura con el descriptor 4. Los comandos **read** y **print** utilizan la opción **-u** para especificar el descriptor utilizado:

A continuación se muestra una versión compatible con bash:

### Redirección de un bloque

Es posible redirigir únicamente los comandos situados en el interior de una estructura de control (un bloque). Las redirecciones tienen que escribirse detrás de la palabra clave que cierra la estructura de control. En la ejecución, se realizan **antes** del tratamiento de la estructura de control.

#### Ejemplo

En el script **leearch4.sh**, las redirecciones solo conciernen a los comandos internos al bucle **while**:

Redirigir un bloque a los archivos abiertos más arriba:

```
$ nl leearch2.sh
1  #!/usr/bin/ksh

2  exec <tel.txt >out.txt

3  numero=0
4  while read linea
5  do
6      ((numero+=1))
7      print "Línea n° $numero => $linea"
8  done
9  print "Fin de archivo"

$ leearch2.sh
$ nl out.txt
1  Línea n° 1 => Méndez Roca, Gisela|calle Ruiseñor|28023|
Madrid|915351478
2  Línea n° 2 => Ruiz del Castillo, Marcos|calle Balmes|08020|
Barcelona|932282177
3  Línea n° 3 => Gómez Bádenas, Josefina|calle Sagasta|13190|
Corral de Calatrava|926443602
4  Línea n° 4 => Expósito Heredia, Pedro|calle del castillo|
38870|La Calera|984122369
5  Fin de archivo

$
```

```
$ nl leearch2_sh.sh
1  #!/usr/bin/sh

2  exec <tel.txt >out.txt

3  numero=0
4  while read linea
5  do
6      numero=`expr $numero + 1`
7      echo "Línea n° $numero => $linea"
8  done
9  echo "Fin de archivo"
```

ksh

```
$ nl leearch3.sh
1  #!/usr/bin/ksh

2  exec 3<tel.txt 4>out.txt

3  numero=0
4  while read -u3 linea
5  do
6      ((numero+=1))
7      print -u4 "Línea n° $numero => $linea"
8  done
9  print -u4 "Fin de archivo"

10 exec 3<&-
11 exec 4>&-

$
$ leearch3.sh
$ nl out.txt
1  Línea n° 1 => Méndez Roca, Gisela|calle Ruiseñor|28023|
Madrid|915351478
2  Línea n° 2 => Ruiz del Castillo, Marcos|calle Balmes|08020|
```

```
Barcelona|932282177
3 Línea nº 3 => Gómez Bádenas, Josefina|calle Sagasta|13190|
Corral de Calatrava|926443602
4 Línea nº 4 => Expósito Heredia, Pedro|calle del castillo|
38870|La Calera|984122369
5 Fin de archivo
```

ksh	bash
-----	------

```
$ nl leearch3_bash.sh
1  #! /bin/bash

2  exec 3<tel.txt 4>out.txt

3  numero=0
4  while read linea <&3
5  do
6      ((numero+=1))
7      echo "Línea nº $numero => $linea" >&4
8  done
9  echo "Fin de archivo" >&4
10 exec 3<&-
11 exec 4>&-

$
$ leearch3.sh
$ nl out.txt
1  Línea nº 1 => Méndez Roca, Gisela|calle Ruiseñor|28023|
Madrid|915351478
2  Línea nº 2 => Ruiz del Castillo, Marcos|calle Balmes|08020|
Barcelona|932282177
3  Línea nº 3 => Gómez Bádenas, Josefina|calle Sagasta|13190|
Corral de Calatrava|926443602
4  Línea nº 4 => Expósito Heredia, Pedro|calle del castillo|
38870|La Calera|984122369
5  Fin de archivo
$
```

```
$ nl leearch4.sh
1  #! /usr/bin/ksh

2  numero=0
3  print "Antes de while" # Usar echo en bash

4  while read linea
5  do
6      ((numero+=1))
7      print "Línea nº $numero => $linea"
8  done <tel.txt >out.txt
9  print "Fin de archivo"

$ leearch4.sh
Antes de while
Fin de archivo
$ nl out.txt
1  Línea nº 1 => Méndez Roca, Gisela|calle Ruiseñor|28023|
Madrid|915351478
2  Línea nº 2 => Ruiz del Castillo, Marcos|calle Balmes|08020|
Barcelona|932282177
3  Línea nº 3 => Gómez Bádenas, Josefina|calle Sagasta|13190|
Corral de Calatrava|926443602
4  Línea nº 4 => Expósito Heredia, Pedro|calle del castillo|
38870|La Calera|984122369
$
```

```
$ nl leearch5.sh
1  #! /usr/bin/ksh

2  exec 3<tel.txt 4>out.txt

3  numero=0
4  echo "Antes de while"

5  while read linea
6  do
7      ((numero+=1))
8      echo "Línea nº $numero => $linea"
9  done <&3 >&4
10 echo "Fin de archivo"

11 exec 3<&-
12 exec 4>&-

$ leearch5.sh
Antes de while
Fin de archivo
$ nl out.txt
1  Línea nº 1 => Méndez Roca, Gisela|calle Ruiseñor|28023|
Madrid|915351478
2  Línea nº 2 => Ruiz del Castillo, Marcos|calle Balmes|08020|
Barcelona|932282177
3  Línea nº 3 => Gómez Bádenas, Josefina|calle Sagasta|13190|
Corral de Calatrava|926443602
4  Línea nº 4 => Expósito Heredia, Pedro|calle del castillo|
```

### c. Repartir una línea en campos

Si las líneas de un archivo que se tiene que tratar están estructuradas en campos, es muy fácil recuperar cada uno en una variable. Para ello, hay que modificar el valor de la variable IFS (ver capítulo Las bases de la programación shell - El comando read).

#### Ejemplo

El script **leearch6.sh** genera, a partir del archivo **tel.txt**, una salida por pantalla que toma el formato del archivo tratado, pero añadiendo "(+0)" delante de los teléfonos de los clientes localizados en la península y Baleares y "(-1)" delante de los teléfonos de los clientes localizados en las Islas Canarias (a modo de recordatorio de la zona horaria respecto la de la capital). La variable IFS se inicializa con el valor del carácter que hace de separador de campos en el archivo (en este caso "|"):

```
$ nl leearch6.sh
1  #!/usr/bin/ksh
2  if (( $# != 1 ))
3  then
4    echo "Número de argumentos incorrecto"
5    echo "Uso: $0 archivo"
6    exit 1
7  fi
8  if [[ ( ! -f "$1" ) || ( ! -r "$1" ) ]]
9  then
10   echo "$1 no es un archivo regular o no es
accesible en modo lectura"
11   exit 2
12  fi
13  IFS="|"
14  while read nom dir cp ciudad tel
15  do
16    case "$cp" in
17      3[58]* )
18        horario="(-1)"
19        ;;
20      *)
21        horario="(+0)"
22        ;;
23    esac
24    echo "$nom|$dir|$cp|$ciudad|$horario$tel"
25  done < $1
$
```

El comando **read** recibe cinco nombres de variables como argumento (línea 14), de modo que reparte la línea leída en campos usando el carácter "|" como separador (modificación de la variable IFS en la línea 13). La línea se divide, por tanto, automáticamente. Tan solo queda pendiente comprobar el valor del código postal (línea 16) y volver a unir la línea insertando la diferencia horaria correspondiente al cliente justo delante de su teléfono (línea 24).

Ejecución del script:

```
$ leearch6.sh tel.txt
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|(+0)915351478
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|
(+0)932282177
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de
Calatrava|(+0)926443602
Expósito Heredia, Pedro|calle del castillo|38870|La Calera|
(-1)984122369
$
```

### d. Modificar el separador de línea

```
ksh93  bash
```

La opción **-d** del comando **read** permite modificar el carácter separador de línea (por defecto `\n`).

#### Ejemplo

En el archivo **config.txt**, el carácter **;** es el separador de parámetros:

```
$ cat config.txt
param1:
    option 1.1
    option 1.2
    ;
param2:
    option 2.1
    option 2.2
    ;
```

Modificación del carácter separador de línea:

Ejecución:

```
$ nl readMulti.sh
1  #!/bin/bash
2  typeset -i cpt=1
3  while read -d ' '; linea
4  do
5    echo "Línea $cpt leída : $linea"
6    (( cpt++ ))
7  done < config.txt
```

```
$ readMulti.sh
Línea 1 leída : param1:
```

```
option 1.1
option 1.2
Linea 2 leida : param2
option 2.1
option 2.2
```

# El comando eval

## Sintaxis

```
eval expr1 expr2 ... exprn
```

El comando **eval** permite la realización de una doble evaluación en la línea de comandos. Recibe como argumento un conjunto de expresiones en el que efectúa las operaciones siguientes:

- Primera etapa: los caracteres especiales contenidos en las expresiones se tratan. El resultado del tratamiento genera una o varias expresiones: `eval otra_exp1 otra_exp2 ... otra_expn`. La expresión `otra_exp1` representará el comando Unix que se debe ejecutar en la segunda etapa.
- Segunda etapa: **eval** va a ejecutar el comando `otra_exp1 otra_exp2 ... otra_expn`. Sin embargo, previamente, esta línea se va a someter a una nueva evaluación. Los caracteres especiales se tratan y después el comando se lanza.

## Ejemplo

Definición de la variable **nombre** que contiene "cristina":

```
$ nombre=cristina
```

Definición de la variable **var** que contiene el nombre de la variable definida justo arriba:

```
$ var=nombre
```

¿Cómo imprimir por pantalla el valor "cristina" sirviéndose de la variable **var**? En el comando siguiente, el shell sustituye \$\$ por el PID del shell actual:

```
$ echo $$var  
17689var
```

En el comando siguiente, el nombre de la variable está aislado. Este no podrá funcionar: el shell genera un error de sintaxis, ya que no puede tratar dos caracteres "\$" simultáneamente:

```
$ echo ${$var}  
ksh: ${$var}: bad substitution
```

Es indispensable usar el comando **eval**:

```
$ eval echo \$$var  
cristina  
$
```

- El orden de evaluación del shell se explica en el capítulo Las bases de la programación shell - Interpretación de una línea de comandos.

## Mecanismo del comando eval

Primera etapa: evaluación de los caracteres especiales

```
eval echo \$$var
```



Tratamiento de los caracteres de protección (la barra invertida es obligatoria para evitar el tratamiento del primer "\$" en la primera etapa)



```
eval echo $ $var
```



Sustitución de variables



```
eval echo $nombre
```



Segunda etapa: evaluación y ejecución del comando **echo \$nombre**

```
echo $nombre
```



Sustitución de variables



```
echo cristina
```

Ejecución del comando **echo** que recibe la palabra "cristina" como argumento.

# Gestión de señales

El comportamiento del shell actual respecto a las señales puede modificarse utilizando el comando `trap`.

## 1. Señales principales

Nombre de la señal	Nº	Significado	Comportamiento por defecto de un proceso ante la recepción de la señal	¿Disposición modificable?
HUP	1	Ruptura de una línea de terminal. Durante una desconexión, la señal se recibe por cualquier proceso ejecutado en segundo plano desde el shell en cuestión.	Morir	sí
INT	2	Generado desde el teclado (ver parámetro <code>intr</code> del comando <code>stty -a</code> ). Usado para matar el proceso que corre en primer plano.	Morir	sí
TERM	15	Generado vía el comando <code>kill</code> . Usado para matar un proceso.	Morir	sí
KILL	9	Generado vía el comando <code>kill</code> . Usado para matar un proceso.	Morir	no

➤ En los comandos, las señales pueden ser expresadas en forma numérica o simbólica. Las señales HUP, INT, TERM y KILL poseen el mismo valor numérico en todas las plataformas Unix, particularidad que no cumplen todas las señales. Por tanto, se aconseja usar la forma simbólica.

## 2. Ignorar una señal

### Sintaxis

```
trap '' sig1 sig2
```

### Ejemplo

El shell actual tiene el PID 18033:

```
$ echo $$  
18033
```

El usuario solicita al shell ignorar la posible recepción de las señales HUP y TERM:



```
$ trap '' HUP TERM
```

Envío de las señales HUP y TERM:

```
$ kill -HUP 18033
$ kill -TERM 18033
```

Las señales se ignoran; por tanto, el proceso shell no muere:

```
$ echo $$
18033
$
```

### 3. Modificar el comportamiento asociado a una señal

#### Sintaxis

```
trap 'cmd1;cmd2; ...; cmdn' sig1 sig2
```

#### Ejemplo

El shell actual tiene el PID 18007 y ha sido ejecutado a partir del shell de PID 17924:

```
$ ps
  PID TTY          TIME CMD
 18007 pts/2        0:00 ksh
 17924 pts/2        0:00 ksh
$ echo $$
18007
```

Modificación del comportamiento del shell actual respecto a SIGINT:

```
$ trap 'echo "Señal INT recibida" ; exit 1' INT
```

Se tedeala ^c:

```
$ ^c
```

El shell ejecuta el comando **echo** y después el comando **exit**:

```
Señal INT recibida
```

El usuario es devuelto al shell anterior, cuyo PID es 17924:

```
$ ps
  PID TTY          TIME CMD
 17924 pts/2        0:00 ksh
$ echo $$
17924
```

### 4. Restablecer el comportamiento por defecto del shell respecto a una señal

#### Sintaxis

```
trap - sig1 sig2 ... sign
```

### Ejemplo

Creación del archivo **/tmp/arch**:

```
$ > /tmp/arch
```

Si el shell recibe SIGINT o SIGTERM, tiene que borrar el archivo:

```
$ trap 'rm -f /tmp/arch' INT TERM
```

PID del shell actual:

```
$ echo $$  
18033
```

Envío de la señal SIGINT al shell actual:

```
$ ^c
```

El archivo se ha borrado:

```
$ ls /tmp/arch  
/tmp/arch: No such file or directory
```

Creación de nuevo del archivo **/tmp/arch**:

```
$ > /tmp/arch
```

Restablecimiento del comportamiento por defecto del shell (de forma predeterminada, un shell interactivo ignora SIGINT) respecto a la señal SIGINT.

```
$ trap - INT TERM
```

PID del shell actual:

```
$ echo $$  
18033
```

Envío de la señal SIGINT al shell actual:

```
$ ^c
```

El shell no realiza ningun tratamiento y el archivo sigue estando presente:

```
$ ls /tmp/arch  
/tmp/arch  
$
```

## 5. Usar trap desde un script de shell

El uso de trap desde un script de shell permite prevenir el tratamiento que se ha de realizar ante la posible recepción de una o múltiples señales.

### Ejemplo

El script **captura\_señal.sh** recibe una señal que puede ser **SIGHUP**, **SIGINT** o **SIGTERM**, llama la función "finalizar", que elimina el archivo **\$ARCHTEMP** y finaliza el script:

```
$ nl captura_señal.sh
 1  #!/bin/bash
 2  ARCHTEMP=/tmp/temp.txt
 3  function finalizar {
 4      echo Borrado de $ARCHTEMP - Fin del script
 5      rm -f $ARCHTEMP
 6      exit 1
 7  }
 8  # Asociación de SIGHUP, SIGINT y SIGTERM
 9  # Llamar a la función finalizar si se recibe una señal
10  trap finalizar HUP INT TERM
11  # Creación de ARCHTEMP
12  > $ARCHTEMP
13  # Tratamiento
14  echo "Comienzo del tratamiento"
15  sleep 100
16  echo "Fin del tratamiento"
17  exit 0
```

*Ejecución:*

```
$ captura_señal.sh
Comienzo del tratamiento
^c
Borrado de /tmp/temp.txt - Fin del script
$ echo $?
1
$
```

# Gestión de menús con select

ksh	bash
-----	------

## Sintaxis

```
select var in item1 item2 ... itemn
do
    comandos
done
```

El comando interno **select** es una estructura de control de tipo bucle que permite escribir de manera cíclica un menú. La lista de items, `item1 item2 ... itemn`, se mostrará por pantalla a cada iteración del bucle. Los ítems son indexados automáticamente. La variable `var` se inicializará con el ítem correspondiente a elección del usuario.

Este comando usa también dos variables reservadas:

- La variable `PS3` representa el prompt utilizado para que el usuario teclee su elección. Su valor por defecto es `#!`. Se puede modificar a gusto del programador.
- La variable `REPLY` contiene el índice del ítem seleccionado.



La variable `var` contiene la etiqueta de la elección, y `REPLY`, el índice de esta.

## Ejemplo

```
$ nl menuselect.sh
 1  #! /usr/bin/ksh
 2  function guardar {
 3      echo "Se ha escogido la copia de seguridad"
 4      # Ejecución de la copia de seguridad
 5  }
 6
 7  function restaurar {
 8      echo "Se ha escogido la restauración"
 9      # Ejecución de la restauración
10  }
11
12  PS3="Su elección: "
13  select item in "- Copia de seguridad " "- Restauración " "-
Fin"
14  do
15      echo "Se ha escogido el ítem $REPLY: $item"
16      case "$REPLY" in
17          1)
18              # Llamada a la función de guardar
19              guardar
20              ;;
21          2)
22              # Llamada a la función de restaurar
23              restaurar
24              ;;
25          3) echo "Fin del tratamiento"
26              echo "Saludos ..."
27              exit 0
28              ;;
29      esac
```

```
30     *) echo "Entrada incorrecta"
32     ;;
33     esac
34 done
```

### *Ejecución:*

```
$ menu_select.sh
1) - Copia de seguridad
2) - Restauración
3) - Fin
Su elección: 1
Se ha escogido el ítem 1: - Copia de seguridad
Se ha escogido la copia de seguridad
```

*La introducción de la tecla [Entrar] permite reimprimir por pantalla el menú:*

```
Su elección: [Entrar]
1) - Copia de seguridad
2) - Restauración
3) - Fin
Su elección: 3
Se ha escogido el ítem 3: - Fin
Fin del tratamiento
Saludos ...
```

# Análisis de las opciones de un script con getopts

## Sintaxis

```
getopts lista-opciones-esperadas opción
```

El comando interno **getopts** permite a un script analizar las opciones que le han sido pasadas como argumento. Cada llamada a **getopts** analiza la opción siguiente de la línea de comandos. Para verificar la validez de cada una de las opciones, hay que llamar a **getopts** desde un bucle.

## Definición de una opción

Para **getopts**, una opción se compone de un carácter precedido por un signo "+" o "-".

### Ejemplo

"-c" y "+c" son opciones, mientras que "cristina" es un argumento:

```
# gestusuario.sh -c cristina
# gestusuario.sh +c
```

Una opción puede funcionar sola o estar asociada a un argumento.

### Ejemplo

A continuación se muestra el script **gestusuario.sh**, que permite archivar y restaurar cuentas de usuario. Las opciones **-c** y **-x** significan respectivamente "Crear un archivo" y "Extraer un archivo". Estas son opciones sin argumento. Las opciones **-u** y **-g** permiten especificar la lista de usuarios y la lista de grupos que se han de tratar. Tienen que estar seguidas de un argumento.

```
# gestusuario.sh -c -u cristina,roberto,olivia
# gestusuario.sh -x -g curso -u cristina,roberto
```

Para comprobar si las opciones y los argumentos pasados al script **gestusuario.sh** son los esperados, el programador escribirá:

```
getopts "cxu:g:" opcion
```

Explicación de los argumentos de **getopts**:

- Primer argumento: las opciones se citan una tras otra. Una opción seguida de ":" significa que se trata de una opción con argumento.
- Segundo argumento: `opcion` es una variable de usuario que será inicializada con la opción en curso del tratamiento.

Una llamada a **getopts** recupera la opción siguiente y devuelve verdadero mientras queden opciones para analizar. Cuando una opción tiene asociado un argumento, este se deposita en la variable reservada `OPTARG`.

La variable reservada `OPTIND` contiene el índice de la siguiente opción que se ha de tratar.

```
$ nl gestusuario.sh
1  while getopts "cxu:g:" opcion
2  do
3    echo "getopts ha encontrado la opción $opcion"
4    case "$opcion" in
5      c) echo "Archivado"
6         echo "Índice de la siguiente opción a tratar: $OPTARG"
7         ;;
8      x) echo "Extracción"
```

```

 9      echo "Índice de la siguiente opción a tratar: $OPTIND"
10      ;;
11      u) echo "Lista de usuarios que se tratarán: $OPTARG"
12      echo "Índice de la siguiente opción a tratar: $OPTIND"
13      ;;
14      g) echo "Lista de grupos que se tratarán: $OPTARG"
15      echo "Índice de la siguiente opción a tratar: $OPTIND"
16      ;;
17  esac
18  done
19  echo "Análisis de opciones terminado"
20  exit 0

```

*Llamada al script con opciones válidas:*

```

$ gestusuario.sh -c -u cristina,roberto,olivia -g curso
getopts ha encontrado la opción c
Archivado
Índice de la siguiente opción a tratar: 2
getopts ha encontrado la opción u
Lista de usuarios a tratar: cristina,roberto,olivia
Índice de la siguiente opción a tratar: 4
getopts ha encontrado la opción g
Lista de grupos a tratar: curso
Índice de la siguiente opción a tratar: 6
Análisis de opciones terminado

```

## Opción inválida

Cuando **getopts** detecta una opción inválida, la variable `opcion` se inicializa con el carácter "?" y se muestra un mensaje de error por pantalla. Después, las opciones siguientes se analizarán.

### Ejemplo

*La opción `-y` no forma parte de la lista de opciones esperadas:*

```

$ gestusuario.sh -y -c
gestusuario.sh: getopts: y bad option(s)
getopts ha encontrado la opción ?
getopts ha encontrado la opción c
Archivado
Índice de la siguiente opción a tratar: 3
Análisis de opciones terminado
$

```

## Gestión de errores

Si el carácter ":" se coloca en la **primera posición** en la lista de opciones que se han de tratar (línea 3), los errores se generan de forma diferente. En el caso de una opción inválida:

- `getopts` no mostrará un mensaje de error.
- la variable `OPTARG` se inicializará con el valor de la opción incorrecta (línea 18).

### Ejemplo

*El código puede ser rescrito de la siguiente manera:*

```

$ nl gestusuario.sh
 1 # El carácter ":" en 1ª posición permite gestionar mejor

```

```

2 # las opciones incorrectas
3 while getopt " :cxu:g:" opcion
4 do
5     case "$opcion" in
6         c) echo "Archivado"
7             echo "Índice de la siguiente opción a tratar: $OPTIND"
8             ;;
9         x) echo "Extracción"
10            echo "Índice de la siguiente opción a tratar: $OPTIND"
11            ;;
12        u) echo "Lista de usuarios que se tratarán: $OPTARG"
13            echo "Índice de la siguiente opción a tratar: $OPTIND"
14            ;;
15        g) echo "Lista de grupos que se tratarán: $OPTARG"
16            echo "Índice de la siguiente opción a tratar: $OPTIND"
17            ;;
18        \?) echo "$OPTARG: OPCIÓN INVÁLIDA - Adiós"
19            exit 1
20            ;;
21        esac
22    done
23    echo "Análisis de opciones terminado"
24    exit 0

```


\$

El mensaje de error generado automáticamente por `getopts` no aparece y `$OPTARG` ha sido sustituido por el valor de la opción incorrecto:

```

$ gestusuario.sh -y -c
y: OPCIÓN INVÁLIDA - Adiós
$

```

 En la línea 18, el "?" tiene que protegerse para que no sea interpretado por el shell.

### **Opción válida pero sin argumento**

Cuando el argumento de una opción está ausente, la variable `opcion` se inicializa con el carácter ":" y `OPTARG` contiene el valor de la opción involucrada (línea 14).

#### Ejemplo

En este caso, la estructura de control `case` trata el caso del argumento ausente:

```

$ nl gestusuario.sh
1 # El carácter ":" en 1ª posición permite gestionar mejor
2 # las opciones incorrectas
3 while getopt " :cxu:g:" opcion
4 do
5     case "$opcion" in
6         c) echo "Archivado"
7             ;;
8         x) echo "Extracción"
9             ;;
10        u) echo "Lista de usuarios que se tratarán: $OPTARG"
11            ;;
12        g) echo "Lista de grupos que se tratarán: $OPTARG"
13            ;;
14        :) echo "La opción $OPTARG requiere un argumento - Adiós"
15            exit 1
16            ;;
17        \?) echo "$OPTARG: OPCIÓN INVÁLIDA - Adiós"

```



```
18         exit 1
19         ;;
20     esac
21 done
22 echo "Análisis de opciones terminado"
23 exit 0
```

Llamada olvidando el argumento de la opción `-u`:

```
$ gestuario.sh -c -u
Archivado
La opción u requiere un argumento - Adiós
$
```

Las opciones se almacenan en los parámetros posicionales (`$1`, `$2` ...). Una vez han sido analizados, es posible deshacerse de ellos con el comando `shift` (ver capítulo Las bases de la programación shell - Variables reservadas del shell). Esto es interesante si quedan argumentos por tratar tras las opciones.

### Ejemplo

Las líneas añadidas detrás del bucle `while` permiten retirar las opciones de la lista de argumentos (línea 26). La expresión `OPTIND-1` representa el número de opciones analizadas y, por consiguiente, el valor del desplazamiento que se debe realizar.

```
$ nl gestuario.sh | tail
20     esac
21 done
22     echo "Análisis de opciones terminado"
23     echo "Antes de shift:"
24     echo "Lista de argumentos: $*"
25     echo "Índice de la siguiente opción a tratar: $OPTIND"
26     shift $((OPTIND-1))
27     echo "Después de shift:"
28     echo "Lista de argumentos: $*"
29     exit 0
$ gestuario.sh -c -u cristina,roberto arch1 arch2 arch3 arch4
Archivado
Lista de usuarios que se tratarán: cristina,roberto
Análisis de opciones terminado
Antes de shift:
Lista de argumentos: -c -u cristina,roberto arch1 arch2 arch3
arch4
Índice de la siguiente opción a tratar: 4
Después de shift:
Lista de argumentos: arch1 arch2 arch3 arch4
$
```

## Gestión de un proceso en segundo plano

El comando **wait** permite al shell esperar la finalización de un proceso ejecutado en segundo plano.

### Sintaxis

Esperar la finalización del proceso cuyo PID se pasa como argumento:

```
wait pid1
```

Esperar la finalización de todos los procesos ejecutados en segundo plano desde el shell actual:

```
wait
```

ksh
-----

bash
------

Esperar la finalización del proceso cuyo número de job se pasa como argumento:

```
wait %job
```

### Ejemplos

El comando **find** se ejecuta en segundo plano. Tiene el PID 13415:

```
$ find / -name passwd 1> /tmp/res 2> /dev/null&
[1] 13415
$ jobs
[1]+  Running   find / -name passwd >/tmp/res 2>/dev/null &
```

El shell se duerme esperando la finalización del proceso 13415:

```
$ wait 13415 # o wait %1
```

El shell se despierta cuando el proceso 13415 ha finalizado:

```
[1]+  Exit 1      find / -name passwd >/tmp/res 2>/dev/null
$
```



El PID del último comando ejecutado en segundo plano se encuentra en la variable especial **#!**.

El script **esperaproc.sh** ejecuta una copia de seguridad en segundo plano. Durante su ejecución, el shell realiza otras acciones. Después espera el final de la copia antes de realizar su verificación:

```
$ nl esperaproc.sh
 1  #! /usr/bin/ksh
 2  # Ejecución de un comando de copia de seguridad en segundo
plano
 3  find / | cpio -ocvB > /dev/rmt/0 &
 4  echo "El PID del proceso en segundo plano es: $!"
 5  # Mientras que el comando de copia de seguridad se ejecuta,
 6  # el script hace otras acciones
 7  echo "Inicio de las otras acciones"
 8  ...
 9  echo "Fin de las otras acciones"

10 # Espera el fin de la copia de seguridad para continuar
11 echo "-Sincronización - A la espera del fin de la copia"
```

```
12 wait $!  
13 echo "Copia terminada."  
14 echo "Comprobación de la copia"  
15 cpio -icvtB < /dev/rmt/0 > /backup/log_`date +%d%m%y`  
16 exit 0
```

\$

\$ **esperaproc.sh**

El PID del proceso en segundo plano es: 23014

Inicio de las otras acciones

Fin de las otras acciones

-Sincronización - A la espera del fin de la copia

... # El shell se duerme

...

Copia terminada.

Comprobación de la copia

...

...

\$

# Script de archivado incremental y transferencia sftp automática

## 1. Objetivo

Se trata de escribir un script que guarde una copia de seguridad de forma incremental de un directorio de una máquina de producción. Los archivos de copia (archivos cpio comprimidos) se transferirán a un servidor de copias de seguridad (servidor **venus**), en un directorio cuyo nombre dependerá del mes y año de la copia.

Directorios de la máquina de producción:

- /root/admin/backup: directorio de los scripts de copia de seguridad.
- /home/document: directorio de los documentos que se han de guardar.
- /home/lbackup: directorio local de archivos. Este directorio se limpiará todos los meses.

Directorios de la máquina de copias de seguridad:

- /home/dbackup/2015/01: archivos del mes de enero de 2015.
- /home/dbackup/2015/02: archivos del mes de febrero de 2015.

En el ejemplo que se presenta a continuación, estos directorios estarán creados previamente. No es el script de copia de seguridad el que los crea (pero sería fácilmente realizable).

La figura 2 representa el sistema de archivos de los dos servidores.

La copia de seguridad incremental usará tantos niveles de copia de seguridad como días tenga el mes. En principio, una copia de nivel 0 (copia de todos los archivos del directorio /home/document) se realiza el primer día de cada mes. Los días siguientes, solamente se archivarán los archivos modificados desde el día anterior.

Se utilizaran archivos indicadores de nivel (nivel0, nivel1...) para reflejar la fecha en la que las copias se llevan a cabo.

### Ejemplo

*El 01/01/2015: Copia de seguridad de nivel 0: creación del archivo de control "nivel0" y copia de seguridad de todos los archivos y directorios que estén en /home/document. A continuación, transferencia del archivo al servidor de copias.*

*El 02/01/2015: Copia de seguridad de nivel 1: creación del archivo de control "nivel1". A continuación, los archivos que sean más recientes que el archivo de control "nivel0" se copian. Transferencia del archivo al servidor de copia.*

*El 03/01/2015: Copia de seguridad de nivel 2: creación del archivo de control "nivel2". A continuación, los archivos que sean más recientes que el archivo de control "nivel1" se copian. Transferencia del archivo al servidor de copias.*

*etc.*

*El 01/02/2015: Copia de seguridad de nivel 0: borrado de todos los archivos de control del mes anterior. Creación de un nuevo archivo de control "nivel0" y copia de seguridad de todos los archivos y directorios que estén en /home/document. A continuación, transferencia del archivo al servidor de copias.*

*etc.*



La copia de seguridad incremental permite tener una copia diaria, archi-vando únicamente los archivos modificados o añadidos desde el día anterior. En caso de pérdida de un archivo, es suficiente con restaurar la última copia que contenga dicho archivo. Si el usuario desea restaurar todo el directorio **/home/document**, hay que restaurar todos los archivos del mes en curso,

empezando por el archivo de nivel 0.

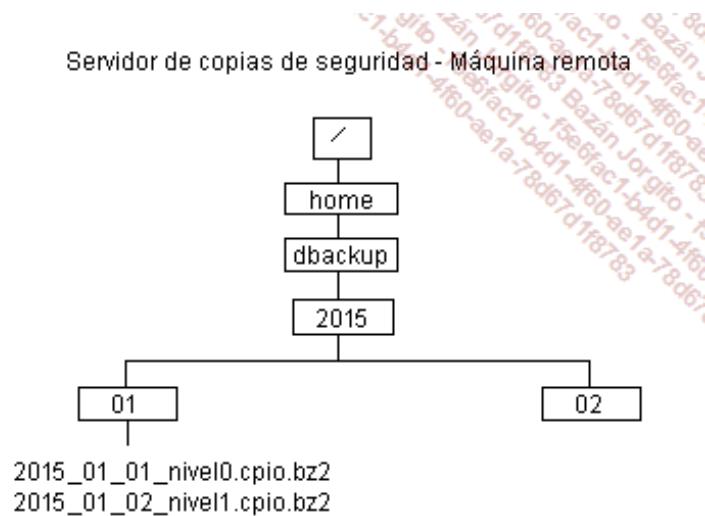
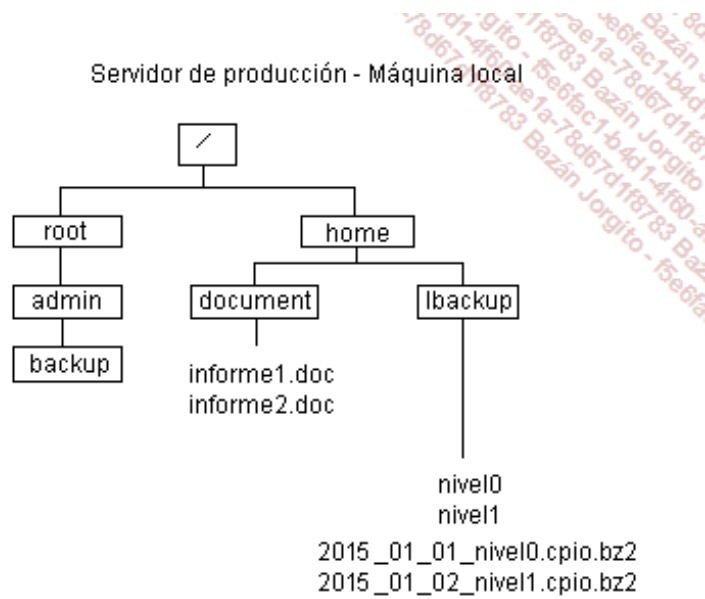


Figura 2: Sistemas de archivos de los servidores de producción y de copias de seguridad

El script de copias de seguridad se descompone en dos archivos:

- El archivo **uploadBackup.sh** es el programa principal.
- El archivo **funciones.inc.sh** contiene funciones llamadas desde el archivo **uploadBackup.sh**.

## 2. El archivo uploadBackup.sh

A continuación se muestra el código fuente con comentarios del script **uploadBackup.sh**:

```
1  #!/bin/bash
2  # -----
3  # Variables globales
4  # -----
5  # Directorio de los scripts en shell de backup
6  SCRIPTDIR="/root/admin/backup"
```

```

7 # Directorio de los archivos que se han de copiar
8 DATADIR=/home/document

9 # Directorio local de los archivos
10 ARCHIVEDIR="/home/lbackup"
11 # -----
12 # Librerías de funciones
13 # -----
14 # Incluir funciones
15 . $SCRIPTDIR/funciones.inc.sh

16 # Archivo de log
17 LOG=$ARCHIVEDIR/`getDate`.log
18 # -----
19 # Programa
20 # -----
21 exec 1>$LOG 2>&1

22 # -----
23 # Determinar el nivel de copia de seguridad
24 # El día 1 de cada mes => nivel 0
25 # -----
26 diaDeMes=`getDayForCalcul`
27 ((nivel=$diaDeMes-1))

28 case $nivel in
29 0) # Copia de seguridad total
30     # Limpieza del directorio de archivado
31     rm -i $ARCHIVEDIR/*.bz2 $ARCHIVEDIR/nivel*
32
33     # Creación del archivo de nivel (nivel 0)
34     touch $ARCHIVEDIR/nivel0
35     archivado="$ARCHIVEDIR/`getDate`_nivel0.cpio"

36     find $DATADIR | cpio -ocv | bzip2 -c >
$archivado.bz2
37     ;;
38 *)
39     touch $ARCHIVEDIR/nivel$nivel
40     archivado="$ARCHIVEDIR/`getDate`_nivel${nivel}.cpio"

41     ((nivelAnt=$nivel-1))
42     if [[ ! -f $ARCHIVEDIR/nivel$nivelAnt ]] ; then
43         echo "Archivo de nivel $nivelAnt ausente"
44         exit 1
45     fi

46     # Copia de seguridad
47     find $DATADIR -newer $ARCHIVEDIR/nivel$nivelAnt\
| cpio -ocv | bzip2 -c > archivado.bz2
48     ;;
49 esac

50 # Comprobación de la validez del archivado
51 if isArchivadoInvalido $archivado.bz2 ; then
52     echo "Archivado $archivado.bz2 INVÁLIDO - Archivo no transferido"
53     exit 1 ;
54 fi
55
56 if transferir ${archivado}.bz2 ; then
57     echo "Transferencia realizada con éxito"
58     exit 0
59 fi

60 echo "Error en la transferencia"

```

## Comentarios:

- Líneas 6, 8 y 10: Inicialización de tres variables globales que representan los nombres de los directorios que se manipularán en este script.
- Línea 15: Inclusión en el entorno actual (mediante el comando `.`) de las funciones definidas en el archivo **funciones.inc.sh**. La variable global `SCRIPTDIR` se utiliza por la orden de inclusión; por tanto, hay que definirla previamente: hay que poner especial atención en el orden de las instrucciones.
- Línea 17: Definición de otra variable global. La variable global `LOG` se inicializa realizando la llamada a la función `getDate` definida en el archivo **funciones.inc.sh**. Por tanto, hay que incluir este archivo previamente.
- Línea 21: Redirección de la salida estándar y de la salida de error estándar del script al archivo de log.
- Líneas 26 y 27: Cálculo del nivel de copia de seguridad a partir del día actual.

*Caso de la copia de seguridad total (nivel 0)*

- Línea 29: El primer día del mes todos los archivos se almacenan en formato `cpio` y se comprimen en formato `bz2`.
- Línea 31: Los archivos locales del mes anterior se eliminan.
- Línea 34: Creación del archivo de control `nivel0`, cuya fecha de última modificación (en este caso, la fecha de creación) se usará para realizar las copias de seguridad futuras.

*Caso de las copias de seguridad incrementales:*

- Línea 39: Creación del archivo de control reflejando el nivel de copia de seguridad.
- Línea 47: Copia de seguridad incremental: solamente los archivos modificados después del nivel anterior se tratan. La opción `-newer` del comando **find** permite realizar esta operación.

*Vuelta al tratamiento común:*

- Línea 51: Verificación de la validez del archivado con la función `isArchivadoInvalido`.
- Línea 56: Transferencia del archivado al servidor de copias de seguridad con la función `transferir`.

 Los comandos `cpio` y `bzip2` se presentan al capítulo Los comandos filtro.

### 3. El archivo `funciones.inc.sh`

#### La función `transferir`

Recibe un nombre de archivo como argumento y lo transfiere a la máquina remota **venus**, usando la cuenta **dbackup**. La función `transferir` devuelve un estado verdadero (0) en el caso de éxito o falso (1) de lo contrario.

```

1 # -----
2 # Función transferir
3 # Argumento: archivado a transferir
4 # -----
5 function transferir {
6     typeset mes

```

```

7 typeset ano
8
9 typeset archATransferir=$1
10
11 mes=`getMonth`
12 ano=`getYear`
13
14 sftp -b - "dbackup@$venus" <FIN
15 cd /home/dbackup/$ano/$mes
16 pwd
17 put $archATransferir
18 FIN
19
20 (( $? != 0 )) && return 1
21
22 # Comprobar si el archivado es válido en la máquina
de copias de seguridad
23
24 archEnMaquinaDest=$(basename $archATransferir)
25 ssh dbackup@$venus bzip2 -t
/home/dbackup/$ano/$mes/$archEnMaquinaDest
26
27 case $? in
28 0);;
29 255)
30 echo "Error del comando ssh"
31 return 1
32 ;;
33 *)
34 echo "Archivado\
/home/dbackup/$ano/$mes/$archEnMaquinaDest INVÁLIDO"
35 return 1
36 ;;
37 esac
38
39 return 0
40 }

```

#### Comentarios:

- Línea 9: El nombre del archivo que se transfiere se recibe por parámetro (\$1).
- Línea 14: Transferencia del archivo a la máquina **venus**, de manera no interactiva. Los comandos de sftp se leen por la entrada estándar (opción -b y doble redirección en lectura). El entorno de trabajo ha sido configurado para que la contraseña no se tenga que informar (ver el comando sftp del capítulo Los comandos filtro - Comandos de red seguros). La cuenta de la máquina remota es **dbackup**.
- Línea 20: Si el comando **sftp** genera error, la función devuelve el estado falso.
- Línea 25: Verificación (bzip2 -t) de la validez del archivado en la máquina distante.
- Líneas de la 27 a la 37: Comprobación del código de retorno de ssh y bzip2.
- Línea 28: Si el código de retorno vale 0, ssh y bzip2 se han ejecutado sin problemas y el archivado es válido.
- Línea 29: Si el código de retorno vale 255, ssh ha provocado un error.
- Línea 33: Si el código de retorno no es ni 0 ni 255, el archivado es inválido.

#### La función isArchivadoInvalido

Esta función recibe un archivo de tipo bz2 como argumento, devuelve verdadero si el archivado es inválido, falso en caso contrario.



```
41 #-----
42 # Argumento: nombre del archivado
43 # Verificación de la validez del archivado
44 # Retorno: verdadero si el archivado es válido,
# falso en caso contrario
45 #-----
46 function isArchivadoInvalido {
47     typeset archivado=$1
48     bzip2 -t $archivado 2>/dev/null && return 1
49     return 0
50 }
```

### **La función getDate**

La función getDate muestra la fecha del día, en formato aaaa\_mm\_dd.

```
51 # -----
52 # Función getDate
53 # Genera la fecha en formato aaaa_mm_dd
54 # -----
55 function getDate {
56     date '+%Y_%m_%d'
57 }
```

### **La función getYear**

La función getYear muestra el año actual, en formato aaaa.

```
58 # -----
59 # Función getYear
60 # Genera el año en formato aaaa
61 # -----
62 function getYear {
63     date '+%Y'
64 }
```

### **La función getMonth**

La función getMonth muestra el mes actual, en formato mm.

```
65 # -----
66 # Función getMonth
67 # Genera el mes en formato mm
68 # -----
69 function getMonth {
70     date '+%m'
71 }
```

### **La función getDayForCalcul**


La función getDayForCalcul muestra el día actual numéricamente con 1 o 2 cifras.



Este valor se destina para cálculos, no es necesario prefijarlo con 0 (que significa número octal para ciertos comandos de cálculo), ni espacio (que no es un carácter numérico).

```
72 # -----
```

```
73 # Función getDayForCalcul
74 # Genera el día en formato d o dd
75 # -----
76 function getDayForCalcul {
77     date '+%e' | sed 's/ //'
78 }
```

-  El comando **sed** se presenta en el capítulo El comando sed. Los comandos **ssh** y **sftp** se presentan en el capítulo Los comandos filtro.

# Ejercicios

Los archivos proporcionados para los ejercicios están disponibles en la carpeta dedicada al capítulo, en el directorio **Ejercicios/archivos**.

## 1. Funciones

### a. Ejercicio 1: funciones simples

Comandos útiles: **df**, **who**.

Escriba un script **audit.sh**:

- Escriba una función **users\_connect** que mostrará la lista de los usuarios conectados actualmente.
- Escriba una función **disk\_space** que mostrará el espacio en disco disponible.
- El programa principal mostrará el siguiente menú:

```
- 0 - Fin
- 1 - Mostrar la lista de usuarios conectados
- 2 - Mostrar el espacio en disco
Su opción:
```

Introducir la opción del usuario y llamar a la función adecuada. •

### b. Ejercicio 2: funciones simples, valor de retorno

Comandos filtro útiles: **awk**, **tr -d** (ver capítulo Los comandos filtro). Otros comandos útiles: **df**, **find**.

Escriba un script **explore\_sa.sh**:

- Programa principal:
  - El programa principal mostrará el menú siguiente:

```
0 - Fin
1 - Eliminar los archivos de tamaño 0 de mi directorio principal
2 - Controlar el espacio de disco del SA raíz
Su opción:
```

Introduzca la opción del usuario. ■

- La opción 0 provocará la finalización del script.
- La opción 1 llamará a la opción **limpieza**.
- La opción 2 causará la llamada a la función **sin espacio\_d**.
- En función del valor retornado por la función, mostrar el mensaje adecuado.
- Escriba la función **limpieza**: busque, a partir del directorio de inicio del usuario, todos los archivos que tengan tamaño 0 con objeto de eliminarlos (después de solicitar confirmación para cada archivo).
- Escriba la función **sin\_espacio\_d**: esta función verifica la utilización del sistema de archivos raíz y retorna verdadero si la tasa es superior al 80% y falso en caso contrario.

## Ejemplos de ejecución

```
$ explore_sa.sh

0 - Fin
1 - Eliminar los archivos de tamaño 0 de mi directorio principal
2 - Controlar el espacio de disco del SA raíz
Su opción: 1

rm: eliminar fichero regular vacio
`/home/cristina/.gconf/%gconf.xml'? n
etc.

$ explore_sa.sh

0 - Fin
1 - Eliminar los archivos de tamaño 0 de mi directorio principal
2 - Controlar el espacio de disco del SA raíz
Su opción: 2
Tasa de utilización del sistema de archivos raíz: NORMAL
```

### c. Ejercicio 3: paso de parámetros, retorno de valor

Escriba un script **calcul.sh**, que contendrá:

- Una función **esNum** que recibe un valor como argumento y que retorna verdadero si el valor es un número entero y falso en el caso contrario. Si define variables, estas deberán ser locales.
- Una función **suma** que recibirá un número cualquiera de parámetros (en principio, números). La función debe verificar, empleando la función **esNum**, que los argumentos recibidos son números y mostrar la suma de los argumentos. Si uno de los argumentos es incorrecto, este será ignorado. Si emplea variables, estas deberán ser locales.
- Una función **producto** que recibe un número cualquiera de parámetros (números). La función debe verificar, empleando la función **esNum**, que los argumentos recibidos son números y mostrar el producto de los argumentos. Si uno de los argumentos es incorrecto, este será ignorado. Si emplea variables, estas deberán ser locales.
- El programa principal:
  - El script **calcul.sh** recibirá como argumentos la operación que se ha de realizar, al igual que una serie de números.

```
$ calcul.sh producto 3 5 10
150
$ calcul.sh suma 3 5 10
25
```

Llamar a la función correspondiente a la operación solicitada y pasar los valores recibidos por el script. Mostrar por pantalla el resultado devuelto por la función.

### d. Ejercicio 4: archivos

Comando útil: **printf** (ver capítulo Mecanismos esenciales del shell - Mostrar en pantalla).

Sea el siguiente archivo de datos:

```
$ cat alumnos.txt
Nombre|Clase|Promedio
Luis|6to|3
Carlos|6to|14
```

```
Clarisa|6to|16
Jorge|6to|18
Pedro|6to|8
Damian|5to|10
Daniel|5to|11
Pablo|5to|7
Victor|5to|14
```

Escriba un script **stats.sh** que muestre por pantalla, con formato, las tres columnas del archivo según sigue:

```
$ stats.sh
Nombre      Clase      Promedio
Luis        6to        3
Carlos      6to        14
Clarisa     6to        16
Jorge       6to        18
Pedro       6to        8
Damian      5to        10
Daniel      5to        11
Pablo       5to        7
Victor      5to        14
```

### e. Ejercicio 5: archivos, funciones, menú select

Comando útil: **printf** (ver capítulo Mecanismos esenciales del shell - Mostrar en pantalla).

Sea el siguiente archivo de datos:

```
$ cat alumnos.txt
Nombre|Clase|Promedio
Luis|6to|3
Carlos|6to|14
Clarisa|6to|16
Jorge|6to|18
Pedro|6to|8
Damian|5to|10
Daniel|5to|11
Pablo|5to|7
Victor|5to|14
```

Escriba script **stats\_select.sh** (compatible ksh/bash) que:

- Permita mostrar los archivos de una clase dada.
- Permita el cálculo del promedio de una clase dada.

El script mostrará un menú que podrá ser escrito con la estructura **select**.

#### Ejemplo de ejecución

```
$ stats_select.sh
1) Extracto por clase
2) Promedio de una clase
3) Fin

Su opción: 1
Clase ? 6to
Luis      6to      3
Carlos    6to      14
Clarisa   6to      16
Jorge     6to      18
Pedro     6to      8
```

```
Su opción: 2
Clase ? 5to
Nota : 10
Nota : 11
Nota : 7
Nota : 14
Promedio de la clase de 5to: 10
```

## f. Ejercicio 6: archivos, tablas asociativas (bash 4, ksh93)

Obtenga el archivo **datos.txt** :

```
$ cat datos.txt
Juan Juan|46290|Valencia
Olivia Perez|24200|León
Carlos Izaguirre|24200|León
Alejandro Arevalo|24100|León
Jorge Olvido|26350|La Rioja
Jose Martinez|26350|La Rioja
```

Escriba un script **tabAsoc.sh** que cuente el número de habitantes por ciudad.

Ejemplo de resultado

```
$ tabAsoc.sh
Valencia => 1
La Rioja => 2
León => 3
```

# Introducción

Este capítulo presenta los caracteres especiales de las expresiones regulares. Estas son usadas por cierto número de comandos Unix y forman un mecanismo potente de selección de cadenas de caracteres.

Las expresiones regulares se componen de caracteres ordinarios y de caracteres con un significado particular. Existen dos tipos de expresiones regulares:

- Las expresiones regulares básicas (ERb).
- Las expresiones regulares extendidas (ERe).

Las ERb se usan en los comandos siguientes:

- **vi** (búsqueda y sustitución)
- **grep**
- **expr**
- **sed**


Las ERe se usan en los comandos siguientes:

- **grep** con la opción **-E**, **egrep**
- **awk**

## Caracteres comunes en ERb y ERe

Carácter especial	Significado
^	Comienzo de línea
\$	Fin de línea
.	Un carácter cualquiera
[lista_de_caracteres]	Un carácter de los citados en la lista
[^lista_de_caracteres]	Un carácter que no esté entre los citados en la lista
*	De 0 a n veces el carácter o grupo anterior
\<	Comienzo de palabra. Los caracteres que pueden formar parte de una palabra: [A-Za-z0-9_]
\>	Fin de palabra
\c	Protección del carácter especial c


### Ejemplos

 Los símbolos " " y " ⇆ " utilizados en los ejemplos representan las teclas [espacio] y [tabulación].

Expresión regular	Significado
sol	Cadena que contenga "sol". Ejemplos de correspondencia: Hoy ha hecho <b>sol</b> , ¿y mañana? ii <b>sol</b> todos los días!! Pasar sus vacaciones al <b>sol</b>
^sol	Cadena que empiece por "sol". Ejemplo de correspondencia: <b>sol</b> todos los días
sol\$	Cadena que termine por "sol". Ejemplo de correspondencia: Pasar sus vacaciones al <b>sol</b>
^[A-Z][5-9].	Cadena compuesta de tres caracteres: el primero es una mayúscula, el segundo es una cifra comprendida entre 5 y 9 y el último es un carácter cualquiera. Ejemplos de correspondencia: B6a Z5*
^\$	Cadena compuesta de un comienzo de línea seguido inmediatamente de un fin de línea. Por lo tanto, se trata de la cadena vacía.
^[ \t]*\$ (espacio o tabulación)	Cadena que contenga entre 0 (cadena vacía) y n caracteres de espacio o tabulación.
7 [0-7]A*	Cadena que contenga el carácter 7, seguido de un espacio, seguido de una cifra que no sea 8 y seguida de una A de 0 a n veces. Ejemplos de correspondencia:




	<p>x<b>7 6</b></p> <p>abc<b>7 9A</b>xy</p> <p><b>7 1AAAAAAAAA</b>bbbbbbbb</p>
[0-9][^A-Z_]\$	<p>Cadena cuyo penúltimo carácter es una cifra y el último no es ni una mayúscula ni un carácter subrayado.</p> <p>Ejemplos de correspondencia:</p> <p>AZER<b>1a</b></p> <p><b>3*</b></p> <p>89000<b>8b</b></p>
\<todo	<p>Cadena que contenga una palabra comenzando por "todo":</p> <p>Ejemplos de correspondencia:</p> <p>Hola a <b>todo</b> el mundo</p> <p>hay que decir que de <b>todos</b> modos</p> <p>Ejemplo de no correspondencia:</p> <p>mé<b>todo</b> de uso</p>
\<todo\>	<p>Cadena que contenga la palabra "todo":</p> <p>Ejemplos de correspondencia:</p> <p>Hola a <b>todo</b> el mundo</p> <p>Ejemplos de no correspondencia:</p> <p>Se lo dijo a <b>todos</b></p> <p>mé<b>todo</b> de uso</p>
[0-9][0-9]\.[0-9]	<p>Cadena que contenga dos cifras seguidas del carácter . seguido de otra cifra.</p> <p>Ejemplo de correspondencia:</p> <p>78<b>92.89</b></p>

 Las marcas ^ y \$ pierden su significado si no se colocan al comienzo y al final de la expresión regular respectivamente.

## Caracteres específicos de ERb

Carácter especial	Significado
<code>\{m\}</code>	m veces el carácter anterior
<code>\{m,\}</code>	Al menos m veces el carácter anterior
<code>\{m,n\}</code>	Entre m y n veces el carácter anterior
<code>\(ERb\)</code>	Memorización de una ERb
<code>\1, \2, ...</code>	Recordatorio de memorización

 En las ERb, el carácter "\" otorga un significado especial a los paréntesis y a las llaves.


### Ejemplo

<code> [0-9]{5} </code>	Cadena que contenga un número de cinco cifras rodeado de dos barras verticales. Ejemplos de correspondencia: calle Asunción <b> 08024 </b> Barcelona <b> 13005 </b> Ciudad Real
-------------------------	--

En el capítulo El comando sed se muestran ejemplos de sustitución.

## Caracteres específicos de ERe

Carácter especial	Significado
?	0 o 1 veces el carácter o grupo anterior
+	De 1 a n veces el carácter o grupo anterior
{m}	m veces el carácter anterior
{m,}	Al menos m veces el carácter anterior
{m,n}	Entre m y n veces el carácter anterior
(er1)	Agrupación
er1 er2 er3	Alternativas

 En las ERe, las llaves y los paréntesis pierden su significado especial si se les antepone una barra invertida.

### Ejemplos

ER	Significado
<code>^[+-]?[0-9]+\$</code>	Cadena que represente un nombre entero (al menos una cifra) con la posibilidad de estar precedido de un signo (0 o 1 veces) Ejemplos de correspondencia: <b>2</b> <b>-56</b> <b>+235</b> <b>789654</b>
<code>\\[0-9]{5}\\</code>	Cadena que contenga un número de cinco cifras rodeado por dos barras verticales. Ejemplos de correspondencia: Paseo de la Castellana  <b>28003</b>  Madrid   <b>33001</b>  Oviedo
<code>\\{[0-9]{2}\\}\$</code>	Cadena que termine con dos cifras entre llaves. Ejemplos de correspondencia: <b>nota1{23}</b> <b>{78}</b>
<code>^truc cosa\$</code>	Cadena que empiece con "truc" o acabe con "cosa".
<code>^(truc cosa)+\$</code>	Cadena compuesta de 1 a n ocurrencias de "truc" o de "cosa". Ejemplos de correspondencia: <b>truc</b> <b>cosa</b> <b>truccosatruc</b> <b>cosacosa</b>

# Uso de expresiones regulares por comandos

## 1. El comando vi

Las ERb se usan en el editor **vi** para la búsqueda y sustitución de cadenas de caracteres.

### Sintaxis

Búsqueda (modo comando):

```
/expresión-regular-básica
```

Sustitución (modo ex):

```
:[dirección[,dirección]]s/expresión-regular-básica/  
expresión-de-reemplazo/[flags]
```


En el capítulo El comando sed se muestran ejemplos de sustitución con el comando **sed**, que utiliza la misma sintaxis que el editor **vi**.

## 2. El comando grep

Este párrafo ilustra el uso de expresiones regulares mediante el comando **grep** normalizado por POSIX. Usado con la opción **-E**, el comando entiende las expresiones regulares extendidas.

### Sintaxis básica

```
grep [-iv...] expresión-regular-básica [ arch1 ... ]  
grep [-iv...] -E expresión-regular-extendida [ arch1 ... ]
```

 El comando **grep** usado con la opción **-E** reemplaza el comando **egrep**. Para otras opciones, consultar el manual del comando.

### Ejemplos

A continuación se muestra el archivo **tel2.txt**:

```
$ cat tel2.txt  
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915351478  
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|932282177  
  
Hernández Darín, Alberto|plaza mayor|13190|Corral de Calatrava|  
926448829  
  
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de Calatrava|  
926443602  
  
Martínez Parra, Marta|calle de la Santa Trinidad|38870|La Calera|  
984122119  
Expósito Heredia, Pedro|calle del castillo|38870|La Calera|  
984122369  
  
$
```

*Búsqueda de la cadena "calatrava" en mayúsculas o minúsculas:*

```
$ grep -i 'calatrava' tel2.txt
Hernández Darín, Alberto|plaza mayor|13190|Corral de Calatrava|926448829
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de Calatrava|926443602
$
```

*Búsqueda de las líneas que comiencen por la letra 'G':*

```
$ grep '^G' tel2.txt
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de
Calatrava|926443602
$
```

*Búsqueda de las líneas que terminen por 9:*

```
$ grep '9$' tel2.txt
Hernández Darín, Alberto|plaza mayor|13190|Corral de
Calatrava|926448829
Martínez Parra, Marta|calle de la Santa Trinidad|38870|
La Calera|984122119
Expósito Heredia, Pedro|calle del castillo|38870|
La Calera|984122369
$
```

*Búsqueda de las líneas que contengan dos ocurrencias sucesivas de la letra 'r':*

```
$ grep 'rr' tel2.txt
Hernández Darín, Alberto|plaza mayor|13190|Corral de
Calatrava|926448829
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de
Calatrava|926443602
Martínez Parra, Marta|calle de la Santa Trinidad|38870|
La Calera|984122119
```

*Mismo ejemplo usando una ERb:*

```
$ grep 'r{2}' tel2.txt
Hernández Darín, Alberto|plaza mayor|13190|Corral de Calatrava|
926448829
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de Calatrava|
926443602
Martínez Parra, Marta|calle de la Santa Trinidad|38870|
La Calera|984122119
```


*Mismo ejemplo usando una ERe:*

```
$ grep -E 'r{2}' tel2.txt
Hernández Darín, Alberto|plaza mayor|13190|Corral de
Calatrava|926448829
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de
Calatrava|926443602
Martínez Parra, Marta|calle de la Santa Trinidad|38870|
La Calera|984122119
$
```

*Mostrar todas las líneas que no estén en blanco:*

```
$ grep -v '^[_ ]*$' tel2.txt
```

```
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915351478
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|932282177
Hernández Darín, Alberto|plaza mayor|13190|Corral de Calatrava|
926448829
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de Calatrava|
926443602
Martínez Parra, Marta|calle de la Santa Trinidad|38870|
La Calera|984122119
Expósito Heredia, Pedro|calle del castillo|38870|La Calera|
984122369
$
```

 Una línea en blanco puede ser una línea vacía o una línea que contenga una serie de espacios o tabulaciones.

*Mostrar las líneas que contengan las cadenas calatrava o madrid (sin diferenciar mayúsculas y minúsculas):*

```
$ grep -iE 'Calatrava|madrid' tel2.txt
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915351478
Hernández Darín, Alberto|plaza mayor|13190|Corral de Calatrava|
926448829
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de Calatrava|
926443602
$
```

### 3. El comando expr

#### Sintaxis

expr cadena-de-caracteres: expresión-regular-básica

Este comando ofrece un operador ":" que permite comprobar la correspondencia entre una cadena de caracteres y una expresión regular.

Funcionamiento del operador ":":

- El número de caracteres de cadena-de-caracteres correspondiente a la ERbexpresión-regular-básica se muestra por pantalla.
- Si cadena-de-caracteres se corresponde con expresión-regular-básica, el comando devuelve el código verdadero. Devuelve el código falso en caso contrario.
- La expresión regular se compara con relación al principio de la variable (el "^" **está implícito en la ERb**).
- Si una parte de la expresión regular se graba con \( \), el comando muestra en el terminal la parte de la cadena correspondiente.

#### Ejemplos

Verificar que el usuario haya introducido un número:

```
$ read num1
250
$ read num2
a8
```

La variable **num1** contiene únicamente cifras:

La variable **num2** contiene al menos un carácter que no es una cifra. La ERb no se satisface:

```
$ expr "$num1": '[0-9][0-9]*$' # equivalente a '^ [0-9][0-9]*$'
3
$ echo $?
0
```

```
$ expr "$num2": '[0-9][0-9]*$' # equivalente a '^ [0-9][0-9]*$'
0
$ echo $?
1
$
```

Contar el número de caracteres que hay en una variable:

```
$ linea="Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915351478"
$ expr "$linea": '.*'
57
```

Mostrar la parte de la cadena correspondiente a la grabación (en este caso, el código postal):

```
$ expr "$linea": '.*|[0-9]\{5\}|$'
28023
$
```

Como muestra la figura 1, el carácter \* es avaricioso: busca siempre la cadena más larga (en este caso, la "|" más a la derecha):

```
$ expr "$linea": '\(.*\)|'
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid
```

```
$ expr "$linea" : "\(.*\)|"
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid
└──────────────────────────────────────────────────┘
\(.*)
```

Figura 1: ¡El carácter "\*" es avaricioso!

Solución que permite pararse en el primer "|" (ver figura 2):


```
$ expr "$linea": '\([^|]*\)|'
Méndez Roca, Gisela
$
```

```
$ expr "$linea" : "\([^|]*\)|"
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid
└──────────────────────────────────────────┘
\([^|]*\)
```

Figura 2: Recuperar la cadena más corta

El script **testnum.sh** comprueba si el texto introducido por teclado es un número positivo o negativo. En caso afirmativo, realiza la suma de los números entrados.

```
$ nl testnum.sh
 1  #!/usr/bin/sh
 2  suma=0
 3  while:
 4  do
 5      echo "Introduzca un número entero: \c"
 6      if read numero
 7      then
 8          if ( expr "$numero": '[0-9][0-9]*$' || \
 9              expr "$numero": '-[0-9][0-9]*$') > /dev/null
10      then
11          suma=`expr $suma + $numero`
12      else
13          echo "Entrada incorrecta"
14      fi
15      else
16          break
17      fi
18  done
19  echo "Resultado: $suma"
20  Exit 0
```

 La barra invertida al final de la línea 8 permite enmascarar el carácter "salto de línea" para poder escribir el comando en dos líneas. La salida del comando **expr** se elimina, ya que en este caso se usa el código de retorno.

```
$ testnum
Introduzca un número entero: 1
Introduzca un número entero: 8000
Introduzca un número entero: -10
Introduzca un número entero: 2
Introduzca un número entero: aaa
Entrada incorrecta
Introduzca un número entero: ^d
Resultado: 7993
$
```

## 4. sed y awk

Las utilidades **sed** y **awk** se presentarán respectivamente en los capítulos El comando sed y El lenguaje de programación awk.



# Ejercicios

Los archivos proporcionados para los ejercicios están disponibles en el directorio dedicado al capítulo, en **Ejercicios/archivos**.

## 1. Expresiones regulares

### a. Ejercicio 1: expresiones regulares con vi

Sea el archivo **expr.txt**:

```
$ cat expr.txt
felipe      10 plaza de la concordia      91.511.11.11
annie2     25/27 calle Victor Hugo      91.485.22.48
fernando   20 valencia                   96.221.33.33
cristina   avenida de la ilustración     93/455/78/52
cris       98.622.33.44
jorje      48 bravo murillo             630.22.53.48
XincX     45 plaza de neptuno          915.45.45.78
annie2     25 calle de Victor Hugo      910.48.22.48

XristiX   35/36 calle del querol       920/54/58/45
XarinX    Avda. de la marina           920.54.58.65
```

Realice las siguientes operaciones con el editor **vi** (modo **ex**) en el archivo **expr.txt**:

1. Los números de teléfono que terminen en 48 deberán en adelante terminar en 50.
2. En la línea 2, sustituya la cadena "annie2" por "annie" (sin desplazarse a la línea 2).
3. Sustituir las líneas vacías por "RAS".
4. Sustituya todos los nombres que comiencen y terminen con la letra X por XxxxxxxX.
5. Sustituya cada cifra al final de la línea por 0, salvo si esta cifra es igual a 8.
6. Sustituya los "/" separadores de los números de teléfono por ".".
7. Los campos deberán estar separados por "|" y no por tabulaciones o espacios.
8. Inserte un carácter "|" al principio y al final de cada línea.

### b. Ejercicio 2: grep

Partiendo del archivo **php.ini** proporcionado:

1. Muestre las líneas que comienzan con "mysql".
2. Muestre las líneas que terminan con "On".
3. Muestre las líneas que terminan con "On" y que no tengan un ";" en la primera posición.
4. Muestre las líneas que terminan en On (sin diferenciar mayúsculas y minúsculas).
5. Combinando **find** y **grep**, busque los archivos regulares que contengan la palabra "tr". Las cadenas "<tr>" o "</tr>" no deben ser devueltas.

# Uso del comando sed

El comando **sed** (stream editor) es un editor de texto no interactivo. Permite automatizar el tratamiento de archivos de texto. Este capítulo presenta las principales funcionalidades del comando.

## Sintaxis básica

```
sed [-n] acción [ arch1 ... ]  
sed [-n] -e acción1 [ -e acción2 ... ] [ arch1 ... ]  
sed -f script-sed [ arch1 ... archn ]
```

Las acciones especificadas se ejecutan en cada línea del archivo. El resultado del tratamiento se mostrará por la salida estándar. Si múltiples acciones se especifican en la línea de comandos, cada una de ellas estará precedida con la opción **-e**.



El comando **sed** no modifica el archivo de origen.

## Sintaxis de una acción

[dirección[,dirección]]comando[argumentos]

Una acción se compone sintácticamente de:

- Una parte de dirección que permite especificar sobre qué líneas se debe ejecutar el comando.
- El comando que se va a ejecutar.
- Los argumentos del comando.

## Sintaxis de una dirección

Tipo de dirección	Líneas tratadas
Ninguna dirección	Todas las líneas.
<b>Direcciones de tipo 1</b>	
n	Línea n.
\$	Última línea.
/ERb/	Líneas correspondientes a la expresión regular.
<b>Direcciones de tipo 2</b>	
n1,n2	Línea n1 hasta línea n2.
/ERb1/,/ERb2/	La primera línea tratada será la primera que se corresponda con ERb1. El tratamiento seguirá sobre todas las líneas hasta que sed encuentre una línea que se corresponda con ERb2. Esta última también se tratará.

## Sintaxis del comando

Comando	Argumento	Tipo de dirección soportado (máximo)	Significado
d	Sin argumento	2	No mostrar las líneas especificadas (delete).



# Ejemplos

## 1. Uso de sed en línea de comandos

➤ Los símbolos " " y "↵" usados en los ejemplos representan las teclas "espacio" y "tabulación".

A continuación se muestra el contenido del archivo que se usará en los ejemplos siguientes:

```
$ cat tel2.txt
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915351478
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|932282177

Hernández Darín, Alberto|plaza mayor|13190|Corral de Calatrava|
926448829

Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de Calatrava|
926443602

Martínez Parra, Marta|calle de la Santa Trinidad|38870|La Calera|
984122119
Expósito Heredia, Pedro|calle del castillo|38870|La Calera|
984122369

$
```

### a. El comando d (delete)

El comando **d** impide que se muestren las líneas seleccionadas en la parte de dirección.

#### Ejemplo

*No mostrar las líneas blancas:*

```
$ sed '/^[ \t]*$/d' tel2.txt
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915.351.478
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|
932.282.177
Hernández Darín, Alberto|plaza mayor|13190|Corral de Calatrava|
926/448/829
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de Calatrava|
926.443.602
Martínez Parra, Marta|calle de la Santa Trinidad|38870|La Calera|
984.122.119
Expósito Heredia, Pedro|calle del castillo|38870|La Calera|
984.122.369
$
```

➤ Una línea blanca puede ser una línea vacía o una línea que contenga una serie de espacios o tabulaciones.

### b. El comando p (print)

El comando **p** permite mostrar por pantalla las líneas seleccionadas desde la parte de dirección. Por

defecto, **sed** muestra igualmente todo el contenido del archivo. Para modificar este comportamiento, hay que usar la opción **-n**.

### Ejemplo

Mostrar las líneas 1 a 2. Por defecto, **sed** muestra además el contenido del archivo. Las líneas solicitadas aparecen dos veces:

```
$ sed '1,2p' tel2.txt
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915.351.478
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915.351.478
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|932.282.177
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|932.282.177
Hernández Darín, Alberto|plaza mayor|13190|Corral de Calatrava|
926/448/829

Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de Calatrava|
926.443.602

Martínez Parra, Marta|calle de la Santa Trinidad|38870|La Calera|
984.122.119
Expósito Heredia, Pedro|calle del castillo|38870|La Calera|
984.122.369

$
```

La opción **-n** especifica al comando **sed** que no tiene que volver a mostrar todo el archivo:

```
$ sed -n '1,2p' tel2.txt
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915.351.478
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|
932.282.177

$
```

## c. El comando **w** (write)

El comando **w** permite escribir en un archivo las líneas seleccionadas desde la parte de dirección. Como para el comando **p**, **sed** muestra igualmente todo el contenido del archivo por pantalla. Para modificar este comportamiento, hay que usar la opción **-n**.

### Ejemplo

Almacenar en el archivo **dpt\_08** todas las personas que vivan en la provincia de Barcelona y en el archivo **dpt\_38**, todas las personas que vivan en la provincia de Santa Cruz de Tenerife. La opción **-ese** antepone a cada acción:

```
$ sed -n -e '/^[^|]*|^[^|]*|08/w dpt_08'
-e '/^[^|]*|^[^|]*|38/w dpt_38' tel2.txt
$ cat dpt_08
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|
932.282.177
$ cat dpt_38
Martínez Parra, Marta|calle de la Santa Trinidad|38870|La Calera|
984.122.119
Expósito Heredia, Pedro|calle del castillo|38870|La Calera|
984.122.369

$
```

## d. Negación de un comando (!)

El carácter ! colocado delante del comando permite ejecutar este último sobre todas las líneas, excepto sobre aquellas que se correspondan con la parte de dirección.

### Ejemplo

*No mostrar las líneas blancas (usando la negación):*

```
$ sed -n '/^[ \t]*$/!p' tel2.txt
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915.351.478
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|
932.282.177
Hernández Darín, Alberto|plaza mayor|13190|Corral de Calatrava|
926/448/829
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de Calatrava|
926.443.602
Martínez Parra, Marta|calle de la Santa Trinidad|38870|La Calera|
984.122.119
Expósito Heredia, Pedro|calle del castillo|38870|La Calera|
984.122.369
$
```

## **e. El comando s (sustitución)**

El comando **s** permite sustituir una cadena de caracteres por otra en las líneas seleccionadas por la parte de dirección.

### Primer ejemplo

*Trabajar con el contenido de una variable:*

```
$ echo $arg
user1,user2,user3
$ echo $arg | sed 's/,/ /g'
user1 user2 user3
$ lista_users=$(echo $arg | sed 's/,/ /g')
$ echo $lista_users
user1 user2 user3
$ for user in $lista_users
> do
> ...
```

### Segundo ejemplo

*El carácter "&" usado en la parte de remplazo representa la cadena que se corresponde con la expresión regular (ver figura 1):*

```
$ echo $linea
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915.351.478
$ echo $linea | sed 's/.*/(&)/'
(Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915.351.478)
$
```

```
$ echo $linea | sed 's/.*( & )/'
```

Expresión regular ".\*":

Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915.351.478

. \*

Expresión de remplazo "(&)":

( Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915.351.478 )

( & )

Figura 1: Uso de la sustitución y del carácter especial "&"

Tercer ejemplo

Inserción de la cadena "(Ciudad Real)" justo después de la población de las personas que vivan en dicha provincia. Se harán dos grabaciones (ver figura 2) en la expresión regular y se recordarán en la segunda parte mediante las referencias \1 y \2.

```
$ sed 's/^\([^|]*|\([^|]*|13...|\([^|]*\)\\(.*)\)$/\1
(Ciudad Real)\2/' tel2.txt
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|915.351.478
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|
932.282.177
Hernández Darín, Alberto|plaza mayor|13190|Corral de Calatrava
(Ciudad Real)|926/448/829
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de Calatrava
(Ciudad Real)|926.443.602
Martínez Parra, Marta|calle de la Santa Trinidad|38870|La Calera|
984.122.119
Expósito Heredia, Pedro|calle del castillo|38870|La Calera|
984.122.369
$
```

's/^\([^|]\*|\([^|]\*|13...|\([^|]\*\)\\(.\*)\)\$/\1 (Ciudad Real)\2/'

Expresión regular:

^	[^ ]*		[^ ]*		13 . .	[^ ]*	\\	(.*)	\$
								\1	
								\\	
								\\2	

Expresión de remplazo:

Hernández Darín, Alberto plaza mayor 13190 Corral de Calatrava	926/448/829
_____	
\1	(Ciudad Real) \2

Figura 2: Inserción de una cadena entre dos subcadenas

## 2. Script sed

Un script **sed** es un archivo de texto que contiene un conjunto de acciones `sed` que se ejecutarán en las líneas de un archivo de datos (o en los datos de la entrada estándar si se da el caso).

Todas las acciones se ejecutarán de inicio sobre la primera línea del archivo, después sobre la segunda y así sucesivamente. Cuando la parte de dirección no se especifica, las acciones se aplican sobre todas las líneas.

### Ejemplo

```
$ nl script.sed
 1  /^[ \t ]*$/d
 2  li\
 3  Listado de Clientes:
 4  s/^\([^\|]*\|^\|13...\|^\|*\)\(.*\)$/\1 (Ciudad Real)\2/
 5  s/^\([^\|]*\|^\|08...\|^\|*\)\(.*\)$/\1 (Barcelona)\2/
 6  s/^\([^\|]*\|^\|28...\|^\|*\)\(.*\)$/\1 (Madrid)\2/
 7  s/^\([^\|]*\|^\|38...\|^\|*\)\(.*\)$/\1 (S.C.
Tenerife)\2/
 8  s/\(...\)\(...\)\(...\)$/\1.\2.\3/
 9  $a\
10  Fin del tratamiento\
11  Bye ...
$
```

- Línea 1: Si la línea está en blanco, no mostrarla.
- Línea 2: Insertar la frase citada en la línea 3 justo debajo de la línea 1.
- Línea 4: Insertar (Ciudad Real) después de la población cuando la persona sea de esta provincia.
- Línea 5: Insertar (Barcelona) después de la población cuando la persona sea de esta provincia.
- Línea 6: Insertar (Madrid) después de la población cuando la persona sea de esta provincia.
- Línea 7: Insertar (S.C. Tenerife) después de la población cuando la persona sea de esta provincia.
- Línea 8: Reemplazar las "/" de los números de teléfono por ".".
- Línea 9: Añadir, después de la última línea del archivo (\$), las dos frases citadas en las líneas 9 y 10.

```
$ sed -f script.sed tel2.txt
Listado de Clientes:
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid (Madrid)|
915.351.478
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona
(Barcelona)|932.282.177
Hernández Darín, Alberto|plaza mayor|13190|Corral de Calatrava
(Ciudad Real)|926.448.829
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de Calatrava
(Ciudad Real)|926.443.602
Martínez Parra, Marta|calle de la Santa Trinidad|38870|La Calera
(S.C. Tenerife)|984.122.119
Expósito Heredia, Pedro|calle del castillo|38870|La Calera
(S.C. Tenerife)|984.122.369
Fin del tratamiento
Bye ...
$
```



# Ejercicios

Los archivos proporcionados para los ejercicios están disponibles en la carpeta dedicada al capítulo, en el directorio **Ejercicios/Archivos**.

## 1. Expresiones regulares

### a. Ejercicio 1: inserción de marcadores en un archivo

Sea el archivo **fechas\_curs.txt**:

```
$ cat fechas_curs.txt
unix
28-30 ene
17-19 jun
18-20 nov

shell
23 mar
15 jul
7 sep
```

Empleando el comando **sed**, transforme este archivo de la siguiente forma:

```
unix
<date>28-30 ene</date>
<date>17-19 jun</date>
<date>18-20 nov</date>

shell
<date>23 mar</date>
<date>15 jul</date>
<date>7 sep</date>
```

### b. Ejercicio 2: formato de archivos

Tome el archivo **.bash\_profile**. Muestre el archivo con el comando **nl**, que numera las líneas:

```
$ nl .bash_profile
 1 # .bash_profile

 2 # Get the aliases and functions
 3 if [ -f ~/.bashrc ]; then
 4     . ~/.bashrc
 5 fi

 6 # User specific environment and startup programs
 7 PATH=$PATH:$HOME/bin
```

1. Elimine los espacios que preceden a los números de línea.

```
1 # .bash_profile
2 # Get the aliases and functions
3 if [ -f ~/.bashrc ]; then
4     . ~/.bashrc
```

```
5     fi
6     # User specific environment and startup programs
7     PATH=$PATH:$HOME/bin
```

2. Ponga también el número de línea entre corchetes.

```
[1]     # .bash_profile
[2]     # Get the aliases and functions
. . .
```

# Principio

El lenguaje de programación **awk** es una utilidad adaptada al tratamiento de archivos de texto. Permite realizar acciones sobre registros de datos incluso estructurados en campos. El nombre "awk" tiene como origen las iniciales de cada uno de sus autores: Aho, Weinberger y Kernighan. Este capítulo presenta las funcionalidades principales de las recientes versiones del lenguaje awk, llamadas nawk (new awk) en varias plataformas Unix. En Linux, el comando awk es un enlace simbólico al intérprete gawk (GNU awk).

## 1. Sintaxis

```
awk [-F] '{acción-awk}' [ arch1 ... archn ]  
awk [-F] -f script-awk [ arch1 ... archn ]
```

El comando **awk** recibe como argumento la lista de archivos que se han de tratar. Ante la ausencia de archivos en la línea de comandos, **awk** trabaja con los datos que le lleguen por su entrada estándar. Por lo tanto, este comando puede ponerse después de una tubería de comunicaciones.


## 2. Variables especiales

### a. Variables predefinidas a partir de la ejecución de awk

La tabla siguiente presenta las principales variables internas del lenguaje **awk** presentes en memoria desde el primer momento de su ejecución. El valor de estas variables puede modificarse, si se desea, en función de la estructura de datos que se han de tratar.

Nombre de la variable	Valor por defecto	Función de la variable
RS	Salto de línea (Newline) (\n)	Record Separator: Carácter separador de registros (líneas).
FS	Serie de espacios o tabulaciones	Field Separator: Caracteres separadores de campos.
OFS	Espacio	Output Field Separator: Separador de campo usado para la visualización.
ORS	Salto de línea (Newline) (\n)	Output Record Separator: Carácter separador de registros en la salida.
ARGV	-	Tabla inicializada con los argumentos de la línea de comandos (opciones y nombre del script awk excluidos).
ARGC	-	Número de elementos contenidos en la tabla ARGV.
ENVIRON	Variables de entorno exportadas por el shell	Tabla que contiene las variables de entorno exportadas por el shell.

Por defecto, un registro se corresponde con una línea (conjunto de caracteres terminado con "\n").

 Cuando la variable FS se inicializa con un mínimo de 2 caracteres, este valor se interpreta como una expresión regular.

## b. Variables inicializadas en el momento del tratamiento de una línea

Los registros se tratan de manera sucesiva. El registro actual es automáticamente dividido en campos y un cierto número de variables internas **awk** se inicializan entonces. La tabla siguiente muestra la lista de las principales variables.

Nombre de la variable	Valor de la variable
\$0	Valor del registro actual.
NF	Number of Field: Número de campos del registro actual.
\$1 \$2 ... \$NF	\$1 contiene el valor del primer campo, \$2 el valor del segundo, ..., y \$NF el valor del último campo (NF se reemplaza por su valor).
NR	Number: Índice del registro actual (NR vale 1 cuando la primera línea se está tratando, después se incrementa a partir de que <b>awk</b> cambia de registro).
FNR	File Number: Índice del registro actual relativo al archivo en curso.
FILENAME	Nombre del archivo en curso.

➤ Contrariamente a lo que sucede con las variables de shell, el símbolo "\$" de las variables **awk** \$1, \$2, etc., forma parte del nombre de las variables.

## c. Ejemplos simples

### Primer ejemplo

```
$ ps -ef | awk '{print $1,$8}'
```

En este caso, **awk** trabaja con el resultado del comando **ps -ef**. La parte escrita en itálica representa la acción que **awk** debe ejecutar en cada línea. Las comillas simples son indispensables para impedir que el shell interprete los caracteres destinados al comando **awk**. Las instrucciones tienen que ponerse entre llaves. La función integrada **print** mostrará por pantalla los campos 1 y 8 de cada línea (correspondientes en este caso al propietario y al nombre de cada proceso).

*Resultado del comando:*

```
$ ps -ef | awk '{print $1,$8}'
UID CMD
root init
...
root /usr/bin/gdm
root /usr/bin/gdm
gdm /usr/bin/gdmlogin
gdm /usr/bin/xsri
root /usr/sbin/sshd
cristina -bash
cristina ps
```

### Segundo ejemplo


La función **print** también puede aceptar cadenas de caracteres como argumento:

```
$ ps -ef | awk '{print "User: " , $1 , "\tComando: " , $8}'
```

```

User: UID      Comando:  CMD
User: root    Comando:  init
...
User: root    Comando:  /usr/bin/gdm
User: root    Comando:  /usr/bin/gdm
User: gdm     Comando:  /usr/bin/gdmlogin
User: gdm     Comando:  /usr/bin/xsri
User: root    Comando:  /usr/sbin/sshd
User: cristina Comando:  -bash
User: cristina Comando:  ps
$

```

 \t representa el carácter de tabulación.

### Tercer ejemplo

A continuación se muestra el archivo de texto **tel3.txt**, que va a ser tratado por **awk**. Los campos se separan con el carácter "|":

```

$ cat tel3.txt
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915.351.478
Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|Barcelona|
932.282.177
Hernández Darín, Alberto|plaza mayor 15/17|13190|Corral de
Calatrava|926/448/829
Gómez Bádenas, Josefina|calle Sagasta nº 3|13190|Corral de
Calatrava|926.443.602
Martínez Parra, Marta|calle de la Santa Trinidad 8-10|38870|
La Calera|984.122.119
Expósito Heredia, Pedro|calle del castillo 3|38870|La Calera|
984.122.369
$

```

Modificación del separador de campo (opción **-F**), visualización del número de línea y el nombre del cliente:

```

$ awk -F'|' '{print "Línea: " , NR , "Cliente: " , $1}' tel3.txt
Línea: 1 Cliente: Méndez Roca, Gisela
Línea: 2 Cliente: Ruiz del Castillo, Marcos
Línea: 3 Cliente: Hernández Darín, Alberto
Línea: 4 Cliente: Gómez Bádenas, Josefina
Línea: 5 Cliente: Martínez Parra, Marta
Línea: 6 Cliente: Expósito Heredia, Pedro
$

```

 Si la función **print** no recibe argumentos, imprime \$0.

## 3. Criterios de selección

Es posible seleccionar los registros sobre los cuales se ejecutará la acción.

### Sintaxis

```
awk [-F] 'criterio {acción-awk}' [ arch1 ... archn ]
```

El criterio de selección se puede expresar de diferentes formas.


## a. Expresiones regulares

Los registros que se han de tratar pueden seleccionarse utilizando las expresiones regulares extendidas.

### Primer ejemplo

Mostrar las líneas del archivo **tel3.txt** que contengan por lo menos una "/":

```
$ awk -F'|' ' /[/] / {print $0}' tel3.txt
Hernández Darín, Alberto|plaza mayor 15/17|13190|Corral de
Calatrava|926/448/829
$
```

 Por defecto, el criterio se corresponde con \$0.

Es posible hacer que un campo en particular corresponda con una expresión regular. En este caso, hay que usar el operador de concordancia (~) o el de no concordancia (!~).

### Segundo ejemplo

Mostrar los clientes localizados en la provincia de Ciudad Real:

```
$ awk -F'|' ' $3 ~ /^13/ {print $0}' tel3.txt
Hernández Darín, Alberto|plaza mayor 15/17|13190|Corral de
Calatrava|926/448/829
Gómez Bádenas, Josefina|calle Sagasta nº 3|13190|Corral de
Calatrava|926.443.602
```

Mostrar todos los clientes que no están en la provincia de Ciudad Real:

```
$ awk -F'|' ' $3 !~ /^13/ {print $0}' tel3.txt
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915.351.478
Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|Barcelona|
932.282.177
Martínez Parra, Marta|calle de la Santa Trinidad 8-10|38870|
La Calera|984.122.119
Expósito Heredia, Pedro|calle del castillo 3|38870|La Calera|
984.122.369
$
```

## b. Verificaciones lógicas

El criterio puede ser una expresión compuesta de operadores y que devuelve el valor de verdadero o falso.

### Operadores comunes de verificación

Operador	Significado
<	Menor
>	Mayor
<=	Menor o igual
>=	Mayor o igual
==	Verificación de igualdad
!=	Verificación de desigualdad

~	Correspondencia con una expresión regular
!~	No correspondencia con una expresión regular
!	Negación
&&	Y lógica
	O lógica
(expresión)	Agrupación

### Ejemplo

Mostrar el nombre y el teléfono de los clientes de las líneas 1 y 2:

```
$ awk -F'|' 'NR == 1 || NR == 2 {print $1, " - ", $5}' tel3.txt
Méndez Roca, Gisela - 915.351.478
Ruiz del Castillo, Marcos - 932.282.177
$
```

➤ La cifra 0 y la cadena vacía son valores falsos. Calquier otro valor es verdadero.

### c. Intervalos de líneas

Tratamiento de las líneas de la 2 a la 4:

```
$ awk -F'|' 'NR == 2 , NR == 4 {print $1, " - ", $5}' tel3.txt
Ruiz del Castillo, Marcos - 932.282.177
Hernández Darín, Alberto - 926/448/829
Gómez Bádenas, Josefina - 926.443.602
$
```

## 4. Estructura de un script awk

A partir de un cierto número de acciones que se tengan que realizar sobre los datos, es más cómodo escribir un script **awk**. Un script **awk** puede contener una sección BEGIN, una sección END y de 0 a n secciones intermedias. Todas las secciones son opcionales.

### a. BEGIN

La sección BEGIN se ejecuta antes del tratamiento del primer registro de datos. Se usa esencialmente para inicializar el contexto de ejecución.

### b. Secciones intermedias

Puede haber varias secciones intermedias que se ejecutarán sobre cada registro.

### c. END

La sección END se ejecuta después del tratamiento del último registro de datos. Se usa para explotar los resultados obtenidos del tratamiento de datos.

### d. Comentarios

Un comentario comienza con el carácter "#" y termina con el carácter "\n" (fin de línea).

## e. Variables

El programador puede crear sus propias variables. Una variable está definida a partir del instante en que se ha inicializado y no tiene la necesidad de estar marcada. El uso de una variable que nunca se definió tiene el valor 0 en un contexto numérico y cadena vacía en un contexto de cadena.

## f. Ejemplo

```
$ nl script1.awk
 1 # Sección BEGIN
 2 BEGIN {
 3     print "En la sección BEGIN"
 4     FS="|"
 5     nb_28=0
 6     nb_38=0
 7 }

 8 # Primera sección intermedia
 9 # Tratamiento de los clientes localizados en Madrid
10 $3 ~ /^28/ {
11     print "Madrid: " , $1
12     nb_28+=1
13 }

14 # Segunda sección intermedia
15 # Tratamiento de los clientes localizados en S.C. Tenerife
16 $3 ~ /^38/ {
17     print "S.C. Tenerife: " , $1
18     nb_38+=1
19 }

20 # Sección END
21 END {
22     print "En la sección END"
23     print "Número total de registros: " , NR
24     print "Número de clientes localizados en Madrid: " , nb_23
25     print "Número de clientes localizados en S.C. Tenerife:
" , nb_38
26 }
```

### Sección BEGIN

Modifica el valor del separador de campos e inicializa dos nuevas variables que asumen el papel de contadores.

### Secciones intermedias

Se ejecutan sobre cada registro dos secciones intermedias. Para cada registro:

- Si el código postal comienza por "28", el nombre del cliente se muestra, precedido de la palabra "Madrid". Después el contador **nb\_28** se incrementa.
- Si el código postal comienza por "38", el nombre del cliente se muestra, precedido del texto "S.C. Tenerife". Después el contador **nb\_38** se incrementa.

### Sección END

Muestra el número total de clientes (**NR**), el número total de clientes residentes en la provincia Madrid y después el número total de clientes residentes en la provincia de S.C. Tenerife (**nb\_38**).



*Resultado de la ejecución:*

```
$ awk -f script1.awk tel3.txt  
En la sección BEGIN  
Madrid: Méndez Roca, Gisela  
S.C. Tenerife: Martínez Parra, Marta  
S.C. Tenerife: Expósito Heredia, Pedro  
En la sección END  
Número total de registros: 6  
Número de clientes localizados en Madrid: 1  
Número de clientes localizados en S.C. Tenerife: 2  
$
```

# Operadores

La tabla siguiente agrupa los operadores disponibles en este lenguaje.

Operador	Aridad	Significado
<b>Operadores aritméticos</b>		
+	Binario	Suma.
-	Binario	Resta.
*	Binario	Multiplicación.
/	Binario	División.
%	Binario	Módulo.
^	Binario	Exponenciación.
++	Unario	Incremento de una variable de una unidad.
--	Unario	Decremento de una variable de una unidad.
+=	Binario	$x+=y$ es equivalente a $x=x+y$ .
-=	Binario	$x-=y$ es equivalente a $x=x-y$ .
*=	Binario	$x*=y$ es equivalente a $x=x*y$ .
/=	Binario	$x/=y$ es equivalente a $x=x/y$ .
%=	Binario	$x%=y$ es equivalente a $x=x%y$ .
^=	Binario	$x^=y$ es equivalente a $x=x^y$ .
<b>Operadores de verificaciones</b>		
<	Binario	Menor.
>	Binario	Mayor.
<=	Binario	Menor o igual.
>=	Binario	Mayor o igual.
==	Binario	Verificación de igualdad.
!=	Binario	Verificación de desigualdad.
~	Binario	Correspondencia con una expresión regular.
!~	Binario	No correspondencia con una expresión regular.
<b>Operadores lógicos</b>		
!	Unario	Negación.
&&	Binario	Y lógica.
	Binario	O lógica.
<b>Varios</b>		
=	Binario	Asignación.
e1 ? e2: e3	Ternario	La expresión global vale e2 si e1 es verdadero, e3 en caso contrario.
e1 e2 (operador espacio)	Binario	Concatenación de e1 y e2



## La función printf

**awk** ofrece la función integrada `printf`, similar a la del lenguaje C. Permite formatear los textos de salida.

```
printf ("cadena", expr1, expr2, ..., exprn)
```

`cadena` representa la cadena que se mostrará por pantalla. Puede contener formatos que serán sustituidos por el valor de las expresiones citadas a continuación. Tiene que haber tantos formatos como expresiones.

### Ejemplos de formatos comúnmente utilizados

<code>%20s</code>	Visualización de una cadena (string) de 20 posiciones (alineada a la derecha por defecto).
<code>%-20s</code>	Visualización de una cadena (string) de 20 posiciones con alineación a la izquierda.
<code>%3d</code>	Visualización de un entero (decimal) de 3 posiciones (alineación a la derecha).
<code>%03d</code>	Visualización de un entero (decimal) de 3 posiciones (alineación a la derecha) completado con 0 a la izquierda.
<code>%-3d</code>	Visualización de un entero (decimal) de 3 posiciones (alineación a la izquierda).
<code> %+3d</code>	Visualización de un entero (decimal) de 3 posiciones (alineación a la derecha) con escritura sistemática del signo (un número negativo siempre mostrará su signo).
<code>%10.2f</code>	Visualización de un número en coma flotante de 10 posiciones, 2 de las cuales son decimales.
<code> %+010.2f</code>	Visualización de un número en coma flotante de 10 posiciones, 2 de las cuales son decimales; alineación a la derecha, con escritura sistemática del signo, completado con ceros a la izquierda.

Ejemplos de uso de la función `printf` se muestran a lo largo de este capítulo.

# Redirecciones

Es posible redirigir las salidas del script hacia un archivo o hacia un comando del sistema.

## Sintaxis

instrucción > "archivo"

En la primera llamada, apertura en modo "sobrescritura", seguido de escritura. Las escrituras siguientes se realizan a continuación de la línea anterior.

instrucción >> "archivo"

En la primera llamada, apertura en modo "adición", seguido de escritura. Las escrituras siguientes se realizan a continuación de la línea anterior.

print[f] "...|"comando"

El resultado de la instrucción print se transmite a la entrada estándar del comando mediante una tubería.

## Primer ejemplo

*Apertura en modo sobrescritura:*

```
$ nl redireccion1.awk
1 BEGIN {
2   nombreach = "/tmp/arch.txt"
3   print "Línea 1" > nombreach
4   print "Línea 2" > nombreach
5   print "Línea 3" > nombreach
6   close(nombreach)
7 }
```

## Ejecución:

```
$ date > /tmp/arch.txt      # creación de un archivo no vacío
$ cat /tmp/arch.txt
sáb ene 26 14:14:32 CET 2014
$ awk -f redireccion1.awk  # Ejecución del script awk
$ cat /tmp/arch.txt        # El contenido anterior ha sido
                           # sobrescrito
Línea 1
Línea 2
Línea 3
$
```

## Segundo ejemplo

*Apertura en modo adición:*

```
$ nl redireccion2.awk
1 BEGIN {
2   nombreach = "/tmp/arch.txt"
3   print "Línea 1" >> nombreach
4   print "Línea 2" > nombreach
5   print "Línea 3" > nombreach
```

```
6   close(nombrearch)
7 }
```

*Ejecución:*

```
$ date > /tmp/arch.txt           # creación de un archivo no vacío
$ cat /tmp/arch.txt
sáb ene 26 14:15:54 CET 2014
$ awk -f redireccion2.awk        # Ejecución del script awk
$ cat /tmp/arch.txt              # El contenido anterior se ha
                                 # conservado
sáb ene 26 14:15:54 CET 2014
Línea 1
Línea 2
Línea 3
$
```

Tercer ejemplo

*Escritura en una tubería. Seleccionar las líneas impares:*

```
$ cat agenda.txt
Roberto 08020 roberto@misitio.com
Natalia 28012 natalia@misitio.com
Alejandro 46001 alejandro@misitio.com
$ awk ' NR%2 { print $1 | "sort" } END{ close("sort") }' agenda.txt
Alejandro
Roberto
$
```

*En la primera escritura en la tubería, esta se abre. Todas las órdenes **print** posteriores envían sus datos hacia la misma tubería. El comando **sort** solo se ejecuta una vez. En la sección **END**, se cierra la tubería.*

## Lectura de la línea siguiente: next

La instrucción **next** interrumpe el tratamiento de la línea actual y desencadena la lectura de la línea siguiente, en la que se aplicará el tratamiento integral.

### Ejemplo

El script **transferir1.awk** genera, a partir del archivo **tel3.txt**, una salida por pantalla que tiene el formato del archivo tratado, pero añadiendo **(0)** delante del número de teléfono de los clientes localizados en la zona horaria de la península y **(-1)** delante del número de teléfono de los clientes localizados en la zona horaria de las islas Canarias.

```
$ nl transferir1.awk
 1 BEGIN {
 2     FS="|"
 3 }

 4 $3 ~ /^(35|38)/ {
 5     printf ("%s|%s|%s|%s|(-1)%s\n", $1, $2, $3, $4, $5)
 6     # Salto al registro siguiente
 7     next
 8 }

 9 {
10     printf ("%s|%s|%s|%s|(0)%s\n", $1, $2, $3, $4, $5)
11 }
```

### Ejecución:

```
$ awk -f transferir1.awk tel3.txt
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid|(0)915.351.478
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona|
(0).932.282.177
Hernández Darín, Alberto|plaza mayor|13190|Corral de Calatrava|
(0)926/448/829
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de Calatrava|
(0)926.443.602
Martínez Parra, Marta|calle de la Santa Trinidad|38870|La Calera|
(-1)984.122.119
Expósito Heredia, Pedro|calle del castillo|38870|La Calera|
(-1)984.122.369
$
```

Líneas de la 4 a la 8: la primera sección intermedia trata los códigos postales de las islas Canarias. Cuando el tratamiento termina, la instrucción **next** reinicia el tratamiento de la siguiente línea del archivo. Esto evita la ejecución de la segunda sección intermedia (líneas de 9 a 11), que se dedica al resto de los códigos postales.

# Estructuras de control

**awk** ofrece estructuras de control que normalmente se encuentran en lenguajes de programación. La sintaxis se ha heredado del lenguaje C.

## 1. if

La parte **else** es opcional.

### Sintaxis

```
if (condición) {
    instrucción
    ...
}
else {
    instrucción    ...
}
```

Cuando solo hay una instrucción, las llaves son opcionales:

```
if (condición)
    instrucción
else
    instrucción
```

## 2. for

### Primera sintaxis

```
for (inicialización; condición; incremento) {
    instrucción
    ...
}
```

Cuando solo hay una instrucción, las llaves son opcionales.

Sintaxis equivalente:

```
inicialización
for (; condición; ) {
    instrucción
    ...
    incremento
}
```

### Segunda sintaxis

El bucle **for** también permite tratar una tabla (ver sección Tablas).

```
for (clave in tabla) {
    print clave , tabla[clave]
}
```

Cuando solo hay una instrucción, las llaves son opcionales.



### 3. While

#### Sintaxis

```
while (condición) {  
    instrucción  
    ...  
}
```

### 4. do-while

#### Sintaxis

```
do {  
    instrucción  
    ...  
} while (condición)
```

### 5. break

La palabra clave **break** permite interrumpir un bucle.

#### Principio

```
while (1) {  
    if (condición) break;      # salida del bucle  
    instrucción;  
}  
# Aquí después de break
```

### 6. continue

La palabra clave **continue** permite subir inmediatamente a la condición, sin ejecutar la continuación del bucle.

#### Principio

```
while (1) {  
    if (condición) continue; # subir inmediatamente al while  
    instrucción;              # ejecutar únicamente si la condición  
                              # es falsa  
}
```

## Finalizar un script

La instrucción `exit` permite en todo momento finalizar un script devolviendo el estado al sistema.

### Ejemplo

```
{
    if ( NF < 3) exit 1;    # Fin del script con estado falso
    . . .
    . . .
}

END{
    exit 0                # Fin del script con estado verdadero
}
```

# Tablas

## 1. Tablas indexadas con un entero

El índice de partida es elegido por el programador.

### Ejemplo

Este script inicializa un elemento de la tabla en cada nuevo registro tratado. El archivo tratado es **tel3.txt**. Cada elemento representa el nombre de un cliente. Esta tabla se indexa a partir de 1:

```
$ nl tab.awk
 1 # Sección BEGIN
 2 BEGIN {
 3     FS="|"
 4 }
 5 # Tabla que almacena los nombres de los clientes
 6 {
 7     cliente[NR]=$1
 8 }

 9 # Sección END
10 END {
11     # Visualización de la tabla
12     for (indice=1; indice <= NR; indice++)
13         printf("Cliente n° %4d => %-20s\n",indice, cliente[indice]);
14 }
$
```

Resultado de la ejecución:

```
# awk -f tab.awk tel3.txt
Cliente n°    1 => Méndez Roca, Gisela
Cliente n°    2 => Ruiz del Castillo, Marcos
Cliente n°    3 => Hernández Darín, Alberto
Cliente n°    4 => Gómez Bádenas, Josefina
Cliente n°    5 => Martínez Parra, Marta
Cliente n°    6 => Expósito Heredia, Pedro
#
```

## 2. Tablas asociativas

### a. Definición

Las tablas asociativas tienen sus elementos indexados por una cadena de caracteres. Este índice alfanumérico se conoce con el nombre de clave y el elemento correspondiente se llama valor.

### Ejemplo

El archivo **ventas.txt** contiene información acerca de las ventas de una sociedad. La información está clasificada por poblaciones. *A001:100* representa el código de un artículo y el número de ejemplares vendidos.

```
$ cat ventas.txt
Código de artículo:Número de artículos vendidos

#Madrid
```

```
A001:100
A002:300
A003:500

#Barcelona
A001:1000
A002:30
A003:5
A004:2500

#Sevilla
A001:3000
A002:20
A003:50
A004:200
```

El script **tab2.awk** calcula para cada artículo la cantidad total vendida en España, incluyendo todas las poblaciones. Este script crea una tabla cuyo índice (la clave) se representa por el código del artículo y el valor es un entero que representa el número total de artículos vendidos.

```
$ nl tab2.awk
 1 # Sección BEGIN
 2 BEGIN {
 3     FS=":"
 4 }

 5 # Tabla que almacena el nombre de los clientes
 6 NR != 1 && $0 !~ /^#/ && $0 !~ /^$/ {
 7     ventas[$1]+=$2
 8 }
 9 # Sección END
10 END {
11 # Visualización de la tabla
12     for (articulo in ventas)
13         printf("Código de artículo: %s - Total ventas: %10d\n",
articulo, ventas[articulo])
14 }
$
```

El criterio de selección (línea 6) impide tratar la primera línea del archivo, las líneas que comiencen con una "#" y las líneas vacías. Cada vez que **awk** trata un registro, la tabla **ventas** se actualiza (línea 7). Las claves de la tabla son los códigos de los artículos (\$1) y los valores representan el número de artículos vendidos (adición de \$2 al valor ya presente en la tabla). La sección END (línea 10) muestra la tabla con la ayuda de un bucle **for** (línea 12). La variable **articulo** representa la clave, y la expresión **ventas[articulo]**, el valor.

*Resultado de la ejecución:*

```
$ awk -f tab2.awk ventas.txt
Código de artículo: A001 - Total ventas:         4100
Código de artículo: A002 - Total ventas:          350
Código de artículo: A003 - Total ventas:          555
Código de artículo: A004 - Total ventas:        2700
$
```

## b. Verificar la existencia de un elemento

La palabra clave **in** permite verificar la existencia de una clave en una tabla asociativa. Esta expresión devuelve verdadero si la clave está presente, falso en caso contrario:

```
clave in tabla
```

Por lo tanto, esta expresión puede usarse como condición en una estructura de control:

```
if (clave in tabla) { ... }
```

### **c. Eliminar un elemento**

Es posible borrar un elemento de una tabla asociativa usando la sintaxis siguiente:

```
delete tabla[clave]
```

El par clave-valor se elimina.

## Los argumentos de la línea de comandos

**awk** proporciona un mecanismo que permite pasar argumentos a un script en el momento de su llamada. Las variables **ARGC** y **ARGV** se inicializan por **awk** y permiten tratar los valores pasados en la línea de comandos.

### Ejemplo

El script **agenda.awk** permite buscar información en el archivo **agenda.txt**. El usuario puede buscar la línea correspondiente al nombre de una persona (**nombre=**), a su email (**mail=**) o a su código postal (**cp=**).

```
$ cat agenda.txt
Roberto 08020 roberto@misitio.com
Natalia 28012 natalia@misitio.com
Alejandro 46001 alejandro@misitio.com
```

Llamadas al script:

```
$ awk -f agenda.awk nombre=Roberto agenda.txt
$ awk -f agenda.awk cp=08020 agenda.txt
$ awk -f agenda.awk mail='roberto@misitio.com' agenda.txt
```

Obtención de los argumentos:

```
$ nl agenda1.awk
 1  #!/bin/awk
 2  BEGIN{
 3    print "ARGC = " , ARGC
 4    for (i=0;i<ARGC;i++) {
 5      printf("ARGV[%d] = %s\n", i, ARGV[i])
 6    }
 7  }
$ awk -f agenda1.awk nombre=Roberto agenda.txt
ARGC = 3
ARGV[0] = awk
ARGV[1] = nombre=Roberto
ARGV[2] = agenda.txt
```

**awk** inicializa la tabla **ARGV** con el nombre del comando (**ARGV[0]**) y todos los valores pasados en línea de comandos. Las opciones de **awk** no se encuentran en la tabla de argumentos para no dificultar el tratamiento.

En el script siguiente, todavía no hay sección intermedia. Si se incluyera, habría un error de ejecución, ya que el valor **nombre=Roberto** se interpretará como un nombre de archivo para tratar. La solución: grabar los argumentos de línea de comandos y eliminar de la tabla **ARGV** los que no correspondan a un archivo.

### Ejemplo

```
$ nl agenda2.awk
 1  #!/bin/awk
 2  BEGIN{
 3    # Corte del argumento
 4    numCampos=split(ARGV[1],args,"=") ;
 5    opcion=args[1] ; printf("ARGV[%d] = %s\n", i, ARGV[i])
 6    valor=args[2] ;
 7    # Eliminar el argumento
```

```
8 delete ARGV[1]
9 }
```

```
10 opcion == "nombre" && $1 == valor { print $0 }
11 opcion == "cp" && $2 == valor { print $0 }
12 opcion == "mail" && $3 == valor { print $0 }
```

```
$ awk -f agenda2.awk nombre=Roberto agenda.txt
Roberto 08020 roberto@misitio.com
```

# Funciones integradas

El lenguaje **awk** dispone de funciones integradas.

## 1. Funciones que trabajan con cadenas

Función	Rol
gsub(er,reemp,[cad])	Reemplaza en la cadena "cad" cada ocurrencia correspondiente a la expresión regular "er" por la cadena "reemp". Devuelve el número de sustituciones. Por defecto, "cad" vale \$0.
index(cad1,cad2)	Devuelve la posición de la subcadena "cad2" en la cadena "cad1".
length(cad)	Devuelve la longitud de la cadena "cad".
match(cad,er)	Devuelve la posición en la cadena "cad" de la primera ocurrencia de la expresión regular "er".
split(cad,tab,sep)	Inicializa la tabla "tab" con los campos de la cadena "cad". "sep" representa el separador de campos. Devuelve el número de campos.
sprintf(fmt,e1,...,en)	Idéntico a printf, pero devuelve la cadena formateada.
sub(er,reemp,[cad])	Reemplaza en la cadena "cad" la primera ocurrencia correspondiente a la expresión regular "er" por la cadena "reemp". Devuelve el número de sustituciones. Por defecto, "cad" vale \$0.
substr(cad,pos,lg)	Devuelve la subcadena de "cad" que comienza en la posición "pos" y de longitud "lg".
tolower(cad)	Devuelve el valor de la cadena "cad" convertida en minúsculas.
toupper(cad)	Devuelve el valor de la cadena "cad" convertida en mayúsculas.

## 2. Funciones matemáticas

Función	Rol
cos(x)	Devuelve el coseno de x
exp(x)	Devuelve e elevado a x
int(x)	Devuelve el valor entero de x
log(x)	Devuelve el logaritmo natural de x
sin(x)	Devuelve el seno de x
sqrt(x)	Devuelve la raíz cuadrada de x
atan2(x,y)	Devuelve el arcotangente de y/x
rand()	Devuelve un número aleatorio tal que $0 \leq y < 1$
srand(x)	Inicializa la base de cálculo para <b>rand()</b> en función del valor x. Devuelve la base antigua.

## 3. Otras funciones



## a. La función `getline`

La función `getline` permite:

- Leer la línea siguiente del flujo de datos sin subir al comienzo del tratamiento (no como sucede con `next`).
- Leer una línea desde un archivo, de la entrada estándar o de una tubería.

Valor de retorno:

- 1 en caso de éxito.
- 0 si está al final del archivo.
- -1 en caso de error.

➤ A pesar del hecho de que `getline` esté considerada como una función, no hay que poner los paréntesis cuando se invoque.

### Sintaxis

```
getline [var]
```

Lectura de la línea siguiente del flujo.

```
getline [var] < "arch"
```

Lectura de una línea de un archivo.

```
"comando" | getline [var]
```

Lectura de una línea procedente del resultado de un comando del sistema.

La línea leída por "getline" se almacena en la variable "var" o "\$0" si no se especifica ningún nombre de variable. El nombre de archivo "-" representa la entrada estándar.

### Primer ejemplo

Lectura de la línea siguiente del flujo natural de `awk`.

A continuación, el archivo `conf.txt`, en el cual un dato puede estar escrito en diversas líneas. El final de un dato se representa por el carácter "\n". Si este está precedido de una "\", significa que la continuación del dato está contenida en la siguiente línea, situada debajo.

```
$ nl conf.txt
 1 línea1 \ ↵
 2 continuación línea1 \ ↵
 3 continuación línea1 ↵          final del primer dato
 4 línea2 ↵                       final del segundo dato
 5 línea3 \ ↵
 6 continuación línea3 ↵          final del tercer dato
```

La función del script `conf.awk` es la de reconstruir cada dato en una sola línea:

```

$ nl conf.awk
 1 {
 2   linea = $0

 3   # mientras que la línea termine con \
 4   while (linea ~ /\$/ ) {

 5     # supresión de \
 6     sub(/\$/, "", linea)

 7     # lectura de la línea siguiente del flujo
 8     getline

 9     # concatenación de las 2 líneas
10     linea = linea $0
11   }

12   # cuando la línea esté reconstruida, mostrarla
13   print linea
14 }

```

*Ejecución:*

```

$ awk -f conf.awk conf.txt
línea1 continuación línea1 continuación línea1
línea2
línea3 continuación línea3
$

```

En la línea 2, la línea leída se guarda en la variable `linea`. El bucle `while` (líneas de la 4 a la 11) provoca la lectura de las líneas siguientes, mientras la línea leída termine con un `"\"`. La línea 8 permite leer la línea siguiente del flujo de datos (el archivo **conf.txt**) y proseguir el tratamiento en la línea 10 (reconstrucción de los datos).

### Segundo ejemplo

Lectura del teclado, lectura de un archivo exterior al flujo común.

*A continuación se muestra el archivo **agenda.txt**:*

```

$ cat agenda.txt
Roberto 08020 roberto@misitio.com
Natalia 28012 natalia@misitio.com
Alejandro 46001 alejandro@misitio.com

```

El script **entrada.awk** se compone únicamente de una sección `BEGIN`. Este programa abre el archivo **agenda.txt** para buscar el nombre de la persona introducido por el teclado. En este ejemplo, el archivo **agenda.txt** se abre en la sección `BEGIN`, y no se pasa como argumento en la línea de comandos.

```

$ nl entrada.awk
 1 #! /bin/awk

 2 BEGIN{
 3   # Leer el archivo agenda.txt
 4   while (1) {
 5     # Entrada del nombre
 6     printf "Buscar el nombre: "

 7     # Si se produce error o se recibe final de archivo
(ctrl d) fin del tratamiento
 8     if ((getline nombre < "-" ) != 1) break

```

```

 9      # Si ningún nombre se introduce, nueva solicitud
        de entrada de datos
10      if (length(nombre) == 0) continue

11      # Bucle de lectura del archivo agenda
12      encontrado="false"
13      while ((getline < "agenda.txt") == 1) {
14          # Búsqueda ignorando mayúsculas o minúsculas
15          if (tolower($1) == tolower(nombre)) {
16              encontrado = "true"
17              print $0
18              break
19          }
20      }
21      if (encontrado == "false") print nombre , "no está en
la agenda."
22      }
23      print "\nBye"
24      }

```

El tratamiento está contenido dentro de un bucle infinito (línea 4). En la línea 8, se lee un nombre desde el teclado. Si el usuario ha introducido `^d` o si se produce un error de entrada, el script se para (salida del bucle `while` con `break`). En la línea 10, si el usuario presiona [Entrar], la variable `nombre` estará vacía; el script desencadenará entonces una nueva petición de entrada (`continue`). En la línea 13, hay un bucle de lectura del archivo **agenda.txt**. Si una línea se lee, se compara con el nombre introducido: si hay correspondencia, el script muestra el resultado (línea 17) y el tratamiento desencadena una nueva petición de entrada. Si el nombre no se ha encontrado, se muestra un mensaje en la línea 21.

*Ejecución del script:*

```

$ awk -f entrada.awk
Buscar el nombre: roberto           entrada de un nombre presente
                                   en la agenda
Roberto 08020 roberto@misitio.com
Buscar el nombre: alan             entrada de un nombre ausente
                                   en la agenda
alan no se encuentra en la agenda.
Buscar el nombre: ↵                pulsación de la tecla Enter
Buscar el nombre:
Buscar el nombre: ^d              fin de entrada de datos
Bye
$

```

### Tercer ejemplo

*Lectura del resultado de un comando del sistema. La función `getline` recupera la salida estándar del comando **date** y la almacena en la variable `$0`:*

```

$ nl cmdo.awk
 1 BEGIN {
 2     "date '+%d-%m-%Y'" | getline
 3     print $0
 4 }
$ awk -f cmdo.awk
26-01-2015
$

```

## **b. La función close**

La función `close` permite cerrar un archivo o una tubería de comunicación. Si la llamada a esta

función se omite, los recursos son liberados al finalizar el script.

La orden de cierre es interesante en los casos siguientes:

- Poder reubicarse al comienzo de un archivo dentro del mismo proceso (cierre y después apertura).
- Cerrar una tubería de comunicación para volver a utilizarla.
- Liberar los recursos como y cuando sea necesario (el sistema limita los procesos en lo que concierne al número de archivos/tuberías abiertas simultáneamente).

## Sintaxis

```
close("archivo")
close("comando")
```

## Ejemplos

*Cierre indispensable para poder releer el archivo (la apertura coloca el indicador de posición al comienzo del archivo):*

```
$ nl close_arch.awk
 1 BEGIN {
 2   nombreach = "/tmp/arch.txt"
 3   print "Linea 1" > nombreach
 4   print "Linea 2" > nombreach
 5   print "Linea 3" > nombreach
 6
 7   # Cierre del archivo para poder releerlo
 8   close(nombreach)
 9
10   while ((getline < nombreach) == 1){
11     print          # muestra $0 por defecto
12   }
13 }
```

## c. La función system

La función `system` permite ejecutar un comando del sistema operativo.

## Sintaxis

```
system("comando")
```

La función devuelve el estado devuelto por el comando.


## Ejemplo

*Características del archivo **agenda.txt**:*

```
$ awk ' BEGIN{ system("ls -l agenda.txt") }'
-rw-r--r-- 1 cristina cristina 94 ene 26 11:19 agenda.txt
```

*Lo mismo, pero usando una variable `awk` inicializada anteriormente:*

```
$ awk 'BEGIN{ arch="agenda.txt"; system("ls -l " arch) }'
-rw-r--r-- 1 cristina cristina 94 ene 26 11:19 agenda.txt
$
```

 Recordatorio: el espacio es el operador de concatenación.



# Funciones de usuario

## Sintaxis

Una función puede tener de 0 a n argumentos y devolver 1 valor explícito. Las funciones pueden definirse antes o después de su llamada.

Definición de una función:

```
function nombre_funcion (param1, param2 ..., paramn) {  
    return valor  
}
```

- Los parámetros (param1, param2 ..., paramn) son variables locales. Cualquier otra variable definida en la función es global.

Llamada a una función:

```
valor_devuelto=nombre_funcion(val1, val2, ..., valn)
```

- No tiene que haber espacios entre el nombre de la función y el paréntesis de apertura.

## Ejemplo

El script **transferir.awk** genera, a partir del archivo **tel3.txt**, una salida por pantalla que toma el formato del archivo tratado, pero añadiendo "(0)" delante del número de teléfono de los clientes localizados en la zona horaria de Madrid y "(-1)" delante del número de teléfono de los clientes localizados en las dos provincias de las islas Canarias. La función *transferir* recibe como parámetros el código postal y el número de teléfono actual, y devuelve el teléfono con la zona horaria del cliente:

```
$ nl transferir.awk  
1 # Función que determina la zona horaria del cliente  
2 function transferir (cp , tel) {  
3  
4 # Zona horaria canaria  
5 if ( cp ~ /^(3[58])/ )  
6     zona_horaria="(-1)"  
7 else # Zona horaria peninsular  
8     zona_horaria="(0)"  
9  
10 # Los 2 valores se concatenan  
11 return zona_horaria tel  
12 }  
  
13 # Sección BEGIN  
14 BEGIN {  
15     FS="|"  
16 }  
  
17 # Sección intermedia  
18 {  
19     newtel=transferir($3,$5)  
20     printf("%s|%s|%s|%s|%s\n", $1, $2, $3, $4, newtel);  
21 }
```

Resultado de la ejecución:

```
$ awk -f transferir.awk tel3.txt
```

```
Méndez Roca, Gisela|calle Ruiseñor|28023|Madrid| (0) 915.351.478
```

```
Ruiz del Castillo, Marcos|calle Balmes|08020|Barcelona| (0) .932.282.177
```

```
Hernández Darín, Alberto|plaza mayor|13190|Corral de Calatrava| (0) 926/448/829
```

```
Gómez Bádenas, Josefina|calle Sagasta|13190|Corral de  
Calatrava| (0) 926.443.602
```

```
Martínez Parra, Marta|calle de la Santa Trinidad|38870|
```

```
La Calera| (-1) 984.122.119
```

```
Expósito Heredia, Pedro|calle del castillo|38870|La Calera| (-1) 984.122.369
```

```
$
```

# Ejercicios

Los archivos proporcionados para los ejercicios están disponibles en la carpeta dedicada al capítulo, en el directorio **Ejercicios/Archivos**.

## 1. awk en línea de comandos

### a. Ejercicio 1: awk y otros filtros

Comandos filtro útiles: **awk**, **grep** (ver capítulo Los comandos filtro), **sed** (ver capítulo El comando **sed**). Otro comando útil: **file**.

Muestre los nombres de los archivos de texto del directorio **/etc**.

Ejemplo de resultado

```
adjtime
aliases
asound.conf
auto.master
auto.misc
. . .
```

### b. Ejercicio 2: criterios de selección

1. En su directorio actual, muestre las características de los archivos cuyo nombre comience con un punto (solo estos).

Ejemplo de resultado

```
drwxr-xr-x. 24 cristina cristina 4096  3 feb 12:26 .
drwxr-xr-x. 11 root      root    4096 27 ene 14:06 ..
-rw-----.  1 cristina cristina 14752 22 ene. 12:40 .bash_history
```

2. En su directorio actual, muestre los nombres de los archivos que comienzan con un punto, salvo "." y "..".

Ejemplo de resultado

```
.bash_history
.bash_logout
.bash_profile
.bashrc
```

### c. Ejercicio 3: criterios de selección, visualización de campos, secciones BEGIN y END

A partir del archivo **php.ini** proporcionado:

1. Muestre las líneas que no comiencen por ";" y que terminen en On u Off.

Ejemplo de resultado

```
engine = On
short_open_tag = Off
asp_tags = Off
zlib.output_compression = Off
implicit_flush = Off
```



- Mejore la visualización.

Ejemplo de resultado

```
engine                               On
short_open_tag                       Off
asp_tags                             Off
zlib.output_compression              Off
implicit_flush                      Off
zend.enable_gc                      On
. . .
```

- Recupere el comando anterior y muestre el número de directivas encontradas en el total de las líneas.

Ejemplo de resultado

```
. . .
mssql.secure_connection              Off
tidy.clean_output                   Off
42 directivas encontradas en 1974 líneas
```

## 2. Scripts awk

### a. Ejercicio 4: funciones

Recupere el archivo **Ficha.php** (extracto):

```
$ nl Ficha.php
 1 <?php
   . . .
10 class Ficha
11 {
12     /**
13      * @var integer
14      *
15      * @ORM\Column(name="fic_id", type="integer", nullable=false)
16      * @ORM\Id
17      * @ORM\GeneratedValue(strategy="IDENTITY")
18      */
19     private $ficId;
20
21     /**
22      * @var string
23      *
24      * @ORM\Column(name="fic_description_faits_consequences",
25      type="text", nullable=false)
26      */
27     private $ficDescriptionFaitsConsequences;
28
29     /**
30      * @var string
31      *
32      * @ORM\Column(name="fic_action_immediate", type="text",
33      nullable=true)
34      */
35     private $ficActionImmediate;
```

Prepare un script **delprefix.awk** que efectúe las operaciones siguientes:

- Las líneas que comiencen por **private** deben modificarse:
  - Las tres primeras letras del nombre de la variable (prefijo) deben eliminarse, sea cual sea su valor (en nuestro caso, el prefijo es "fic", pero podría ser cualquier cosa).
  - La letra que siga al signo \$ debe estar en minúsculas.

Después de ejecutar el script, el archivo deberá verse como sigue (extracto):

```

1 <?php
  . . .
10 class Ficha
11 {
12     /**
13      * @var integer
14      *
15      * @ORM\Column(name="fic_id", type="integer", nullable=false)
16      * @ORM\Id
17      * @ORM\GeneratedValue(strategy="IDENTITY")
18      */
19     private $id;

20     /**
21      * @var string
22      *
23      * @ORM\Column(name="fic_description_faits_consequences",
24      type="text", nullable=false)
25      */
26     private $descriptionFaitsConsequences;

27     /**
28      * @var string
29      *
30      * @ORM\Column(name="fic_action_immediate", type="text",
31      nullable=true)
32      */
33     private $actionImmediate;

```

## b. Ejercicio 5: análisis de un archivo de log

Funcionalidades implementadas: secciones **BEGIN**, **END**, argumentos de la línea de comandos, tablas, funciones.

Sea el archivo de log de sistema **error.log**:

```

$ cat error.log
[Mon Sep 30 09:33:00 2013] [notice] Apache/2.2 (Unix)
(Red-Hat/Linux) mod_python/2.7.6
Python/1.5.2 mod_ssl/2.8.4
OpenSSL/0.9.6b DAV/1.0.2 PHP/4.0.6 mod_perl/1.24_01
mod_throttle/3.1.2 configured -- resuming normal operations
[Mon Sep 30 09:33:00 2013] [notice] suEXEC mechanism enabled
(wrapper: /usr/sbin/suexec)
[Mon Sep 30 18:35:34 2013] [notice] caught SIGTERM, shutting down
[Tue Oct 1 10:06:46 2013] [alert] httpd: Could not determine
the server's fully qualified domain name, using 10.0.0.66
for ServerName
[Tue Oct 1 10:06:46 2013] [notice]
Apache/2.2 (Unix) (Red-Hat/Linux) mod_python/2.7.6
Python/1.5.2 mod_ssl/2.8.4
OpenSSL/0.9.6b DAV/1.0.2 PHP/4.0.6 mod_perl/1.24_01
mod_throttle/3.1.2 configured -- resuming normal operations
[Tue Oct 1 10:06:46 2013] [notice] suEXEC mechanism enabled
(wrapper: /usr/sbin/suexec)

```

```
[Tue Oct 1 10:12:51 2013] [notice] SIGHUP received.
Attempting to restart
[Tue Oct 1 10:12:52 2013] [alert] httpd: Could not determine
the server's fully qualified domain name, using 10.0.0.66
for ServerName
[Tue Oct 1 10:12:52 2013] [notice] Apache/2.2 (Unix) (Red-Hat/Linux)
mod_python/2.7.6
Python/1.5.2 mod_ssl/2.8.4
OpenSSL/0.9.6b DAV/1.0.2 PHP/4.0.6 mod_perl/1.24_01
mod_throttle/3.1.2 configured -- resuming normal operations
[Tue Oct 1 10:12:52 2013] [notice] suEXEC mechanism enabled
(wrapper: /usr/sbin/suexec)
```

Prepare un script **error\_log.awk** que permita buscar un tipo de mensaje (**alert** o **notice**) y mostrarlo por pantalla de una forma que resulte más fácil de leer. El tipo de mensaje que se ha de buscar se pasa como argumento en la línea de comandos.

### Ejemplo de ejecución del script

```
$ awk -f error_log.awk alert error.log
LISTA DE MENSAJES DE TIPO : alert
Date : Tue Oct 1 10:06:46 2013
httpd: Could not determine the server's fully qualified
domain name, using 10.0.0.66 for ServerName
Date : Tue Oct 1 10:12:52 2013
httpd: Could not determine the server's fully qualified
domain name, using 10.0.0.66 for ServerName
$
```

## c. Ejercicio 6: generación de un archivo de etiquetas

Funcionalidades implementadas: secciones **BEGIN**, **END**, argumentos de la línea de comandos, tablas, funciones awk.

Recupere el archivo **contactos.txt**:

```
$ cat contactos.txt
Corbalán, Marina
        08003 Barcelona
        933.221.506

Romero, Roberto
        48002 Bilbao
        944.301.265

García, Esteban
        46002 Valencia
        962.318.376

Beltrán, Francisco
        35001 Las Palmas de G.C.
        928.161.827

Revuelta, Carmelo
        35003 Las Palmas de G.C.
        928.532.111
```

Prepare un script **etiq.awk** que deberá devolver el archivo **contactos.txt** en forma de etiquetas. A continuación, el resultado que deberá proporcionar el script:

```
$ awk -f etiq.awk contactos.txt
```

Corbalán, Marina  
08003 Barcelona  
933.221.506

Romero, Roberto  
48002 Bilbao  
944.301.265

García, Esteban  
46002 Valencia  
962.318.376

Beltrán, Francisco  
35001 Las Palmas de G.C.  
928.161.827

Revuelta, Carmelo  
35003 Las Palmas de G.C.  
928.532.111

## Introducción

Este capítulo presenta los principales comandos filtro de Unix. Estos comandos tratan un flujo de datos de la entrada estándar o contenidos en un archivo. Pueden usarse de forma independiente o situados detrás de una tubería de comunicaciones. Los filtros se utilizan en la mayoría de los casos del mismo modo (aunque puede haber excepciones). Las principales opciones de cada comando se presentarán en este capítulo.

## Sintaxis de llamada a comandos filtro

```
comando_filtro -opciones arch1 arch2 ...  
comando_filtro -opciones -  
comando_filtro -opciones < archivo  
comando | comando_filtro -opciones
```

# Visualización de datos

## 1. Consulta de datos, creación de archivos: cat

Junto con la visualización de archivos, a continuación se muestran otros ejemplos de uso del comando **cat**.

### Sintaxis

```
cat [ opciones ] [ archivo ... ]
```

#### Principales opciones:

- e Materializa los finales de línea con un carácter \$.
- t Materializa las tabulaciones con un carácter ^I.
- v Materializa los caracteres no imprimibles.

### Ejemplos

Cuando el comando **cat** no recibe un archivo como argumento, lee de la entrada estándar (teclado). En este caso, los datos leídos del teclado se redirigen al archivo **f1**, lo que tiene como efecto la creación del archivo si este no existe o su sobrescritura si existe.

```
$ cat > f1
Línea1
Línea2
^d
$ cat f1
Línea1
Línea2
```

El archivo **f3** es la concatenación de **f1** y **f2**:

```
$ cat f1
Línea1
Línea2
$ cat f2
Línea3
Línea4
$ cat f1 f2 > f3

$ cat f3
Línea1
Línea2
Línea3
Línea4
```

La opción **-t** permite visualizar las tabulaciones. Estas se materializan con el carácter ^I:

```
$ cat f4
palabra1 palabra2
$ cat -t f4
palabra1^Ipalabra2
```

La opción **-e** permite visualizar los finales de línea. Estos aparecen con la forma del carácter \$:

```
$ cat -e f4
palabra1  palabra2$
```

La opción `-v` permite mostrar de forma visible los caracteres no imprimibles. Un ejemplo de uso se dio en el capítulo Configuración del entorno de trabajo - Histórico de comandos, que hacía referencia a la reutilización de un comando en ksh.

*Mostrar los alias permitiendo usar el recordatorio de comandos con las flechas del teclado:*

```
$ alias | cat -v
__A=^P
__B=^N
__C=^F
__D=^B
...
$
```

## 2. Valor de los bytes de un flujo de datos: od

El comando `od` permite visualizar el valor de cada byte de un archivo de texto o binario. La opción `-c` permite solicitar una interpretación byte a byte. Así es posible visualizar de manera exacta el contenido del archivo, detectar la colocación de espacios, tabulaciones y saltos de línea (CR LF o LF), etc.

### Ejemplos

*El comando `cat` no nos permite saber si el archivo contiene espacios o tabulaciones. Con `od`, las tabulaciones se representan por `\t`, los espacios por un espacio, el carácter LF se representa por `\n`:*

```
$ cat arch_od.txt
Isabel 20
Sofia 12

$
$ od -c arch_od.txt
0000000 I s a b e l \t 2 0 \n S o f i a
0000020          1 2 \n \n
0000027
$
```

*A continuación, el mismo archivo en formato DOS: los finales de línea CR LF se materializan con `\r\n`:*

```
$ od -c fic_od.txt
0000000 I s a b e l \t 2 0 \r \n S o f i a
0000020          1 2 \r \n
0000030

$
```

La columna de la izquierda representa la posición del primer carácter de la línea en el archivo, expresada en octal. Esta información no forma parte del contenido del archivo.

## 3. Filtrado de líneas: grep

El comando `grep` busca una cadena de caracteres en uno o varios archivos de texto y muestra por pantalla las líneas que contienen dicha cadena. La cadena buscada se representa mediante una expresión regular básica (por defecto) o extendida (opción `-E`). Este párrafo presenta las principales opciones del comando `grep`. Las expresiones regulares y su uso con `grep` se han expuesto en el capítulo Expresiones regulares.



➤ Para beneficiarse de todas las opciones de **grep** en los sistemas Solaris, se recomienda usar el comando `/usr/xpg4/bin/grep` como alternativa a `/usr/bin/grep`.

## Sintaxis

```
grep [ opciones ] expreg [ archivo ... ]
grep [ opciones ] -e expreg1 -e expreg2 [ archivo ... ]
grep [ opciones ] -f archivo_expreg [ archivo ... ]
```

### Principales opciones:

<code>-c</code>	Mostrar el número de líneas encontradas.
<code>-eexpreg</code>	Permite especificar múltiples expresiones regulares.
<code>-E</code>	La expresión regular se interpretará como una expresión regular extendida.
<code>-farchivo_expreg</code>	Las expresiones regulares tienen que leerse del archivoarchivo_expreg.
<code>-F</code>	El criterio no debe interpretarse como una expresión regular.
<code>-i</code>	Búsqueda no sensible a mayúsculas/minúsculas.
<code>-l</code>	Mostrar únicamente los nombres de archivos que contengan " <i>expreg</i> ".
<code>-n</code>	Numerar las líneas encontradas.
<code>-q</code>	Modo silencioso.
<code>-v</code>	Buscar las líneas que no contengan " <i>expreg</i> ".
<code>-w</code>	La línea contiene al menos una palabra que se corresponde con " <i>expreg</i> ".
<code>-x</code>	La línea debe corresponderse exactamente con " <i>expreg</i> ".

### Ejemplos

A continuación se muestra el archivo **tel.txt** usado en los ejemplos:

```
$ cat tel.txt
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|
Barcelona|932.282.177
Hernández Darín, Alberto|plaza mayor 15/17|13190|Corral de
Calatrava|926/448/829
Expósito Heredia, Pedro|calle del castillo 3|38870|La Calera|
984.122.369
Puente|callejón del Puerto|46001|Valencia|
46001
```

Mostrar las líneas que contengan la cadena 46001:

```
$ grep 46001 tel.txt
Puente|callejón del Puerto|46001|Valencia|
46001
```

Uso de las expresiones regulares extendidas (`-E`) y hacer una búsqueda no sensible a mayúsculas/minúsculas (`-i`):

```
$ grep -E -i '(madrid|valencia)' tel.txt
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
Puente|callejón del Puerto|46001|Valencia|
```

*Mostrar las líneas que no terminen con la cifra 8:*

```
$ grep -v '8$' tel.txt
Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|Barcelona|
932.282.177
Hernández Darín, Alberto|plaza mayor 15/17|13190|Corral de
Calatrava|926/448/829
Expósito Heredia, Pedro|calle del castillo 3|38870|La Calera|
984.122.369
Puente|callejón del Puerto|46001|Valencia|
46001
```

*Buscar las líneas que comiencen por "R" o acaben con "8":*

```
$ grep -e '^R' -e '8$' tel.txt
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|
Barcelona|932.282.177
```

*Buscar las líneas que terminen con "9" y mostrarlas precedidas de su número:*

```
$ grep -n '9$' tel.txt
3:Hernández Darín, Alberto|plaza mayor 15/17|13190|Corral de
Calatrava|926/448/829
4:Expósito Heredia, Pedro|calle del castillo 3|38870|La Calera|
984.122.369
```

*El modo silencioso: solamente el estado de retorno indica si la búsqueda tiene resultados:*

```
$ grep -q '9$' tel.txt
$ echo $?
0
```

*Mostrar las líneas correspondientes al criterio de búsqueda:*

```
$ grep -c '9$' tel.txt
2
```

*Líneas que contengan la palabra "calle" (la línea que contiene "callejón" no aparece por pantalla):*

```
$ grep -w calle tel.txt
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|Barcelona|
932.282.177
Expósito Heredia, Pedro|calle del castillo 3|38870|La Calera|
984.122.369
```

*Líneas que contengan por lo menos dos caracteres:*

```
$ grep '..' tel.txt
```

```
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|Barcelona|
932.282.177
Hernández Darín, Alberto|plaza mayor 15/17|13190|Corral de
Calatrava|926/448/829
Expósito Heredia, Pedro|calle del castillo 3|38870|La Calera|
984.122.369
Puente|callejón del Puerto|46001|Valencia|
46001
```

*No interpretar el criterio de búsqueda como una expresión regular:*

```
$ grep -F '..' tel.txt
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
```

*Usar una expresión regular extendida (-E) y buscar las líneas que contengan solamente cifras (-x).*

```
$ grep -E -x '[0-9]+' tel.txt
es lo mismo que
$ grep -E '^[0-9]+$' tel.txt
46001
```

*Leer los criterios de búsqueda desde un archivo:*

```
$ cat rech.er
9$
\.\.
$
$ grep -f rech.er tel.txt
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
Hernández Darín, Alberto|plaza mayor 15/17|13190|Corral de
Calatrava|926/448/829
Expósito Heredia, Pedro|calle del castillo 3|38870|La Calera|
984.122.369
```

*Uso de **grep** detrás de una tubería para buscar el proceso **httpd**:*

```
$ ps -ef | grep httpd
root      2455      1  0 Nov10 ?           00:00:01 /usr/sbin/httpd
cristina 17804 17775  0 22:14 pts/1     00:00:00 grep httpd
apache    21302  2455  0 Dic09 ?           00:00:00 /usr/sbin/httpd
apache    21303  2455  0 Dic09 ?           00:00:00 /usr/sbin/httpd
```

## 4. Últimas líneas de un flujo de datos: tail

El comando **tail** permite mostrar las "n" últimas líneas de un flujo de datos. Por ejemplo, este comando es muy práctico para visualizar las últimas líneas escritas en un archivo de log.

### Sintaxis

```
tail [ opciones ] [ archivo ... ]
```

#### **Principales opciones:**

- f El comando muestra en tiempo real las adiciones al final del archivo.
- n Muestra las *n* últimas líneas (10 líneas si la opción no se informa).

## Ejemplos

Mostrar las dos últimas líneas de un archivo de log de Apache, llamado en este caso **access\_log**, que registra todos los accesos a un sitio web:

```
# tail -2 /etc/httpd/logs/access_log

xx.xx.xx.xx - - [16/Dic/2013:18:26:02 +0100] "GET /images/filieres/
filiere_java_r10_c12_f2.gif HTTP/1.1" 304 - "http://www.ociensa.com/
formation.php?filiere=java&cours=uml" "Mozilla/4.0
(compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322;
InfoPath.1)"

xx.xx.xx.xx - - [16/Dic/2013:18:26:02 +0100] "GET /images/filieres/
filiere_java_r13_c5_f2.gif HTTP/1.1" 404 361
"http://www.ociensa.com/
formation.php?filiere=java&cours=uml" "Mozilla/4.0
(compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322;
InfoPath.1)"
#
```

*Vigilancia del archivo en tiempo real: una vez el archivo ha crecido, el comando muestra las últimas líneas añadidas. Para finalizar su ejecución, usar `^c`.*

```
# tail -f /etc/httpd/logs/access_log

. . .

. . .

^c
```

## 5. Primeras líneas de un flujo de datos: head

El comando **head** permite visualizar las "n" primeras líneas de un flujo de datos. Sin opción, el comando muestra las 10 primeras líneas.

### Sintaxis

```
head [ -n ] [ archivo ... ]
```

#### **Principales opciones:**

-n            Muestra las *n* primeras líneas (10 líneas por defecto).

#### Ejemplo

Mostrar la primera línea devuelta por el comando **od**: los primeros bytes del archivo **export.pdf** indican que se trata de un formato PDF (información usada por el comando **file**, que indica el tipo de contenido de un archivo).

```
$ od -c export.pdf | head -1
0000000  %   P   D   F   -   1   .   4   \r   % 342 343 317 323
\r  \n
$ file export.pdf
export.pdf: PDF document, version 1.4
$
```

## 6. Duplicación de la salida estándar: tee

El comando **tee** recupera un flujo de datos de su entrada estándar, lo envía a un archivo pasado como argumento y a su salida estándar. Esto permite tener a la vez el resultado en pantalla y en un archivo.

## Sintaxis

comando | tee [ opciones ] archivo

### Principales opciones:

-a            Añadir al final de "archivo" si este ya existe.

### Ejemplo

A partir del archivo de log **access\_log**, recuperar las líneas que se registraron el 14 de diciembre de 2013, mostrarlas por pantalla y conservarlas en un archivo para su explotación posterior:

```
$ grep '14/Dic/2013' /etc/httpd/logs/access_log | tee /tmp/today
xx.xx.xx.xx - - [14/Dic/2013:00:03:14 +0100] "GET /ressources.php
HTTP/1.0" 200 4845 "-" "Mozilla/5.0 (compatible;
Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp)"
. . .
$ head -1 /tmp/today
xx.xx.xx.xx - - [14/Dic/2013:00:03:14 +0100] "GET /ressources.php
HTTP/1.0" 200 4845 "-" "Mozilla/5.0 (compatible;
Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp)"
$
```

## 7. Numeración de líneas: nl

El comando **nl** permite mostrar un archivo de texto precediendo cada línea con su número. Por defecto, las líneas vacías no se numeran.

## Sintaxis

nl [ opciones ] [ archivo ... ]

### Principales opciones:

-b a            Numerar todas las líneas.

-b t            No numerar las líneas vacías (por defecto).

### Ejemplo

Las líneas en blanco (simbolizadas por ↵ ) no se numeran por defecto:

```
$ nl arch.txt
 1 maría                    francés                    18
 2 francisco                historia                   12
  ↵
 3 maría                    francés                    11
 4 cristina                 aritmética                12
  ↵
  ↵
 5 francisco                historia                    8
```

Numerar todas las líneas:

```
$ nl -ba fic.txt
 1  maría          francés          18
 2  francisco     historia        12
 3  ↵
 4  maría          francés          11
 5  cristina      aritmética      12
 6  ↵
 7  ↵
 8  francisco     historia         8
```

## 8. Presentación de un flujo de datos: pr

El comando **pr** muestra un archivo de texto formateado. Por defecto, cada página ocupa 66 líneas, de las cuales 5 pertenecen a la cabecera y 5 al pie de página.

### Sintaxis

```
pr [ opciones ] [ archivo ... ]
```

#### **Algunas opciones:**

- c *num*      Disposición en *num* columnas.
- htexto      Escribir un título de entrada.
- l *num*      Número de líneas de una página.

### Ejemplos

Formateo simple del archivo **notas.txt**:

```
$ pr notas.txt | more
2014-01-22 11:03          notas.txt          Page 1

Roberto          12
Simón            20
Román            15
Natalia          13
Silvia           14
Nazareno         20
Ernesto          15
Antonio          16
Marc             8
Juan Marcos      18
Paula            11
María            12
Anastasia        7
Cristina         10
Agustín          19
Tomás            14
Nicolás          13
Mohamed          15
. . .
```

Otro ejemplo de formateo: 15 líneas por página, disposición en 2 columnas y adición de un título:

\$ pr -l 15 -c2 -h "Publicación de notas" notas.txt

2014-04-06 11:42

Publicación de notas

Página 1

Roberto	12	Nazareno	20
Simón	20	Ernesto	15
Román	15	Antonio	16
Natalia	13	Marc	8
Silvia	14	Juan Marcos	18

2014-04-06 11:42

Publicación de notas

Página 2

Paula	11	Agustín	19
María	12	Tomás	14
Anastasia	7	Nicolás	13
Cristina	10	Mohamed	15

\$

# Tratamiento de datos

## 1. Recuento de líneas, de palabras y caracteres: wc

El comando **wc** (word count) cuenta el número de líneas, de palabras y de caracteres.

### Sintaxis

```
wc [ opciones ] [ archivo ... ]
```

#### Principales opciones:

- l Contar el número de líneas.
- w Contar el número de palabras.
- c Contar el número de bytes.
- m Contar el número de caracteres.
- C Idéntico a -m.

### Ejemplos

Número de líneas, palabras y caracteres del archivo **notas.txt**:

```
$ wc notas.txt
  5   15  245 notas.txt
```

Número de líneas únicamente:

```
$ wc -l notas.txt
  5 notas.txt
```

Número de caracteres contenidos en un nombre de archivo introducido mediante el teclado (atención al salto de línea añadido por el comando **echo**):

```
$ read nombreach
documento.xls
^d
$ echo "$nombreach\c" | wc -c      (Linux: $ echo -n "$nombreach"
| wc -c )
13
$
```

A continuación se muestra un archivo codificado en UTF-8 (los caracteres acentuados y la letra ñ se codifican en 2 bytes), manipulado en un sistema cuya codificación es también UTF-8:

```
$ echo $LANG
es_ES.utf8
```

El archivo contiene 1 carácter acentuado y la letra ñ:

```
$ cat utf8.txt
cáñamo
```

El comando **od** nos muestra que los caracteres acentuados y las ñ se codifican en 2 bytes:



```
$ od -c utf8.txt
0000000  c 303 241 303 250  a  m  o  \n
0000011
```

Número de bytes del archivo:

```
$ wc -c utf8.txt
9 utf8.txt
```

Número de caracteres del archivo:

```
$ wc -m utf8.txt
7 utf8.txt
```

➤ Las variables de entorno que definen la codificación son LC\_ALL, LC\_CTYPE y LANG. Estas se transmiten a los comandos a través del shell (tienen que exportarse) y se evalúan en este orden: el primer valor leído es el que se tiene en cuenta.

Obtener la lista de valores de localización (valor ligado al país y al idioma) disponibles en el sistema:

```
$ locale -a | grep es_ES
es_ES
es_ES@euro
es_ES.iso88591
es_ES.iso885915@euro
es_ES.utf8
$
```

## 2. Extracción de caracteres: cut

El comando **cut** sirve para recuperar (cortar) caracteres o campos de una línea.

### Sintaxis

Cortar por caracteres:

\$ cut -c3	[ archivo ... ]	El tercer carácter.
\$ cut -c3-5	[ archivo ... ]	Del tercer al quinto carácter.
\$ cut -c-3	[ archivo ... ]	Hasta el tercer carácter.
\$ cut -c3-	[ archivo ... ]	A partir del tercer carácter.
\$ cut -c3,10	[ archivo ... ]	El tercer y el décimo carácter.

Cortar por campos:

\$ cut -dsep -f3	[ archivo ... ]	El tercer campo.
\$ cut -dsep -f3-5	[ archivo ... ]	Del tercer al quinto campo.
\$ cut -dsep -f-3	[ archivo ... ]	A partir del tercer campo.
\$ cut -dsep -f3-	[ archivo ... ]	A partir del tercer campo.
\$ cut -dsep -f3,10	[ archivo ... ]	El tercer y el décimo campo.

La opción **-d** permite definir el carácter separador de campos. El carácter separador es por defecto la tabulación.

## Ejemplos

Cortar las dos primeras cifras de un código postal:

```
$ echo 89000 | cut -c1-2
89
```

Mostrar el primer, el tercer y el sexto campo de las 3 últimas líneas del archivo **/etc/passwd**:

```
$ tail -3 /etc/passwd | cut -d: -f1,3,6
crisrina:505:/home/cristina
svn:506:/home/svn
sebastian:507:/home/sebastian
```

Si el carácter separador es un carácter especial del shell, hay que protegerlo:

```
$ linea="Newton|Cristina|London"
$ echo $linea | cut -d'|' -f1
Newton
```

## 3. Ordenación de datos: sort

El comando **sort** permite ordenar las líneas de un flujo de datos tipo texto.

### Sintaxis

```
sort [ opciones ] -k campo[.car][opciones],campo[.car][opciones]
[ archivo ... ]
```

#### **Principales opciones:**

- b Opción que se usará cuando el criterio de ordenación sea alfanumérico y los campos estén separados por una serie de espacios/tabulaciones variable.
- k campo[.car] [opciones], campo[.car] [opciones] Especificar el/los campo/s que se tendrá/n en cuenta como criterio de ordenación. El criterio de ordenación puede comenzar o acabar en una determinada posición de carácter del campo.
- n Precisar que el criterio de ordenación debe interpretarse como un valor numérico y no como una cadena de caracteres.
- r Solicitar una ordenación decreciente.
- t *separador* Definir el carácter separador de campos (espacio por defecto).

## Ejemplos

A continuación se muestra un archivo basado en el archivo **/etc/passwd**, que se ordenará con el comando **sort**.

```
$ cat passwd.txt
root:x:0:0:root:/root:/bin/bash
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
crisrina:x:505:508::/home/cristina:/bin/bash
daemon:x:2:2:daemon:/sbin:/sbin/nologin
oliverio:x:502:502::/home/oliverio:/bin/bash
```

Si no se especifican opciones, el comando **sort** realiza una ordenación lexicográfica:

```
$ sort passwd.txt
crystina:x:505:508::/home/cristina:/bin/bash
daemon:x:2:2:daemon:/sbin:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
oliverio:x:502:502::/home/oliverio:/bin/bash
root:x:0:0:root:/root:/bin/bash
```

Por defecto, el criterio de ordenación se considera como una cadena alfanumérica. La opción `-k` permite especificar las columnas que forman parte de la composición del criterio de ordenación (índice del campo de inicio y del campo final incluidos).

### Ejemplos

*Ordenación basándose en el tercer campo (UID) y uso de `cut` para aislar el criterio de búsqueda. El criterio de ordenación se trata como una cadena de caracteres:*

```
$ sort -t: -k 3,3 passwd.txt | cut -d: -f3
0
14
2
502
505
8
$
```

*Para que el criterio de ordenación sea tratado de manera numérica, hay que incluir la opción `-n`:*

```
$ sort -t: -k 3n,3 passwd.txt | cut -d: -f3
0
2
8
14
502
505
$
```

*Ejemplo de criterio de ordenación alfanumérico en el campo 2. Sin la opción `-b`, la ordenación se realiza de forma incorrecta:*

```
$ cat notas.txt
crystina          aritmética      12
valeria          francés         18
francisco        historia        12
valeria          francés         11
francisco        historia        8

$ sort -k 2,2 notas.txt
valeria          francés         11
valeria          francés         18
crystina          aritmética      12
francisco        historia        12
francisco        historia        8

$ sort -k 2b,2 notas.txt
crystina          aritmética      12
valeria          francés         11
valeria          francés         18
francisco        historia        12
francisco        historia        8
```

Uso de dos criterios de ordenación: ordenación por la asignatura (valor alfanumérico), después por las notas (valor numérico):

```
$ sort -k 2b,2 -k 3n,3 notas.txt
crístina                aritmética                12
valeria                 francés                   11
valeria                 francés                   18
francisco               historia                   8
francisco               historia                   12
```

## 4. paste

El comando **paste** concatena las líneas de los archivos pasados como argumento: las líneas de índice "n" de cada archivo se concatenan. Con la opción **-s**, cada archivo se trata de forma independiente y se obtendrán todas sus líneas concatenadas en una sola. Los elementos concatenados están separados por una tabulación.

### Sintaxis

```
paste [ opciones ] [ archivo ... ]
```

#### Principales opciones:

- s Concatenar todas las líneas en una sola.
- d listasep Los caracteres citados en listasep serán usados para separar los campos de salida.

#### Ejemplo

A continuación se muestran dos archivos de notas de alumnos. Los alumnos siempre se citan en el mismo orden:

```
$ cat francés.txt
Cristina      12
Francisco     8
Valeria       18
$
$ cat historia.txt
Cristina      11
Francisco     10
Valeria       14
$
```

Colocar las notas en la misma línea:

```
$ paste francés.txt historia.txt
Cristina      12      Cristina      11
Francisco     8       Francisco     10
Valeria       18      Valeria       14
$
```

Eliminar la tercera columna:

```
$ paste francés.txt historia.txt | awk ' {printf("%-10s%5d%5d\n",
$1,$2 , $4)} '
Cristina      12      11
Francisco     8       10
Valeria       18      14
```

```
$
```

Con la opción `-s`, concatenar todas las líneas en una sola:

```
$ cut -d: -f1 passwd.txt
root
mail
ftp
cristina
daemon
oliverio

$ cut -d: -f1 passwd.txt | paste -s
root mail ftp cristina daemon oliverio
$
```

Uso de la opción `-s` con dos archivos pasados como argumento:

```
$ paste -s francés.txt historia.txt
Cristina 12 Francisco 8 Valeria 18
Cristina 11 Francisco 10 Valeria 14
```

## 5. split

El comando **split** permite dividir un archivo en fragmentos; cada fragmento se almacena en archivos llamados PREFIJOaa, PREFIJOab... PREFIJO, que tiene como valor por defecto "x". Si ningún nombre de archivo se pasa como argumento, se usará la entrada estándar.

### Sintaxis

```
split [ opciones ] [ archivo [PREFIJO]]
```

#### Principales opciones:

- `-b num` El archivo se divide en trozos de "num" bytes.
- `-l num` El archivo se divide en trozos de "num" líneas.

### Ejemplos

A continuación se muestra un archivo que contiene 9 líneas:

```
$ cat líneas.txt
línea1
línea2
línea3
línea4
línea5
línea6
línea7
línea8
línea9
```

El archivo se divide en archivos de 3 líneas cada uno. Por defecto, los archivos de salida se llaman xaa, xab, xac:

```
$ split -l3 líneas.txt

$ ls x*
xaa xab xac
```

```
$ cat xaa
línea1
línea2
línea3
```

Cambiar el prefijo *x* de los nombres de archivo de salida por el prefijo *línea*:

```
$ split -l3 líneas.txt línea
$ ls línea??
líneaaa  líneaab  líneaac
```

## 6. Transformación de caracteres: tr

El comando **tr** (translate) permite la aplicación de un tratamiento a ciertos caracteres de un flujo de datos: eliminación, sustituciones... Este comando explota únicamente los datos que recibe por la entrada estándar.

### Sintaxis

```
tr [ opciones ] conjunto1 [conjunto2]
```

#### Principales opciones:

`[-d caracteres]` La lista de los caracteres que tienen que eliminarse del flujo de datos.

### Sustitución de caracteres

Se deben especificar dos conjuntos de caracteres. Cada conjunto contendrá el mismo número de caracteres. Todo carácter de *conjunto1* que se encuentre en el flujo de datos se sustituirá por el carácter de la misma posición de *conjunto2*.

#### Ejemplos

La cadena **abcd** se procesa: cada carácter **b** se transformará en **8** y cada carácter **d** en **9**.

```
$ echo abcd | tr bd 89
a8c9
```

Es posible usar el concepto de los intervalos de caracteres, utilizando corchetes. En este caso, cada minúscula se transformará en su mayúscula correspondiente:

```
$ echo abcd | tr "[a-z]" "[A-Z]"
ABCD
```

Para procesar un archivo, hay que usar la redirección:

```
$ tr "[a-z]" "[A-Z]" < arch.txt
CRISTINA      12
VALERIA       18
...
```

### Eliminación de caracteres

La opción `-d` permite eliminar ciertos caracteres del flujo de datos.

#### Ejemplos

Transformación de un archivo en formato DOS (final de línea "\r\n") al formato Unix ("\n"). Eliminación del carácter "\r":

```
$ tr -d '\r' < archdos.txt > archunix.txt

$ od -c archunix.txt
0000000  c  r  i  s  t  i  n  a      1  2  \n  v  a  l  e
0000020  r  i  a      \t  1  8  \n
```

## 7. Eliminación de líneas repetidas: uniq

El comando **uniq** permite eliminar las líneas repetidas de un archivo. Solamente las líneas idénticas consecutivas son procesadas. El resultado se almacena en un archivo de salida, si este se especifica, o por la salida estándar en caso contrario.

### Sintaxis

```
uniq [ opciones ] [ archivo_de_entrada [ archivo_de_salida ] ]
```

#### Principales opciones:

- d                   Mostrar las repeticiones.
- c                   Contar las repeticiones.

### Ejemplos

En el archivo **contactos.txt**, las líneas 1, 6, 7 por un lado y las líneas 2, 5, 8 por el otro son idénticas:

```
$ nl contactos.txt
  1  Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
  2  Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|
Barcelona|932.282.177
  3  Hernández Darín, Alberto|plaza mayor 15/17|13190|Corral
de Calatrava|926/448/829
  4  Expósito Heredia, Pedro|calle del castillo 3|38870|
La Calera|984.122.369
  5  Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|
Barcelona|932.282.177
  6  Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
  7  Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
  8  Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|
Barcelona|932.282.177
```

El archivo se ordena para que las líneas idénticas se coloquen consecutivamente:

```
$ sort contactos.txt
Expósito Heredia, Pedro|calle del castillo 3|38870|La Calera|
984.122.369
Hernández Darín, Alberto|plaza mayor 15/17|13190|Corral de
Calatrava|926/448/829
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|Barcelona|
932.282.177
```

```
Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|Barcelona|
932.282.177
Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|Barcelona|
932.282.177
```

*Eliminación de repetidos:*

```
$ sort contactos.txt | uniq
Expósito Heredia, Pedro|calle del castillo 3|38870|La Calera|
984.122.369
Hernández Darín, Alberto|plaza mayor 15/17|13190|Corral de
Calatrava|926/448/829
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|
Barcelona|932.282.177
```

*Mostrar delante de cada línea su número de apariciones en el archivo:*

```
$ sort contactos.txt | uniq -c
      1 Expósito Heredia, Pedro|calle del castillo 3|38870|
La Calera|984.122.369
      1 Hernández Darín, Alberto|plaza mayor 15/17|13190|Corral
de Calatrava|926/448/829
      3 Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
      3 Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|
Barcelona| 932.282.177
```

*Mostrar únicamente las líneas que tienen duplicados:*

```
$ sort contactos.txt | uniq -d
Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|
Barcelona|932.282.177
```

*Lo mismo, con el número de ocurrencias en el archivo:*

```
$ sort contactos.txt | uniq -dc
      3 Méndez Roca, Gisela|calle Ruiseñor. nº 12|28023|Madrid|
915..351.478
      3 Ruiz del Castillo, Marcos|calle Balmes nº 256|08020|
Barcelona|932.282.177
```



# Compresión, archivado y conversión

## 1. Compresión: gzip, bzip2

Los comandos **gzip/gunzip**, **bzip2/bunzip2** comprimen/descomprimen cada archivo pasado como argumento en la línea de comandos. El comando **bzip2** ofrece una mejor tasa de compresión. Estos comandos se consideran estándar en los sistemas Linux y pueden instalarse en los sistemas Unix, en caso de no estarlo. El comando **gzip** produce la extensión **.gz** y el comando **bzip2** produce la extensión **.bz2**.

### Sintaxis de gzip

*Compresión:*

```
gzip [ opciones ] [ archivo ... ]  
Descompresión:
```

*Descompresión:*

```
gunzip [ opciones ] [ archivo ... ]  
zcat [ opciones ] [ archivo ... ]
```

### **Principales opciones:**

- c    Enviar el resultado de la compresión/descompresión por la salida estándar. El comando `gunzip -c` es equivalente al comando `zcat`.
- t    Comprobar la validez del archivo.

Si el nombre del archivo se omite, los datos que se han de procesar se leen de la entrada estándar.

### Ejemplo

*Compresión del archivo **java.doc**:*

```
$ gzip java.doc
```

*El archivo original se reemplaza por el archivo comprimido:*

```
$ ls java*  
java.doc.gz
```

*Descompresión: el archivo original se restituye y reemplaza al archivo comprimido:*

```
$ gunzip java.doc  
$ ls java*  
java.doc
```

*Compresión con envío del flujo comprimido hacia la salida estándar (-c). Este modo de proceder permite conservar el archivo original:*

```
$ gzip -c java.doc > java.doc.gz  
$ ls java*  
java.doc java.doc.gz
```

*Descomprimir sin suprimir el archivo comprimido:*

```
$ gunzip -c java.doc.gz > java.doc
```

Lo mismo con el comando **zcat**:

```
$ zcat java.doc.gz > java.doc
```

Los comandos **bzip2**, **bunzip2** y **bzcat** funcionan de un modo parecido, con una ligera diferencia en el uso de las opciones.

## Sintaxis de bzip2

*Compresión:*

```
bzip2 [ opciones ] [ archivo ... ]
```

*Descompresión:*

```
bunzip2 [ archivo ... ]  
bzip2 -d [ opciones ] [ archivo ... ]  
bzcat [ opciones ] [ archivo ... ]
```

### Principales opciones:

- c Enviar el resultado de la compresión/descompresión por la salida estándar. El comando `bzip2 -dc` es equivalente al comando `bzcat`.
- d Descompresión.
- t Comprobar la validez del archivo.

## 2. Archivos tar

El comando **tar** permite crear un archivado a partir de uno o varios archivos. Si el archivo es un directorio, se archivará con toda su subestructura.

### Sintaxis

Creación de un archivado:

```
tar -c [-zv] -f archivado archivos_para_archivar ...
```

Verificación de un archivado:

```
tar -t [-zv] -f archivado [ archivos_para_verificar ... ]
```

Extracción de un archivo:

```
tar -x [-zv] -f archivado [ archivos_para_extraer ... ]
```

### Principales opciones:

- c Creación de un archivado.
- t Verificación de un archivado.
- x Extracción de un archivado.
- f *archivo.tar* Nombre del archivado que se ha de crear, verificar o extraer.
- f - En el caso de creación, el contenido del archivado se envía a la salida estándar. Si es una extracción o verificación, el contenido del archivado se muestra por la salida estándar.
- v Modo verbose (detallado).
- z En Linux únicamente: permite generar la compresión en más de un archivado.

## Ejemplos

Archivado y compresión del directorio **docs**: el flujo producido por el comando **tar** se envía por la salida estándar (y por consiguiente a la tubería) y es recuperado por el comando **gzip**, que comprime este flujo y lo dirige al archivo **docs.tar.gz**.

```
$ ls -R docs
docs:
java.doc  php.doc  shell.doc  unix.doc  xml.doc
$
$ tar cvf - docs | gzip -c > docs.tar.gz
docs/
docs/shell.doc
docs/java.doc
docs/xml.doc
docs/unix.doc
docs/php.doc
```

Descompresión y extracción: el comando **gunzip** descomprime el archivo **docs.tar.gz** y envía el resultado por la salida estándar. El comando **tar** explota el flujo que llega por la tubería y realiza la extracción.

```
$ gunzip -c docs.tar.gz | tar xvf -
docs/
docs/shell.doc
docs/java.doc
docs/xml.doc
docs/unix.doc
docs/php.doc
```

- En este caso, los nombres en el archivado se expresan con ruta relativa; por lo tanto, los archivos se crean de nuevo desde el directorio actual cuando se restauran.

## 3. Archivos cpio

El comando **cpio** también permite realizar archivados. Contrariamente a lo que sucede con el comando **tar**, no es recursivo: todos los nombres de archivo que se incluyen en la composición del archivado tienen que leerse de la entrada estándar. El archivado se envía por la salida estándar. El comando se usa casi siempre detrás de una tubería, precedida del comando **find**, que permite generar los nombres de archivo deseados.

Para verificación y restauración, el archivado que se ha de procesar se espera por la entrada estándar.

### Sintaxis

*Creación de un archivado:*

```
find . | cpio -o [ -cv ] > archivado
```

*Verificación de un archivado:*

```
cpio -it [ -cv ] < archivado [ archivos_para_verificar ]
```

*Extracción de un archivado:*

```
cpio -i [ -cdv ] < archivado [ archivos_para_extraer ]
```

## Principales opciones:

- c Creación/verificación/extracción: archivados con un formato de cabecera portable entre sistemas Unix.
- d Extracción: creación de los directorios si es que no existen.
- i Extracción de un archivado.
- Verificación de un archivado.
- it
- o Creación de un archivado.
- v Modo verbose.

## Ejemplos

Creación de un archivado **cpio** que contiene el directorio **docs** y sus subdirectorios:

```
$ find docs
docs
docs/shell.doc
docs/java.doc
docs/xml.doc
docs/unix.doc
docs/php.doc
docs/java.doc.gz

$ find docs | cpio -ocv > docs.cpio
docs
docs/shell.doc
docs/java.doc
docs/xml.doc
docs/unix.doc
docs/php.doc
docs/java.doc.gz
2 bloques
```

Verificación del archivado:

```
$ cpio -icvt < docs.cpio
drwxrwxr-x 2 cristina cristina 0 Ene 1 22:28 docs
-rw-r--r-- 1 cristina ociensa 0 Ene 1 21:40 docs/shell.doc
-rw-r--r-- 1 cristina ociensa 0 Ene 1 21:40 docs/java.doc
-rw-r--r-- 1 cristina ociensa 0 Ene 1 21:40 docs/xml.doc
-rw-r--r-- 1 cristina ociensa 0 Ene 1 21:40 docs/unix.doc
-rw-r--r-- 1 cristina ociensa 0 Ene 1 21:40 docs/php.doc
-rw-r--r-- 1 cristina ociensa 29 Ene 1 22:28 docs/
java.doc.gz
2 bloques
```

Creación y compresión del archivado:

```
$ find docs | cpio -ocv | bzip2 -c > docs.cpio.bz2
docs
docs/shell.doc
docs/java.doc
docs/xml.doc
docs/unix.doc
docs/php.doc
docs/java.doc.gz
2 bloques
```

```
$ cd /tmp
$ bzip2 -dc docs.cpio.bz2 | cpio -icvd
docs
docs/shell.doc
docs/java.doc
docs/xml.doc
docs/unix.doc
docs/php.doc
docs/java.doc.gz
2 bloques
```

- La misma observación que para el comando **tar**: los nombres en el archivado se guardan con su ruta relativa y se crean de nuevo desde el directorio actual cuando se restauren.

## 4. Copia física, transformaciones: dd

El comando **dd** realiza una copia física de un flujo de entrada hacia un flujo de salida. Cada byte deseado del flujo de entrada se copia hacia el flujo de salida, independientemente de lo que represente el dato.

### Sintaxis

```
dd [ if=archivo_de_entrada ] [ of=archivo_de_salida ] [ ibs=numbytes ]
[ obs=numbytes ] [ bs=numbytes ] [ skip=numbloques ] [ seek=numbloques ]
[ count=numbloques ]
```

#### Principales opciones:

<code>if=archivo_de_entrada</code>	Archivo que se va a tratar. Si no se especifica ninguno, se procesará la entrada estándar.
<code>of=archivo_de_salida</code>	Archivo de salida. Si no se especifica ninguno, se usará la salida estándar.
<code>ibs=numbytes</code>	Tratar los datos de entrada en bloques de <code>numbytes</code> bytes (por defecto: 512 bytes).
<code>obs=numbytes</code>	Escribir los datos de salida en bloques de <code>numbytes</code> bytes (por defecto: 512 bytes).
<code>bs=numbytes</code>	Tratar los datos de entrada y salida en bloques de <code>numbytes</code> .
<code>skip=numbloques</code>	Saltar <code>numbloques</code> en la entrada antes de empezar el proceso.
<code>seek=numbloques</code>	En la salida, escribir desde el bloque <code>numbloques</code> .
<code>count=numbloques</code>	Escribir únicamente <code>numbloques</code> de entrada hacia la salida.

### Ejemplos

Copia de un disco en otro disco en bloques de 8 kilobytes:

```
# dd if=/dev/dsk/c0t0d0s2 of=/dev/dsk/c0t1d0s2 bs=8k
```

Copia de seguridad en cinta remota (máquina saturno) en formato cpio gzipeado:

```
# cd /export/home
# find . | cpio -ocv | gzip -c | ssh saturno dd of=/dev/rmt/0 bs=1k
```

Restauración desde la máquina remota en la máquina local:

```
# cd /export/home
# ssh saturno dd if=/dev/rmt/0 bs=1k | gunzip -c | cpio -icvd
```

Eliminar los 3 primeros bytes parásitos del archivo **index.php**. Cada bloque es de 1 byte, los 3 primeros bloques no se procesan:

```
$ od -c index.php | head -1
0000000 357 273 277 < ? p h p \r \n o b _ s t a
$ dd bs=1 skip=3 if=index.php of=index2.php
$ od -c index2.php | head -1
0000000 < ? p h p \r \n o b _ s t a r t ( )
```

Eliminar el último byte de un archivo de tamaño 186 073 bytes. El tamaño del bloque se fija a 186 072 bytes. El número de bloques procesados es 1:

```
$ ls -l f1.txt
-rw-r--r-- 1 cristina clientes 186073 Ene 4 18:48 f1.txt
$ dd if=f1.txt of=f1trunc.txt bs=186072 count=1
1+0 records in
1+0 records out
$ ls -l f1trunc.txt
-rw-r--r-- 1 cristina clientes 186072 Ene 4 18:49 f1trunc.txt
$
```

## 5. Cambio de codificación: iconv

El comando **iconv** permite transformar la codificación de un flujo de datos, normalmente de la codificación "utf8" a "iso8859-15" y viceversa.

### Sintaxis

```
iconv opciones [ archivo_para_tratar ]
```

#### Principales opciones:

-f codificación\_de\_entrada Codificación del archivo de entrada.  
-t codificación\_de\_salida Codificación del archivo de salida.

### Ejemplos

Conversión del archivo **utf8.txt** (UTF-8) a **iso.txt** (ISO8859-15): el archivo contiene la serie de caracteres "cáñamo":

```
$ od -c utf8.txt
0000000 c 303 241 303 261 a m o \n
0000011
$ iconv -f UTF8 -t ISO885915 utf8.txt > iso.txt
$ od -c iso.txt
0000000 c 341 361 a m o \n
0000007
```

# Comandos de red seguros

Esta sección presenta el comando de conexión a distancia "ssh" y el comando de transferencia de archivos **sftp**.

## 1. Conexión remota: ssh

El comando **ssh** (secure shell) permite conectarse a una máquina remota. Los datos intercambiados entre las máquinas están encriptados, lo que hace a este comando más interesante que sus homólogos **telnet** y **rlogin**. También es posible la ejecución de un comando a distancia.

### Sintaxis

*Conexión a una máquina remota:*

```
ssh [-l nombre_login] nombre_máquina
```

*Ejecución de un comando a una máquina remota:*

```
ssh nombre_login@nombre_máquina [comando]
```

Con la primera conexión al servidor, este envía su clave pública. Esta clave representa el identificador único de la máquina.

El comando **ssh** solicita al usuario si reconoce el valor de la clave y si acepta la conexión. En caso afirmativo, la clave del servidor se almacena en el archivo **\$HOME/.ssh/known\_hosts** de la máquina cliente.

- Teóricamente, el usuario de la máquina cliente tiene que verificar con el administrador de la máquina remota que la clave propuesta es correcta antes de aceptar la primera conexión.

### Ejemplos

*Primera conexión al servidor www.misitio.com: el usuario de la máquina local es "root"; ssh usa este usuario en la máquina remota:*

```
# ssh www.misitio.com
The authenticity of host 'www.misitio.com (162.44.116.12)'
can't be established.
RSA key fingerprint is 1f:7b:e4:99:b1:c7:56:59:a0:a8:b1:ed:56:7f:19:a3.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'www.misitio.com,162.39.106.10' (RSA)
to the list of known hosts.
root@www.misitio.com's password: *****
#
```

*Conectarse como cristina en la máquina www.misitio.com:*

```
# ssh -l cristina www.misitio.com
cristina@ www.misitio.com 's password: *****
```

*Ejecutar el comando ls -l en la máquina remota:*

```
# ssh www.misitio.com ls -l
root@ www.misitio.com 's password: *****
```

➤ Si **ssh** se usa en un script, es posible comprobar su estado de retorno. Si **ssh** produce algún error, retorna el estado 255 (error de conexión al servidor, por ejemplo). Si devuelve otro valor, este representa el estado devuelto por el comando que se ha pasado como argumento de **ssh**. A modo de recordatorio, el estado del último comando se obtiene mediante de la variable  `$?` .

## 2. Transferencia de archivos: **sftp**

El comando **sftp** (*secure file transfer protocol*) permite transferir archivos de manera encriptada entre dos máquinas. Es una alternativa al comando **ftp**. En la primera conexión a la máquina remota, el comportamiento observado es el mismo que en el comando **ssh** (ver sección Comandos de red seguros - Conexión remota: **ssh**, en este capítulo).

### Sintaxis

```
sftp [ opciones ] [[nombre_login@]nombre_máquina[:archivo
[archivo]]]
sftp [ opciones ] [[nombre_login@]nombre_máquina[:directorio]]
```

#### Principales opciones:

- b *archivo* Modo batch. Los comandos "sftp" se leen desde un *archivo*.
- b - Modo batch. Los comandos "sftp" se leen desde la entrada estándar.

En los siguientes ejemplos, la máquina local se llama **pluton**, y la máquina remota, **venus**.

#### Ejemplos

Establecer una conexión con la máquina **venus** para transferir archivos. Por defecto, **sftp** intenta conectarse con el mismo usuario de la máquina local (*cristina*):

```
cristina@pluton $ sftp venus
Connecting to venus...
cristina@venus's password: *****
sftp>
```

Establecer una conexión a la máquina **venus** con el usuario **root**:

```
cristina@pluton $ sftp root@venus
Connecting to venus...
root@venus's password: *****
sftp>
```

Una vez se establece la conexión, el usuario se encuentra con el prompt `sftp`, desde donde podrá ejecutar comandos **sftp**.

### Ayuda acerca de los comandos de **sftp**

```
sftp> help
Available commands:
cd path          Change remote directory to 'path'
lcd path         Change local directory to 'path'
. . .
```

### Salir de **sftp**



```
sftp> quit
```

## a. Comandos de sftp que se ejecutan en la máquina local

El comando **sftp** ofrece comandos internos. La mayor parte reproducen el comportamiento de comandos **unix**. El nombre de los comandos comienza por la letra "l", lo que permite recordar que se ejecutan en la máquina local.

### Cambiar de directorio

```
sftp> lcd /tmp
```

### Saber el directorio actual

```
sftp> lpwd  
Local working directory: /tmp
```

### Listar los archivos del directorio actual

```
sftp> ll  
sftp> ll -l
```

### Crear un directorio

```
sftp> mkdir /tmp/cristina
```

### Escape al shell

```
sftp> !  
cristina@pluton:/tmp$ wc -l f1  
cristina@pluton:/tmp$ # . . . otros comandos unix ...  
cristina@pluton:/tmp$ exit #  
exit  
sftp>
```

Para volver al comando **sftp**, es suficiente con ejecutar el comando **exit** en el shell.

### Ejecutar un comando Unix en la máquina local

```
sftp> ! comando_unix
```

 Los comandos Unix no implementados por **sftp** internamente tienen que ejecutarse de este modo.

## b. Comandos que se ejecutan en la máquina remota

El comando **sftp** ofrece comandos internos que permiten realizar acciones en la máquina remota.

### Cambiar de directorio

```
sftp> cd /tmp
```

### **Listar archivos**

```
sftp> ls  
sftp> ls -l
```

### **Eliminar un archivo**

```
sftp> rm /tmp/f1.txt
```

### **Crear un directorio**

```
sftp> mkdir /tmp/cristina
```

### **Eliminar un directorio vacío**

```
sftp> rmdir /tmp/cristina
```

### **Renombrar un archivo o un directorio**

```
sftp> rename /tmp/f1.txt /tmp/f1new.txt
```

## **c. Comandos de transferencia**

Los comandos de transferencia permiten recuperar archivos de la máquina remota a la máquina local (get) o enviar archivos de la máquina local a la máquina remota (put).

### **Ejemplos**

*Cambio de directorio remota y localmente:*

```
sftp> cd /tmp  
sftp> lcd /tmp  
sftp> pwd  
Remote working directory: /tmp  
sftp> lpwd  
Local working directory: /tmp
```

*Recuperar un archivo de la máquina remota (venus) en el directorio actual (/tmp) de la máquina local (pluton):*

```
sftp> get f1.txt
```

*Recuperar los archivos de la máquina remota en un directorio (/tmp/cristina) de la máquina local:*

```
sftp> pwd  
Remote working directory: /tmp  
sftp> get * /tmp/cristina
```

*Enviar archivos de la máquina local (pluton) a la máquina remota (venus):*

```
sftp> put *.txt
```

#### d. Conexión automática sin contraseña

Es práctico poder realizar transferencias de archivos entre máquinas de manera no interactiva y, para ello, el usuario no tiene que estar obligado a introducir la contraseña por teclado. La configuración que permite acceder a una cuenta de una máquina remota sin tener que introducir la contraseña implica el uso de los comandos **ssh**, **sftp** y **scp** (copia de archivos entre dos máquinas).

- La conexión sin contraseña disminuye el nivel de seguridad; por tanto, esta configuración se usará únicamente cuando sea indispensable.

Las etapas de la configuración son las siguientes:

- Mediante el comando **ssh-keygen**, el usuario generará en la máquina local una clave pública y una clave privada, sin usar **passphrase**. Ambas claves se almacenan en local respectivamente en los archivos **\$HOME/.ssh/id\_rsa.pub** y **\$HOME/.ssh/id\_rsa** (archivos por defecto). Si se usa el protocolo SSH versión 2, es posible crear claves de tipo RSA o DSA.
- La clave pública generada en la máquina local tiene que añadirse al archivo **\$HOME/.ssh/authorized\_keys** de la máquina remota. El directorio **\$HOME** representa el directorio de inicio de la cuenta con la cual nos conectaremos. Si el archivo **.ssh/authorized\_keys** no existe, solo hay que crearlo manualmente.

#### Ejemplo

Creación de las claves pública y privada de tipo **RSA**:

```
cristina@pluton $ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/cristina/.ssh/id_rsa): ↵
Created directory '/home/cristina/.ssh'.
Enter passphrase (empty for no passphrase): ↵
Enter same passphrase again: ↵
Your identification has been saved in /home/cristina/.ssh/id_rsa.
Your public key has been saved in /home/cristina/.ssh/id_rsa.pub.
The key fingerprint is:
37:57:48:f4:2b:99:4a:33:64:35:60:31:43:27:78:47 cristina@pluton
cristina@pluton $ cd .ssh ↵ ls -l total 8
-rw----- 1 cristina ociensa 1675 ene  9 22:16 id_rsa
-rw-r--r-- 1 cristina ociensa  399 ene  9 22:16 id_rsa.pub
$ cat id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEAqZWxivOa54Zvh90rmB6RO9igxEZyrs3h
CWtEwfBJTlN1
eDdezA/c1VBcrwANBCFZ6rLXmRdXS1w367q/kGB6pvnLena6m5/mSHVZETp
5MRzE0T8atS7Oe/SqjNIsyHWMF6V+jc/0m8XZxRZzbX44wXw6eHSOY5jarnq
0esdc+afgB+EiwVEW8UjSXsDFNEuuk/MfSyY3RHjYZg5BP57NRU5bIhzeDXtxisQ
KbN50toAe3br5F0xlhC9M2njPCRbzD3ZsteHQA2gp4cEhQbW03mYMDctCQHj
HjBJi9/A70ZyHEep1YjlufwTfWb4M1wG/g+vs2vuX-wiVIg1NLAFe2YQ==
cristina@pluton
$
```

En la máquina remota, añadir el contenido del archivo **id\_rsa.pub** al archivo **/home/olivia/.ssh/authorized\_keys**:

```
olivia@venus$ cd .ssh
olivia@venus$ vi authorized_keys
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAqZWxivOa54Zvh90rmB6RO9igxEZyrs3hCWtE
WfBJTlN1eDdezA/c1VBcrwANByCFZ6rLXmRdXS1w367q/kGB6pvnLena6m5/
mSHVZEtp5MRzE0T8atS7Oe/SqjNI syHWMF6V+jxc/0m8XZxRZzbX47wXw6eHSOY5
jarnq0esdc+6rNB+EiwVEW8UjSXsDFNEuuk/MfSyY3RHjYZg5BP57NRU5bIhzeDX
txisxKbN50toAe3br5FoxlhC9M2njPCRbzDTfFSeHQA2gp4cEhQbW03mYMDctCQHj
HjBJi9/A70ZyHEep1Yj1ufwTfWb4M1wG/g+vs2vuXwiVIg1NLAFe2YQ==
crisrina@pluton
```

*El usuario **crisrina** de la máquina **pluton** puede conectarse a la máquina **venus** con la cuenta **deolivia**, sin tener que introducir una contraseña.*

```
crisrina@pluton$ ssh -l olivia venus
Last login: Wed Jan  9 22:30:33 2014 from xx.xx.xxx.xx
Linux 2.4.34.5.
$
```

Un ejemplo de uso no interactivo del comando **sftp** se muestra en el capítulo Aspectos avanzados de la programación shell - Script de archivado incremental y transferencia sftp automática.

# Otros comandos

## 1. El comando xargs

El comando **xargs**, colocado detrás de una tubería, recupera las cadenas pasadas por argumento de la entrada estándar y ejecuta **comando2** pasándole estas cadenas como argumento.

### Sintaxis

```
comando1 | xargs [ opciones ] comando2
```

#### Principales opciones:

- t Muestra el comando o los comandos realmente ejecutados (registro de la ejecución).
- n *numarg comando 2* Sólo recibirá *numarg* argumentos y se ejecutará tantas veces como sea necesario.

En los siguientes ejemplos, el comando **time** se usa para medir el tiempo de ejecución de un comando.

El resultado se interpreta de la siguiente manera:

- **user** : tiempo de ejecución de código de usuario.
- **sys** : tiempo de ejecución del código del núcleo.
- **real** : duración total de la ejecución (user + sys + inactividad).

### Ejemplos

Buscar todos los archivos **.txt** y ejecutar el comando **grep** tantas veces como archivos se encuentren:

```
$ time find / -name "*.txt" -exec grep "php" {} \; 2>&-
real    0m23.911s
user    0m0.547s
sys     0m1.199s
```

Búsqueda de todos los archivos **.txt**. Todos los nombres de archivo emitidos por el comando **find** se pasan como argumento al comando **grep**, que se ejecuta una sola vez:

```
$ time find / -name "*.txt" | -exec grep "php" {} \; 2>&-
real    0m23.911s
user    0m0.547s
sys     0m1.999s
```

Si el comando no soporta un gran número de argumentos, es posible especificar el número máximo de ellos. En este caso, el comando **grep** es invocado tantas veces como sea necesario, con 6 argumentos en cada llamada.

```
$ time find / -name "*.txt" | xargs -n6 grep "php" 2>&-
real    0m1.195s
user    0m0.346s
sys     0m0.528s
```

Registrar la ejecución del comando con la opción **-t**:

```

$ find . -name "*.txt" | xargs -t -n4 wc -l
wc -l ./historia.txt ./doble.txt ./who.txt ./contactos.txt
  3 ./historia.txt
  3 ./doble.txt
  7 ./who.txt
  8 ./contactos.txt
 21 total
wc -l ./notas_un_espacio.txt ./filtros/f.txt ./arch.txt
./notas.txt
  5 ./notas_un_espacio.txt
  1 ./filtros/f.txt
  8 ./arch.txt
  5 ./notas.txt
 19 total
wc -l ./passwd.txt ./líneas.txt ./scripts/notas.txt ./francés.txt
  6 ./passwd.txt
  9 ./líneas.txt
 22 ./scripts/notas.txt
  3 ./francés.txt
 40 total
$

```

## 2. Comparar dos archivos: cmp

El comando **cmp** compara dos archivos de texto o binarios y devuelve un estado verdadero si son iguales o falso en caso contrario. Si los archivos son diferentes, se muestra la posición de la primera diferencia.

### Sintaxis

```
cmp [ opciones ] archivo1 archivo2
```

#### Principales opciones:

-s Modo silencioso.

Si archivo1 o archivo2 se remplazan por "-", se procesará la entrada estándar.

#### Primer ejemplo

Comparación de dos archivos diferentes:

```

$ cmp líneas1.txt líneas2.txt
líneas1.txt líneas2.txt son distintos: byte 8, línea 2
$ echo $?
1

```

Comparación de dos archivos iguales:

```

$ cmp líneas2.txt líneas3.txt
$ echo $?
0

```

#### Segundo ejemplo

El script **igual.sh** recibe dos archivos como argumento y los compara. La opción **-s** de **cmp** hace trabajar al comando en modo silencioso:

```

$ nl igual.sh
 1 if cmp -s "$1" "$2"; then
 2   echo "$1 y $2 son iguales"

```

```
3 else
4 echo "$1 y $2 son distintos"
5 fi
```

Llamada con dos archivos diferentes:


```
$ igual.sh líneas1.txt líneas2.txt
líneas1.txt y líneas2.txt son distintos
```

Llamada con dos archivos iguales:

```
$ igual.sh líneas2.txt líneas3.txt
líneas2.txt y líneas3.txt son idénticos
```

### 3. Líneas comunes entre dos archivos: comm

El comando **comm** da a conocer las líneas comunes entre dos archivos.

 Los archivos tienen que estar ordenados previamente.

#### Sintaxis

```
comm [ opción ] archivo1 archivo2
```

#### Principales opciones:

- 1 No mostrar las líneas específicas del `archivo1`.
- 2 No mostrar las líneas específicas del `archivo2`.
- 3 No mostrar las líneas comunes entre los 2 archivos.

Si `archivo1` o `archivo2` se reemplazan por "-", se procesa la entrada estándar

#### Ejemplo

El comando **comm** muestra tres columnas: la primera columna representa las líneas específicas de **f1**, la segunda columna las líneas específicas de **f2** y la tercera columna las líneas comunes entre **f1** y **f2**.

```
$ nl f1
1 aaaa
2 bbbb
3 cccc
4 dddd

$ nl f2
1 bbbb
2 cccc
3 dddd
4 eeee

$ comm f1 f2
aaaa
      bbbb
      cccc
      dddd

      eeee
```

Lo mismo sin mostrar las líneas específicas de **f1**:

```
$ comm -1 f1 f2
    bbbb
    cccc
    dddd
eeee
```

No mostrar las líneas específicas de **f2**:

```
$ comm -2 f1 f2
aaaa
    bbbb
    cccc
    dddd
```

No mostrar las líneas comunes:

```
$ comm -3 f1 f2
aaaa

    eeee
```



# Soluciones del capítulo Mecanismos esenciales del shell

## 1. Funcionalidades varias

### a. Ejercicio 1: comandos internos y externos

¿Son los comandos **umask** y **chmod** comandos internos?

```
$ type umask
umask is a shell builtin
$ type chmod
chmod is a tracked alias for /usr/bin/chmod
```

**umask** es un comando interno. **chmod** es un comando externo. El comando interno **type** permite saber si un comando se encuentra implementado de manera interna.

### b. Ejercicio 2: generación de nombres de archivo

Sea la siguiente lista de archivos:

```
$ ls
bd.class.php  header.inc.php  install.txt  readme.txt
prueba       index.php      mail.class.php
```

1. Muestre los nombres de archivo que terminan en **.php**.

```
$ ls *.php
bd.class.php  header.inc.php  index.php  mail.class.php
```

2. Muestre los nombres de archivo que tengan la letra **e** en segunda posición.

```
$ ls ?e*
header.inc.php  readme.txt
```

3. Muestre los nombres de archivo cuya primera letra esté comprendida entre **a** y **e**.

```
$ ls [a-e]*
bd.class.php  prueba
```

4. Muestre los nombres de archivo que no comienzan por una vocal.

```
$ ls [!aeiouy]*
bd.class.php  header.inc.php  mail.class.php  readme.txt
```

### Expresiones complejas (ksh, bash)

En bash, debemos activar el reconocimiento de expresiones complejas:

```
$ shopt -s extglob
```

5. Muestre los nombres de archivo que no terminan en **.php**.

```
$ ls !(*.php) prueba      install.txt  readme.txt
```

6. Muestre los nombres de archivo que no terminan ni con **.txt** ni con **.php**.

```
$ ls !(*.php|*.txt)
prueba
```

### c. Ejercicio 3: separador de comandos

¿Cómo se escriben los dos comandos siguientes en la misma línea?

```
$ cd /tmp
$ ls -l
```

Debemos emplear el carácter especial del shell; que sirve como separador de los dos comandos:

```
$ cd /tmp ; ls -l
```

## 2. Redirecciones

### a. Ejercicio 1

Liste todos los procesos del sistema y redirija el resultado a un archivo.

```
$ ps -ef > resu
```

o

```
$ ps -ef 1> resu
```

### b. Ejercicio 2

Sea el comando **who -A**, que genera un mensaje de error:

```
$ who -A
who : opción inválida -- 'A'
```

1. Relance este comando y redirija los errores a un archivo.

```
$ who -A 2> error
```

2. Relance este comando y haga desaparecer los errores sin redirigir a un archivo en disco.

```
$ who -A 2> /dev/null
```

### c. Ejercicio 3

Ejecute los comandos siguientes:

```
$ touch fic_existe $ chmod 600 fic_existe fic_noexiste
chmod: no se puede acceder a "fic_noexiste": No existe el archivo
o el directorio
```

1. Redirija el resultado del comando **chmod** a un archivo, los errores a otro.

```
$ chmod 600 fic_existe fic_noexiste 1> resu 2> error
```

2. Redirija los resultados y los errores del comando a un mismo archivo.

```
$ chmod 600 fic_existe fic_noexiste 1> resu 2>&1
```

#### d. Ejercicio 4

¿Qué hace el comando siguiente?

```
$ ls > resu -l
```

Este comando hace lo mismo que `ls -l > resu`. Sea cual sea la ubicación de las redirecciones en la línea de comandos, estas son tratadas antes de la ejecución del comando.

#### e. Ejercicio 5

Ejecute los comandos **date**, **who** y **ls** y guarde el resultado de los tres comandos en un archivo (una sola línea de comando).

```
$ ( date ; who ; ls ) > archivo
```

#### f. Ejercicio 6

Ejecute los comandos **date** y **who -A** y almacene el resultado de los dos comandos en un archivo **resu** (una sola línea de comando). Recuerde: el comando **who -A** genera un mensaje de error.

```
$ (date ; who -A) > resu 2>&1
```

### 3. Tuberías de comunicación

#### a. Ejercicio 1

Muestre la lista de procesos paginando el resultado.

```
$ ps -ef | more
```

También podemos paginar con los comandos **pg** o **less**, según su disponibilidad en el sistema.

#### b. Ejercicio 2

Combinando los comandos **ps** y **grep**, muestre la lista de los procesos **httpd** que funcionan en el sistema.

```
$ ps -ef | grep httpd
```

### c. Ejercicio 3

Combinando los comandos **tail** y **head**, muestre la sexta línea del archivo **/etc/passwd**.

```
$ head -6 /etc/passwd | tail -1  
sync:x:5:0:sync:/sbin:/bin/sync
```

### d. Ejercicio 4

Cree los archivos siguientes:

```
$ touch f2 f1 ficl.txt FIC.c Fic.doc fIc.PDF fic
```

Cuente el número de archivos cuyo nombre contenga la palabra **fic**. La búsqueda no deberá discriminar entre mayúsculas y minúsculas.

```
$ ls | grep -i fic | wc -l
```

# Soluciones del capítulo Configuración del entorno de trabajo

## 1. Variables de entorno

### a. Ejercicio 1

1. Haga que se muestre la lista de todas las variables de entorno.

```
$ set
```

2. Haga que se muestre la lista de las variables de entorno exportadas.

```
$ env
```

3. Haga que se muestre el valor de las variables de entorno **PATH** y **HOME**.

```
$ echo $PATH $ echo $HOME
```

### b. Ejercicio 2

Cree un directorio **bin** en el directorio de inicio. En **bin**, cree o recupere el programa corto siguiente (script shell que muestra una frase en la pantalla):

```
$ pwd
/home/christie/bin
$ vi micomando
echo "Ejecución de micomando "
$ chmod u+x micomando
```

1. Vuelva al directorio de inicio.

```
$ cd ; pwd /home/cristina
```

2. Modifique la variable **PATH** de forma que este comando funcione:

```
$ micomando
```

Para que el comando pueda ser encontrado, debemos agregar su ruta a la variable **PATH**.

```
$ echo $PATH
/bin:/usr/bin
$ PATH=$PATH:$HOME/bin $ echo $PATH /bin:/usr/bin:/home/cristina/bin
$ micomando
Ejecución de micomando
```

3. Haga que esta configuración sea permanente.

Debemos añadir este parámetro en el archivo `$HOME/.profile` (sh, ksh) o `.bash_profile` (bash).

```
$ vi .bash_profile           (.profile en sh,ksh)
PATH=$PATH:$HOME/bin
export PATH
```

Hacemos que el shell actual vuelva a leer el archivo:

```
$ . $HOME/.bash_profile     (.profile en sh,ksh)
```

## 2. Alias de comando

### a. Ejercicio 1

1. Haga que se muestren los alias del shell actual (ksh, bash).

```
$ alias
```

2. Cree un alias **p** que corresponda al comando **ps -ef | more**.

```
$ alias p='ps -ef | more'
```

3. Pruebe el alias.

```
$ p
```

4. Destruya el alias.

```
$ unalias p
```

5. Haga que este alias sea permanente.

Debemos definir un alias en el archivo **.bashrc** en bash o **.kshrc** en ksh.

```
$ vi .bashrc                 (.kshrc en ksh)
alias p='ps -ef | more'
$ . $HOME/.bashrc           (.kshrc en ksh)
```

### b. Ejercicio 2

Mientras se encontraba definiendo un alias, el usuario desafortunadamente ha tocado la tecla [Entrar] antes de haber podido cerrar las comillas. ¿A qué corresponde el carácter **>** y cómo salir de esta situación?

```
$ alias 'l=ls -l
Entrar
>
```

Se trata del prompt **PS2** del shell que se muestra cuando el intérprete detecta un error de sintaxis del shell. Este permite seguir con la escritura del comando.

```
$ alias 'l=ls -l Entrar
> '
Entrar
$
```

---

Para abandonar el comando, bastará con pulsar la tecla [Ctrl] **C**.

# Soluciones del capítulo Las bases de la programación shell

## 1. Variables, caracteres especiales

### a. Ejercicio 1: variables

1. Defina una variable que contenga su nombre. Muestre esta variable

```
$ nombre=Cristina
$ echo $nombre
```

2. Defina una variable que contenga su nombre seguida de su apellido. Muestre esta variable.

```
$ nombreapellido='Cristina Deffaix'
$ echo $nombreapellido
```

3. Elimine las dos variables (dejándolas indefinidas).

```
$ unset nombre nombreapellido
```

### b. Ejercicio 2: variables

Defina una variable que contenga su apellido, y otra que contenga su nombre. Utilizando un solo echo, muestre las dos variables, separadas por un carácter de subrayado (apellido\_nombre).

```
$ apellido='Deffaix'
$ nombre=Cristina
```

Incorrecto: el carácter subrayado es interpretado como parte del nombre de la variable.

```
$ echo $nombre_ $apellido
Deffaix
```

Correcto: debemos emplear los caracteres que aíslan el nombre de una de las variables.

```
$ echo ${nombre}_ $apellido
Cristina_Deffaix
```

### c. Ejercicio 3: sustitución de comando

1. En un solo comando, muestre la fecha actual:

```
Hoy es mié 4 feb 14:32:22 CET 2015
```

```
$ echo Hoy es $(date)
```

o

```
$ echo Hoy es `date`
```



2. Igual pero aplique a la fecha el formato siguiente:

Hoy es el 04/02/2015

```
$ echo Hoy es $(date +%d/%m/%Y)
```

#### d. Ejercicio 4: caracteres de protección

El directorio actual contiene los archivos **f1**, **f2** y **f3**:

```
$ ls
f1 f2 f3
```

¿Qué obtendrá con los comandos siguientes?:

1.  

```
$ echo *
f1 f2 f3
```

El shell sustituye **\*** por la lista de los archivos del directorio actual. El comando **echo** recibe la lista de archivos como parámetros.

2.  

```
$ echo \*
*
```

El carácter **\** evita que el shell interprete el carácter **\***.

3.  

```
$ echo "*"
*
```

Las comillas dobles evitan que el shell interprete el carácter **\***.

4.  

```
$ echo '*'
*
```

Las comillas evitan que el shell interprete el carácter **\***.

5.  

```
$ age=20
$ echo $edad
20
```

El shell sustituye la variable **\$edad** por su valor. El comando **echo** recibe el resultado de la sustitución.

6.  

```
$ echo \$edad
$edad
```

El carácter **\** evita que el shell interprete el carácter **\$**.

7.  

```
$ echo "$edad"
20
```

Ubicado entre comillas dobles, el carácter **\$** conserva su significado especial. La variable es sustituida.

8.  

```
$ echo '$edad'
$edad
```

Ubicado entre comillas, el carácter **\$** pierde su significado especial.

9.

```
$ echo "Tú eres $(logname) y tienes -> $edad años"  
Tú eres cristina y tienes -> 20 años
```

Al estar ubicado entre comillas dobles, las sustituciones de comando y las variables son realizadas. En contraposición, el carácter **>** pierde su significado de redirección.

10.

```
$ echo Tú eres $(logname) y tienes -> $edad años
```

Este comando no muestra nada, porque la salida estándar se encuentra redirigida a un archivo llamado **20**. Después de efectuadas las sustituciones, el comando es este:

```
$ echo Tú eres cristina y tienes-> 20 años
```

Lo que equivale a:

```
$ echo Tú eres cristina y tienes - años > 20
```

## 2. Variables, visualización y lectura del teclado

### a. Ejercicio 1: variables

Escriba un script **primer.sh** y realice las operaciones siguientes:

- Inicialice una variable **nombre**.
- Inicialice una variable **miFecha** que contendrá la fecha actual.
- Muestre las dos variables.

Ejecute este script.

```
$ nl primer.sh  
1  #!/bin/bash          # Adaptar según shell  
  
2  nombre=Cristina  
3  miFecha=$(date)  
4  echo "Mi nombre es $nombre ; Hoy es $miFecha"
```

Cambie los permisos para que el script sea ejecutable:

```
$ chmod u+x primer.sh
```

La carpeta del script debe encontrarse en la variable **PATH** para poder llamar al script según sigue:

```
$ primer.sh  
Mi nombre es Cristina ; Hoy es vie feb 6 15:57:49 CET 2015
```

### b. Ejercicio 2: parámetros posicionales

Escriba un shell script **wcount.sh** que produzca el resultado siguiente:

```
$ bash wcount.sh oso pájaro  
El nombre del script es: wcount.sh  
El primer argumento es: oso
```

```
El segundo argumento es: pájaro
Todos los argumentos: oso pájaro
Número total de argumentos: 2
El primer argumento contiene: 3 caracteres
El segundo argumento contiene: 6 caracteres
```

Script:

```
$ nl wcount.sh
1  #! /bin/bash # adaptar según shell

2  echo El nombre del script es: $0
3  echo El primer argumento es: $1
4  echo El segundo argumento es: $2
5  echo Todos los argumentos: $*
6  echo Número total de argumentos: $#

7  # La opción -n permite eliminar el newline añadido por echo
8  echo "El 1er argumento contiene: $(echo -n "$1" | wc -c) caracteres"
9  echo "El 2do argumento contiene: $(echo -n "$2" | wc -c) caracteres"

10 # En ksh:
11 # echo "El 2do argumento contiene: $(echo "$2\c" | wc -c)
    caracteres"
```

### c. Ejercicio 3: lectura de teclado

Escriba un script **hello.sh** que:

1. Solicite al usuario la introducción de su nombre y lo almacene en una variable.
2. Solicite al usuario la introducción de su apellido y lo almacene en otra variable.
3. Muestre un mensaje de bienvenida al usuario.
4. Muestre el PID del shell que ejecuta el script.

Ejemplo

```
$ hello.sh
Introduzca su nombre: Cristina
Introduzca sus apellidos: Perez Lopez
Buenos días Cristina Perez Lopez
El PID del shell es 2569
```

Script:

```
$ nl hello.sh
1  #! /bin/bash      #Adaptar según shell

2  echo -n "Introduzca su nombre: "
3  # ksh: echo "Introduzca su nombre: \c"
4  read nombre

5  echo -n "Introduzca sus apellidos: "
6  read apellido

7  echo Buenos días $nombre $apellido
8  echo El PID del shell es $$
```

## 3. Tests y aritmética

## a. Ejercicio 1: tests a los archivos

1. Verifique si el archivo (o directorio) **/etc** existe.

Solución Bourne/ksh/bash:

```
$ [ -d /etc ]          # verdadero si /etc es un directorio
$ echo $?
0                      # verdadero
```

Solución ksh/bash:

```
$ [ -e /etc ]          # verdadero si el archivo existe (la opción -e
no existe en Bourne)
$ [[ -e /etc ]]        # verdadero si el archivo existe
$ [[ -d /etc ]]        # verdadero si el archivo es un directorio
$ echo $?
0                      # verdadero
```

2. Verifique si es posible acceder al archivo **/etc/hosts** en lectura.

```
$ [ -r /etc/hosts ]   # Bourne/bash/ksh
$ [[ -r /etc/hosts ]] # bash/ksh
$ echo $?
0                      # verdadero
```

3. Verifique si el archivo **/etc/hosts** es ejecutable.

```
$ [ -x /etc/hosts ]   # Bourne/bash/ksh
$ [[ -x /etc/hosts ]] # bash/ksh
$ echo $?
1                      # falso
```

4. Verifique si el archivo **/usr** es un directorio y si se puede atravesar.

```
$ [ -d /usr -a -x /usr ] # Bourne/bash/ksh
$ [[ -d /usr && -x /usr ]] # bash/ksh
$ echo $?
0                      # verdadero
```

5. Verifique si el archivo **/dev/null** es un archivo especial de dispositivo.

```
$ [ -c /dev/null -o -b /dev/null ] # Bourne/bash/ksh
$ [[ -c /dev/null || -b /dev/null ]] # bash/ksh
$ echo $?
0                      # verdadero
```

## b. Ejercicio 2: tests de cadenas de caracteres

Definir las siguientes variables:

```
$ s1=si
$ s2=no
$ vacia=""
$ arch1=informe.pdf
```

1. Pruebe si **\$s1** es igual a **\$s2**.

```
$ [ $s1 = $s2 ]          # Bourne/bash/ksh
$ [[ $s1 = $s2 ]]       # bash/ksh
$ echo $?
1                        # falso
```

2. Pruebe si **\$s1** es diferente de **\$s2**.

```
$ [ $s1 != $s2 ]        # Bourne/bash/ksh
$ [[ $s1 != $s2 ]]     # bash/ksh
$ echo $?
0                        # verdadero
```

3. Pruebe si **\$vacia** está vacía.

```
$ [ -z $vacia ]         # Bourne/bash/ksh
$ [[ -z $vacia ]]      # bash/ksh
$ echo $?
0
```

4. Pruebe si **\$vacia** no está vacía.

```
$ [ -n $vacia ]         # Bourne/bash/ksh
$ [[ -n $vacia ]]      # bash/ksh
$ echo $? 1
```

5. Pruebe si **\$arch1** termina en **.doc** (bash/ksh solamente).  
En bash, debemos activar la opción **shopt -s extglob**.

```
$ echo $arch1
informe.pdf
$ [[ $arch1 = *.doc ]]
$ echo $?
1                        # falso
```

6. Pruebe si **\$arch2** termina en **.doc** o en **.pdf**.  
En bash, debemos activar la opción **shopt -s extglob**.

```
$ [[ $arch1 = *.*(doc|pdf) ]]
$ echo $?
0                        # verdadero
```

### c. Ejercicio 3: tests numéricos

Definir las variables **num1** y **num2** con los valores siguientes:

```
$ num1=2
$ num2=100
```

Verifique si **\$num1** es mayor que **\$num2** empleando los comandos **[ ]**, **[[ ]]** y **(( ))**.

```
$ [ $num1 -gt $num2 ]
```

```
$ echo $? 1
$ [[ $num1 -gt $num2 ]]
$ echo $?
1
$ (( $num1 > $num2 ))
$ echo $?
1
```

¡Observe que no debe utilizar el operador de test en cadenas de caracteres!

```
$ [[ $num1 > $num2 ]]
$ echo $?
0
```

#### d. Ejercicio 4: aritmética

En la línea de comandos, inicialice dos variables numéricas **num1** y **num2**:

```
$ num1=3
$ num2=5
```

1. Inicialice una variable **res** con la suma de **num1** y **num2**. Muestre **res**. Proporcione una solución compatible con Bourne y una solución específica bash/ksh.

```
# Bourne/bash/ksh
$ res=`expr $num1 + $num2`
$ echo $res
8

# bash/ksh
$ (( res = $num1 + $num2 ))
$ echo $res
8

# Funciona de la misma forma:
$ (( res = num1 + num2 ))
$ echo $res
8
```

2. Sin inicializar la variable **res**, muestre por pantalla la suma de dos números. Proporcione a su vez las dos soluciones.

```
# Sin inicializar variable (Bourne)
$ expr $num1 + $num2
8

# Sin inicializar variable (bash/ksh)
$ echo=$(( $num1 + $num2 ))
8
```

3. Inicialice una variable **res** con el resultado de la multiplicación de **num1** y **num2**. Muestre **res**. Proporcione una solución compatible con Bourne y una solución específica bash/ksh.

```
# Multiplicación: sintaxis falsa (el Shell reemplaza el * por
# la lista de archivos del directorio actual)
$ res=`expr $num1 * $num2`
expr: error de sintaxis
```

```
# Syntaxis correcta (Bourne/ksh/bash)
$ res=`expr $num1 \* $num2`
$ echo $res
15
# En bash/ksh, el * se protege dentro del comando (( ))
$ (( res = num1 * num2 ))
$ echo $res
15
```

## e. Ejercicio 5: operadores lógicos de los comandos [ ], [[ ]] y operadores lógicos del shell

Ejecute los comandos siguientes:

```
$ > arch
$ chmod 444 arch
$ ls -l arch
-r--r--r-- 1 cristina perez 0 2 feb 17:23 arch
```

1. Si el archivo **\$arch** no es ejecutable, muestre "Permiso x no indicado".

```
$ [ ! -x $arch ] && echo Permiso x no indicado
$ [[ ! -x $arch ]] && echo Permiso x no indicado # excepto Bourne
Permiso x no indicado
```

0

```
$ [ -x $arch ] || echo Permiso x no indicado
$ [[ -x $arch ]] || echo Permiso x no indicado # excepto Bourne
Permiso x no indicado
```

2. Si el archivo **\$arch** no es ni ejecutable ni accesible para escritura, muestre "Permisos wx no indicados".

```
$ [ ! -w $arch -a ! -x $arch ] && echo Permisos wx no indicados

# excepto Bourne :
$ [[ ! -w $arch && ! -x $arch ]] && echo Permisos wx no indicados
Permisos wx no indicados
```

## 4. Estructuras de control if, case, bucle for

### a. Ejercicio 1: los comandos [ ] y [[ ]], la estructura de control if

Escriba un script **compare.sh**:

- Número de argumentos recibidos: dos nombres de archivo ordinarios.
- Verificar que el número de argumentos es igual a 2 y que los archivos son de tipo ordinario.
- Si los argumentos son correctos, el script deberá decir si los dos archivos son del mismo tamaño; en caso contrario, deberá decir cuál es el mayor de los dos.

Script compatible Bourne, ksh y bash:

```
$ nl compare1.sh
1 if [ $# -ne 2 -o ! -f "$1" -o ! -f "$2" ] ; then
2     echo "Uso: $0 archivo1 archivo2"
```

```

3   exit 1
4   fi

5   tam1=`ls -l $1 | awk '{print $5}'`
6   tam2=`ls -l $2 | awk '{print $5}'`

7   if [ $tam1 -eq $tam2 ] ; then
8       echo "Los archivos $1 y $2 son del mismo tamaño: $tam1 bytes"
9   elif [ $tam1 -gt $tam2 ] ; then
10      echo "El archivo $1 es el de mayor tamaño: $tam1 bytes"
11      else
12          echo "El archivo $2 es el de mayor tamaño: $tam2 bytes"
13      fi
14      echo "Bye ..."

```

Script compatible ksh y bash:

```

$ nl compare2.sh
1  #!/bin/bash      # Adaptar según shell
2  if [[ $# -ne 2 || ! -f "$1" || ! -f "$2"  ]] ; then
3      echo "Uso : $0 archivo1 archivo2"
4      exit 1
5  fi

6  tam1=(ls -l $1 | awk '{print $5}')
7  tam2=(ls -l $2 | awk '{print $5}')

8  if [ $tam1 -eq $tam2 ] ; then
9      echo "Los archivos $1 y $2 son del mismo tamaño: $tam1 bytes"
10 elif [ $tam1 -gt $tam2 ] ; then
11     echo "El archivo $1 es el de mayor tamaño: $tam1 bytes"
12 else
13     echo "El archivo $2 es el de mayor tamaño: $tam2 bytes"
14 fi
15 echo "Bye ..."

```

**b. Ejercicio 2: estructuras de control case, bucle for**

Escriba un script **tipoarch.sh** que tome los nombres de archivo por argumento. Si el archivo termina en **.doc** o **.pdf**, muestre un mensaje específico. En caso contrario muestre "Ni DOC, ni PDF".

Ejemplo

```

$ tipoarch.sh f1.doc f2.pdf f3.txt
f1.doc: Archivo DOC
f2.pdf: Archivo PDF
f3.txt: Ni DOC, ni PDF

```

Script compatible Bourne, ksh y bash:

```

$ nl tipoarch.sh
1  #!/bin/bash      # Adaptar según shell

2  for fic in $*
3  do
4      case $fic in
5          *.doc|*.docx)
6              echo "$fic : Archivo DOC"
7              ;;
8          *.pdf)
9              echo "$fic : Archivo PDF"
10             ;;

```



```
11      *)
12      echo "$fic : Ni DOC, ni PDF"
13      ;;
14  esac
15  done
```

## 5. Bucles

### a. Ejercicio 1: bucle for, comando tr

Escriba un script **may\_min.sh**:

- Argumento opcional: un nombre de directorio. El valor por defecto será el directorio actual.
- Verificar que el posible argumento es un directorio.
- El script renombrará los archivos del directorio: los nombres de archivo en mayúsculas serán convertidos a minúsculas.

Script compatible ksh, bash:

```
$ nl may_min.sh
 1  #!/bin/bash          # Adaptar según shell

 2  dir=${1:-.}
 3  if [[ ! -d $dir ]] ; then
 4      echo "Error $dir no es un directorio"
 5      exit 1
 6  fi

 7  echo "Procesando $dir"
 8  cd $dir

 9  for nomfic in $(ls)
10  do
11      # En ksh: echo "$nomfic\c"
12      nuevoNom=$( echo -n $nomfic | tr '[A-Z]' '[a-z]')
13      echo $nuevoNom
14      mv $nomfic $nuevoNom
15  done
```

### b. Ejercicio 2: bucle for, aritmética

Escribir un script **proc\_users.sh**:

- Argumentos: uno o más nombres de usuario.
- Verificar el número de argumentos recibido: debe haber al menos un argumento.
- Para cada usuario recibido como argumento, muestre en la pantalla el número de procesos que le pertenezcan. Si el usuario no está definido en el sistema, este será ignorado.

Script compatible bash y ksh:

```
$ nl proc_users.sh
 1  #!/bin/bash          # Adaptar según shell
 2  if (( $# < 1 )) ; then
 3      echo "Uso: $0 user1 user2 ... usern"
 4  fi

 5  # Bucle en cada usuario
 6  for user in $*
 7  do
 8      # verificar que el usuario está definido en el sistema
```

```

9  grep "^$user:" /etc/passwd > /dev/null
10 if (( $? != 0 )) ; then
11     echo "$user no es un usuario válido"
12     continue
13 fi

14 # Número de líneas devueltas por el comando ps
15 nbLineas=$( ps -u $user | wc -l )
16 # No contar la línea de encabezado
17 nbProc=$(( $nbLineas - 1 ))

18 if (( $nbProc == 0 )) ; then
19     echo "El usuario $user no tiene procesos activos"
20 else
21     echo "El usuario $user tiene $nbProc procesos en ejecución"
22 fi
23 done

```

### c. Ejercicio 3: bucles for, while

Escriba un script **consulte.sh**:

- Argumento opcional: un nombre de directorio. Verifique que el argumento recibido es un directorio.
- Si el script no recibe un nombre de directorio, trate el directorio actual.
- Busque todos los archivos ordinarios que se encuentren bajo este directorio (incluyendo subniveles).
- Para cada archivo de contenido texto accesible en lectura, pregunte al usuario si desea consultar el archivo. Los archivos que no sean de texto se ignorarán.
- El usuario podrá introducir 's', 'S', 'si', 'SI' para consultar el archivo, 'n', 'N', 'no', 'NO' para no consultarlo, o 'q' para salir del script. Cualquier otra respuesta generará una nueva pregunta al usuario.
  - Si el usuario desea consultar el archivo, muestre el contenido paginado de este en la pantalla.
  - Si el usuario no desea consultar el archivo, pase al archivo siguiente.

Script compatible bash et ksh:

```

$ nl consulte.sh
1  #!/bin/bash # Adaptar según shell
2  # Si se recibe un argumento, verificar que se trata
de un directorio
3  if [[ $# -eq 1 && ! -d $1 ]] ; then
4      echo "Uso $0 [ directorio ]"
5      exit 1
6  fi

7  # Si no se recibe un argumento, procesar el directorio actual
8  rep=${1:-.}

9  listeFic=$(find $rep -type f 2> /dev/null )
10 for fic in $listeFic
11 do
12     if file $fic | grep text 1> /dev/null 2>&1 && [[ -r $fic ]]
13     then
14         while true
15         do
16             # En ksh retirar opcion -e
17             echo -e "\nConsultar el archivo $fic ? s/n - q para salir \c"
18             read respuesta

```

```
19     case $respuesta in
20         s|s|SI|si)
21             more $fic
22             break # salir del while y pasar al archivo
                siguiente
23             ;;
24         n|n|no|NO)
25             break # salir del while y pasar al archivo
                siguiente
26             ;;
27         q|quit)
28             exit 0
29             ;;
30         *)
31             echo "Respuesta no válida"
32             ;;
33     esac
34 done
35 fi
36 done
```

\$

# Soluciones del capítulo Aspectos avanzados de la programación shell

## 1. Funciones

### a. Ejercicio 1: funciones simples

Escriba un script **audit.sh**:

- Escriba una función **users\_connect** que mostrará la lista de los usuarios conectados actualmente.
- Escriba una función **disk\_space** que mostrará el espacio en disco disponible.
- El programa principal mostrará el siguiente menú:

```
- 0 - Fin
- 1 - Mostrar la lista de usuarios conectados
- 2 - Mostrar el espacio en disco
Su opción:
```

Introducir la opción del usuario y llamar a la función adecuada. •

Script compatible bash y ksh:

```
$ nl audit.sh
 1  #!/bin/bash  # Adaptar según shell
 2
 3  function pause
 4  {
 5      echo "Pulse Entrar para continuar "
 6      read x
 7  }
 8
 9  function users_connect
10  {
11      who
12  }
13
14  function disk_space
15  {
16      df -k
17  }
18
19  while true
20  do
21      clear
22      echo "- 0 - Fin"
23      echo "- 1 - Mostrar la lista de usuarios conectados"
24      echo "- 2 - Mostrar el espacio en disco"
25      echo "Su opción: \c"
26      read opcion
27
28      case $opcion in
29
30          0)      exit 0
31
32
33          1)      users_connect
34
35
36          *)
37
38      esac
```

```

29  2)
30      disk_space
31      ;;

32  *)   echo "Opción incorrecta"
33      ;;
34  esac
35  pause
36  done

```

## b. Ejercicio 2: funciones simples, valor de retorno

Escriba un script **explore\_sa.sh**:

- Programa principal:
  - El programa principal mostrará el menú siguiente:

```

0 - Fin
1 - Eliminar los archivos de tamaño 0 de mi directorio principal
2 - Controlar el espacio de disco del SA raíz
Su opción:

```

- Introduzca la opción del usuario.
- La opción 0 provocará la finalización del script.
- La opción 1 llamará a la opción **limpieza**.
- La opción 2 causará la llamada a la función sin **espacio\_d**.
- En función del valor retornado por la función, mostrar el mensaje adecuado.
- Escriba la función **limpieza**: busque, a partir del directorio de inicio del usuario, todos los archivos que tengan tamaño 0 con objeto de eliminarlos (después de solicitar confirmación para cada archivo).
- Escriba la función **sin\_espacio\_d**: esta función verifica la utilización del sistema de archivos raíz y retorna verdadero si la tasa es superior al 80% y falso en caso contrario.

Script compatible bash y ksh:

```

$ nl explore_sa.sh
 1  #!/bin/bash # Adaptar según shell
 2  # Escribir una función que retorne verdadero si la tasa de
 3  # ocupación del Sistema de Archivos supera el 80%, falso en caso
   # contrario.

 4  function limpieza {
 5      find $HOME -size 0 -exec rm -i {} \;
 6  }

 7  function sin_espacio_d {
 8      typeset percent=`df -k / | grep "/"$ | awk '{print $5}' | tr -d %`
 9      if (( $percent > 80 )) ; then
10          return 0
11      fi
12      return 1
13  }

14  while true
15  do
16      echo "0 - Fin"
17      echo "1 - Eliminar los archivos de tamaño 0 de mi directorio
   principal "

```

```

18 echo "2 - Controlar el espacio de disco del SA raíz "
19 # No existe opción -e en ksh
20 echo -e "Su opción: \c"
21 read opcion

22 case $opcion in
23     0)
24         exit 0
25         ;;
26     1) limpieza ;;
27     2) if sin_espacio_d ; then
28         echo "Tasa de ocupación de SA raíz: ALERTA: $percent%"
29     else
30         echo "Tasa de ocupación de SA raíz: NORMAL"
31     fi
32     ;;
33 esac
34 done

```

### c. Ejercicio 3: paso de parámetros, retorno de valor

Escriba un script **calcul.sh**, que contendrá:

- Una función **esNum** que recibe un valor como argumento y que retorna verdadero si el valor es un número entero y falso en el caso contrario. Si define variables, estas deberán ser locales.
- Una función **suma** que recibirá un número cualquiera de parámetros (en principio, números). La función debe verificar, empleando la función **esNum**, que los argumentos recibidos son números y mostrar la suma de los argumentos. Si uno de los argumentos es incorrecto, este será ignorado. Si emplea variables, estas deberán ser locales.
- Una función **producto** que recibe un número cualquiera de parámetros (números). La función debe verificar, empleando la función **esNum**, que los argumentos recibidos son números y mostrar el producto de los argumentos. Si uno de los argumentos es incorrecto, este será ignorado. Si emplea variables, estas deberán ser locales.
- El programa principal:
  - El script **calcul.sh** recibirá como argumentos la operación que se ha de realizar, al igual que una serie de números.

```

$ calcul.sh producto 3 5 10
150
$ calcul.sh suma 3 5 10
25

```

Llamar a la función correspondiente a la operación solicitada y pasar los valores recibidos por el script. Mostrar por pantalla el resultado devuelto por la función.

Script compatible bash y ksh:

```

$ nl calcul.sh
 1  #!/bin/bash

 2  shopt -s extglob          # No funciona en ksh

 3  function esNum {
 4      if [[ $1 = ?([+-])+([0-9]) ]] ; then
 5          return 0
 6      fi
 7      return 1
 8  }

```

```

9  function suma {
10  typeset resultado=0
11  for nb in $*
12  do
13      if esNum $nb ; then
14          (( resultado += $nb ))
15      fi
16  done
17  echo $resultado
18  }

19  function producto {
20  typeset resultado=1
21  for nb in $*
22  do
23      if esNum $nb ; then
24          (( resultado *= $nb ))
25      fi
26  done
27  echo $resultado
28  }

29  if (( $# < 3 )) ; then
30  echo "Uso: $0 {suma|producto} nb1 nb2 [ ... ]" 1>&2 #
redirección de la salida de error
31  exit 1
32  fi

33  operacion=$1
34  shift

35  case $operacion in
36  producto|suma)
37      res=$(($operacion $*)
38      ;;
39  *)
40      echo "$operacion: Operación no soportada"
41      exit 1
42      ;;
43  esac
44  echo Resultado: $res

```

#### d. Ejercicio 4: archivos

Sea el siguiente archivo de datos:

```

$ cat alumnos.txt
Nombre|Clase|Promedio
Luis|6to|3
Carlos|6to|14
Clarisa|6to|16
Jorge|6to|18
Pedro|6to|8
Damian|5to|10
Daniel|5to|11
Pablo|5to|7
Victor|5to|14

```

Escriba un script **stats.sh** que muestre por pantalla, con formato, las tres columnas del archivo según sigue:

```

$ stats.sh

```

Nombre	Clase	Promedio
Luis	6to	3
Carlos	6to	14
Clarisa	6to	16
Jorge	6to	18
Pedro	6to	8
Damian	5to	10
Daniel	5to	11
Pablo	5to	7
Victor	5to	14

Script compatible Bourne, bash y ksh:

```
$ nl stats.sh
 1  #!/bin/bash  # Adaptar según el shell
 2
 3  exec 0< alumnos.txt
 4  IFS='|'
 5
 6  read entidad1 entidad2 entidad3
 7  printf "%-15s%-10s%10s\n" $entidad1 $entidad2 $entidad3
 8
 9  while read nombre clase promedio
10  do
11      printf "%-15s%-10s%10s\n" $nombre $clase $promedio
12  done
```

**e. Ejercicio 5: archivos, funciones, menú select**

Sea el siguiente archivo de datos:

```
$ cat alumnos.txt
Nombre|Clase|Promedio
Luis|6to|3
Carlos|6to|14
Clarisa|6to|16
Jorge|6to|18
Pedro|6to|8
Damian|5to|10
Daniel|5to|11
Pablo|5to|7
Victor|5to|14
```

Escriba script **stats\_select.sh** (compatible ksh/bash) que:

- Permita mostrar los archivos de una clase dada.
- Permita el cálculo del promedio de una clase dada.

El script mostrará un menú que podrá ser escrito con la estructura **select**.

Script compatible bash y ksh:

```
$ nl stats_select.sh
 1  #!/bin/bash
 2
 3  function extraccionClase {
 4      typeset criter=$1 i=1
 5      IFS='|'
 6
 7      while read nom clase promedio
 8      do
 9          if (( $i == 1 )) ; then          # Entidad
```



```

10     printf "%-15s%-10s%10s\n" $entidad1 $entidad2 $entidad3
11     elif [[ $clase = $criter ]] ; then
12         printf "%-15s%-10s%10d\n" $nom $clase $promedio
13     fi
14     (( i = $i + 1 ))
15 done < $ficIn
16 }

17 function promedioClase {
18     typeset criter=$1 i=1 suma=0
19     IFS='|'

20     while read nom clase promedio
21     do
22         if [[ $clase = $criter ]] ; then
23             echo Nota: $promedio
24             (( suma += $promedio ))
25             (( i = $i + 1 ))
26         fi
27     done < $ficIn

28     (( i-- ))
29     # Cálculo entero:
30     #echo "Promedio de la clase de $criter: $(( $suma / $i ))"
31     # Cálculo flotante:
32     echo "Promedio de la clase de $criter: " $( echo "escala=1 ;
$suma / $i" | bc )
33 }

34 # Archivo a tratar
35 ficIn=alumnos.txt
36 PS3="Su opción: "

37 # Menú
38 select item in "Extracto por clase" "Promedio de una clase"
"Fin"
39 do
40     case "$REPLY" in
41         1|2)
42             # selección entre 1 y 2: introducir la clase
43             echo -n "Clase ? "
44             read clase
45
46             # Llamar a la función adecuada
47             if [[ $REPLY = 1 ]] ; then
48                 extraccionClase $clase
49             else
50                 mediaclase $clase
51             fi
52             ;;
53         3)
54             echo "Hasta luego..."
55             exit 0
56             ;;
57         *) echo "opción incorrecta"
58             ;;
59     esac
60 done

```

## f. Ejercicio 6: archivos, tablas asociativas (bash 4, ksh93)

Obtenga el archivo **datos.txt**:

```

$ cat datos.txt
Juan Juan|46290|Valencia

```

```
Olivia Perez|24200|León  
Carlos Izaguirre|24200|León  
Alejandro Arevalo|24100|León  
Jorge Olvido|26350|La Rioja  
Jose Martinez|26350|La Rioja
```

Escriba un script **tabAsoc.sh** que cuente el número de habitantes por ciudad.

Script compatible ksh93 y bash4:

```
$ nl tabAsoc.sh  
1  #! /bin/bash  
2  # Creación de una tabla asociativa  
3  typeset -A tabCiudad  
4  IFS="|"  
  
5  while read nom cp ciudad  
6  do  
7    (( tabCiudad[$ciudad]++ ))  
8  done < datos.txt  
9  for ciudad in ${!tabCiudad[*]}  
10 do  
11   echo "$ciudad => ${tabCiudad[$ciudad]}"  
12 done
```

# Soluciones del capítulo Las expresiones regulares

## 1. Expresiones regulares

### a. Ejercicio 1: expresiones regulares con vi

Sea el archivo **expr.txt**:

```
$ cat expr.txt
felipe      10 plaza de la concordia      91.511.11.11
annie2     25/27 calle Victor Hugo      91.485.22.48
fernando   20 valencia                   96.221.33.33
cristina   avenida de la ilustración     93/455/78/52
cris       98.622.33.44
jorje      48 bravo murillo             630.22.53.48
XincX     45 plaza de neptuno          915.45.45.78
annie2    25 calle de Victor Hugo      910.48.22.48

XristiX   35/36 calle del querol       920/54/58/45
XarinX    Avda. de la marina           920.54.58.65
```

Realice las siguientes operaciones con el editor **vi** (modo ex) en el archivo **expr.txt**:

1. Los números de teléfono que terminen en 48 deberán en adelante terminar en 50.

```
:1,$s/48$/50/
```

2. En la línea 2, sustituya la cadena "annie2" por "annie" (sin desplazarse a la línea 2).

```
:2s/annie2/annie/
```

3. Sustituir las líneas vacías por "RAS".

```
:1,$s/^$/RAS/
```

4. Sustituya todos los nombres que comiencen y terminen con la letra X por XxxxxxxX.

```
:1,$s/X[ ^ \ \ ]*X/XxxxxxxX/
```

5. Sustituya cada cifra al final de la línea por 0, salvo si esta cifra es igual a 8.

```
:1,$s/[0-79]$/0/
```

6. Sustituya los "/" separadores de los números de teléfono por ".".

```
:1,$s/\(..\)\(..\)\(..\)\(..\)$/\1.\2.\3.\4/
```

7. Los campos deberán estar separados por "|" y no por tabulaciones o espacios.

```
:1,$s/[ \t ]+ [|] */|/g
```

8. Inserte un carácter "|" al principio y al final de cada línea.

```
:1,$s/^\(.*\)$/|1|/
```

## b. Ejercicio 2: grep

Partiendo del archivo **php.ini** proporcionado:

1. Muestre las líneas que comienzan con "mysql".

```
$ grep '^mysql' php.ini
```

2. Muestre las líneas que terminan con "On".

```
$ grep 'On$' php.ini
```

3. Muestre las líneas que terminan con "On" y que no tengan un ";" en la primera posición.

```
$ grep '^^[^;]*On$' php.ini
```

4. Muestre las líneas que terminan en On (sin diferenciar mayúsculas y minúsculas).

```
$ grep -i 'On$' php.ini
```

5. Combinando **find** y **grep**, busque los archivos regulares que contengan la palabra "tr". Las cadenas "<tr>" o "</tr>" no deben ser devueltas.

```
$ find rep -type f -exec grep '[^</]\<tr\>' {} /dev/null \;
```

**/dev/null** permite pasar un segundo nombre de archivo al comando **grep**. Como **grep** recibe varios nombres de archivo como argumento, este muestra el nombre de archivo correspondiente a la línea encontrada.

# Soluciones del capítulo El comando sed

## 1. Expresiones regulares

### a. Ejercicio 1: inserción de marcadores en un archivo

Sea el archivo **fechas\_curs.txt**:

```
$ cat fechas_curs.txt
unix
28-30 ene
17-19 jun
18-20 nov

shell
23 mar
15 jul
7 sep
```

Empleando el comando **sed**, transforme este archivo de la siguiente forma:

```
unix
<date>28-30 ene</date>
<date>17-19 jun</date>
<date>18-20 nov</date>

shell
<date>23 mar</date>
<date>15 jul</date>
<date>7 sep</date>
```

El comando **sed**:

```
$ sed 's/^[0-9].*$/<date>&</date>/' fechas_curs.txt
```

o

```
$ sed 's/^\([0-9].*\)$/<date>\1</date>/' fechas_curs.txt
```

### b. Ejercicio 2: formato de archivos

Tome el archivo **.bash\_profile**. Muestre el archivo con el comando **nl**, que numera las líneas:

```
$ nl .bash_profile
 1 # .bash_profile

 2 # Get the aliases and functions
 3 if [ -f ~/.bashrc ]; then
 4     . ~/.bashrc
 5 fi

 6 # User specific environment and startup programs
 7 PATH=$PATH:$HOME/bin
```

1. Elimine los espacios que preceden a los números de línea.

```
$ nl .bash_profile | sed 's/^ \ \ \ \ //'
```

o

```
$ nl .bash_profile | sed 's/^ \ {4\}///'
1      # .bash_profile
2      # Get the aliases and functions
3      if [ -f ~/.bashrc ]; then
4          . ~/.bashrc
5      fi
6      # User specific environment and startup programs
7      PATH=$PATH:$HOME/bin
```

2. Ponga también el número de línea entre corchetes.

```
$ nl .bash_profile | sed -e 's/^ \ \ \ \ //' -e 's/\([0-9][0-9]*\)/[\1]/'
[1]    # .bash_profile
[2]    # Get the aliases and functions
[3]    if [ -f ~/.bashrc ]; then
[4]        . ~/.bashrc
[5]    fi
[6]    # User specific environment and startup programs
[7]    PATH=$PATH:$HOME/bin
```

# Soluciones del capítulo El lenguaje de programación awk

## 1. awk en línea de comandos

### a. Ejercicio 1: awk y otros filtros

Muestre los nombres de los archivos de texto del directorio `/etc`.

```
# Efectuado en Linux
$ cd /etc
$ file * | grep text | awk '{print $1}' | sed 's/:$//'
adjtime
aliases
asound.conf
auto.master
auto.misc
. . .
```

### b. Ejercicio 2: criterios de selección

1. En su directorio actual, muestre las características de los archivos cuyo nombre comience con un punto (Solo estos).

```
$ ls -la | awk '$9 ~ /^\./ {print}'
```

2. En su directorio actual, muestre los nombres de los archivos que comienzan con un punto, salvo "." y "..".

```
$ ls -a | awk '/^\.[^.] / {print}'
```

### c. Ejercicio 3: criterios de selección, visualización de campos, secciones BEGIN y END

A partir del archivo `php.ini` proporcionado:

1. Muestre las líneas que no comiencen por ";" y que terminen en On u Off.

```
$ awk '/^[^;].*(On|Off)$/ { print }' php.ini
engine = On
short_open_tag = Off
asp_tags = Off
zlib.output_compression = Off
implicit_flush = Off
```

2. Mejore la visualización.

```
$ awk -F '=' '/^[^;].*(On|Off)$/ { printf("%-40s%-10s\n", $1, $2)
}' php.ini
engine                               On
short_open_tag                       Off
asp_tags                              Off
zlib.output_compression              Off
implicit_flush                       Off
zend.enable_gc                       On
. . .
```

- Recupere el comando anterior y muestre el número de directivas encontradas en el total de las líneas.

```
$ awk -F '=' 'BEGIN { nbDir=0 } /^[^;]*.(On|Off)$/ {
printf("%-40s%-10s\n",$1,$2) ; nbDir++} END{ print nbDir "
directivas encontradas en " NR " líneas }' php.ini

. . .
mysql.secure_connection          Off
tidy.clean_output                Off
42 directivas encontradas en 1974 líneas
```

## 2. Scripts awk

### a. Ejercicio 4: funciones

Recupere el archivo **Ficha.php** (extracto):

```
$ nl Fichas.php
 1 <?php
   . . .
10 class Ficha
11 {
12     /**
13      * @var integer
14      *
15      * @ORM\Column(name="fic_id", type="integer", nullable=false)
16      * @ORM\Id
17      * @ORM\GeneratedValue(strategy="IDENTITY")
18      */
19     private $ficId;

20     /**
21      * @var string
22      *
23      * @ORM\Column(name="fic_description_faits_consequences",
type="text", nullable=false)
24      */
25     private $ficDescriptionFaitsConsequences;

26     /**
27      * @var string
28      *
29      * @ORM\Column(name="fic_action_immediate", type="text",
nullable=true)
30      */
31     private $ficActionImmediate;
```

Prepare un script **delprefix.awk** que efectúe las operaciones siguientes:

- Las líneas que comiencen por **private** deben modificarse:
  - Las tres primeras letras del nombre de la variable (prefijo) deben eliminarse, sea cual sea su valor (en nuestro caso, el prefijo es "fic", pero podría ser cualquier cosa).
  - La letra que siga al signo \$ debe estar en minúsculas.

Después de ejecutar el script, el archivo deberá verse como sigue (extracto):

```
1 <?php
```



```

. . .
10 class Ficha
11 {
12     /**
13      * @var integer
14      *
15      * @ORM\Column(name="fic_id", type="integer", nullable=false)
16      * @ORM\Id
17      * @ORM\GeneratedValue(strategy="IDENTITY")
18      */
19     private $id;

20     /**
21      * @var string
22      *
23      * @ORM\Column(name="fic_description_faits_consequences",
24      type="text", nullable=false)
25      */
26     private $descriptionFaitsConsequences;

27     /**
28      * @var string
29      *
30      * @ORM\Column(name="fic_action_immediate", type="text",
31      nullable=true)
32      */
33     private $actionImmediate;

```

El script **delprefix.awk**:

```

$ nl delprefix.awk
1  #! /usr/bin/awk -f

2  # Las líneas "private $ficId;"
3  /private \${a-z}/ {
4  # Obtener los 3 primeros caracteres del prefijo
5  sub(/\$.../, "$", $0) ;

6  # Recuperar los trozos de cada lado del $
7  split($0, tab, "\$") ;
8  # Recuperar el carácter que está después del $
9  print tab[1] "$" tolower(substr(tab[2], 1, 1)) substr(tab[2], 2)
10
11  next
12  }
13  {
14  print $0
15  }

```

Ejecución del script **awk**:

```
$ awk -f delprefix.awk Ficha.php
```

o

```
$ delprefix.awk Ficha.php
```

## b. Ejercicio 5: análisis de un archivo de log

Funcionalidades implementadas: secciones **BEGIN**, **END**, argumentos de la línea de comandos, tablas, funciones.

Sea el archivo de log de sistema **error.log**:

```
$ cat error.log
[Mon Sep 30 09:33:00 2013] [notice] Apache/2.2 (Unix)
(Red-Hat/Linux) mod_python/2.7.6 Python/1.5.2 mod_ssl/2.8.4
OpenSSL/0.9.6b DAV/1.0.2 PHP/4.0.6 mod_perl/1.24_01
mod_throttle/3.1.2 configured -- resuming normal operations
[Mon Sep 30 09:33:00 2013] [notice] suEXEC mechanism enabled
(wrapper: /usr/sbin/suexec)
[Mon Sep 30 18:35:34 2013] [notice] caught SIGTERM, shutting down
[Tue Oct 1 10:06:46 2013] [alert] httpd: Could not determine the
server's fully qualified domain name, using 10.0.0.66 for ServerName
[Tue Oct 1 10:06:46 2013] [notice] Apache/2.2 (Unix) (Red-Hat/Linux)
mod_python/2.7.6 Python/1.5.2 mod_ssl/2.8.4
OpenSSL/0.9.6b DAV/1.0.2 PHP/4.0.6 mod_perl/1.24_01
mod_throttle/3.1.2 configured -- resuming normal operations
[Tue Oct 1 10:06:46 2013] [notice] suEXEC mechanism enabled
(wrapper: /usr/sbin/suexec)
[Tue Oct 1 10:12:51 2013] [notice] SIGHUP received.
Attempting to restart
[Tue Oct 1 10:12:52 2013] [alert] httpd: Could not determine
the server's fully qualified domain name, using 10.0.0.66
for ServerName
[Tue Oct 1 10:12:52 2013] [notice] Apache/2.2 (Unix) (Red-Hat/Linux)
mod_python/2.7.6 Python/1.5.2 mod_ssl/2.8.4
OpenSSL/0.9.6b DAV/1.0.2 PHP/4.0.6 mod_perl/1.24_01
mod_throttle/3.1.2 configured -- resuming normal operations
[Tue Oct 1 10:12:52 2013] [notice] suEXEC mechanism enabled
(wrapper: /usr/sbin/suexec)
```

Prepare un script **error\_log.awk** que permita buscar un tipo de mensaje (**alert** o **notice**) y mostrarlos por pantalla de una forma que resulte más fácil de leer. El tipo de mensaje que se ha de buscar se pasa como argumento en la línea de comandos.

```
$ awk -f error_log.awk alert error.log
```

El script **error\_log.awk**:

```
$ nl error_log.awk
 1 BEGIN {
 2     # Comprobar el número de argumentos
 3     if (ARGC != 3) {
 4         printf("Usa: awk -f error_log.awk tipo_mensaje
archivo_log\n") | "cat 1>&2"
 5         exit 1
 6     }
 7
 8     tipoMensaje = ARGV[1]
 9     cadenaMensaje = "\\[" ARGV[1] "\\]"
10     delete ARGV[1]
11     print "LISTA DE LOS MENSAJES DE TIPO: " , tipoMensaje
12
13 }
14
15 $0 ~ cadenaMensaje {
16     # Usar el | como separador de campos
17     sub(/^\[/, "|")
18     sub(/\] \[/, "|")
19     sub(/\] /, "|");
20     split($0, linea, "|")
21
22     print "Fecha: " linea[1]
```

```
23     print linea[3]
24 }
```

### c. Ejercicio 6: generación de un archivo de etiquetas

Funcionalidades implementadas: secciones **BEGIN**, **END**, argumentos de la línea de comandos, tablas, funciones awk.

Recupere el archivo **contactos.txt**:

```
$ cat contactos.txt
Corbalán, Marina
    08003 Barcelona
    933.221.506

Romero, Roberto
    48002 Bilbao
    944.301.265

García, Esteban
    46002 Valencia
    962.318.376

Beltrán, Francisco
    35001 Las Palmas de G.C.
    928.161.827

Revuelta, Carmelo
    35003 Las Palmas de G.C.
    928.532.111
```

Prepare un script **etiq.awk** que deberá devolver el archivo **contactos.txt** en forma de etiquetas. A continuación, el resultado que deberá proporcionar el script:

```
$ awk -f etiq.awk contactos.txt
Corbalán, Marina           Romero, Roberto
08003 Barcelona           48002 Bilbao
933.221.506                944.301.265

García, Esteban           Beltrán, Francisco
46002 Valencia            35001 Las Palmas de G.C.
962.318.376                928.161.827

Revuelta, Carmelo
35003 Las Palmas de G.C.
928.532.111

$
```

El script **etiq.awk**:

```
$ nl etiq.awk
 1 BEGIN{
 2     indEtiqueta = 1
 3 }
 4 # Seleccionar la línea del nombre
 5 ($0 !~ /^$/ && $0 !~ /^t/ ) {
 6     # Guardar el nombre
 7     nombre[indEtiqueta] = $0
 8     # Leer línea de código postal / población
 9     # y guardarlos
10     getline
```

```
11     gsub (/^t/ , "" , $0)
12     adr[indEtiqueta] = $0
13
14     # Leer línea del teléfono y guardarlo
15     getline
16     gsub (/^t/ , "" , $0)
17     tel[indEtiqueta] = $0
18
19     indEtiqueta++
20 }
21
22 END {
23     últimoInd = indEtiqueta - 1
24     # Visualización
25     for (i=1 ; i <= últimoInd ; i=i+2) {
26         printf("%d\n",i);
27         printf("%-30s%-30s\n", nombre[i], nombre[i+1])
28         printf("%-30s%-30s\n", adr[i], adr[i+1])
29         printf("%-30s%-30s\n", tel[i], tel[i+1])
30     }
31 }
```

## Caracteres especiales de shell

Caracteres	sh (Bourne)	ksh	bash	Significado
espacio tabulación salto de línea	sí	sí	sí	Separadores de palabras en la línea de comandos.
&	sí	sí	sí	Segundo plano.
<< >>	sí	sí	sí	Tubería y redirecciones.
>	no	sí	sí	Sobrescritura de un archivo (opción <b>noclobber</b> ).
(cmdo1;cmdo2) {cmdo1;cmdo2}	sí	sí	sí	Agrupación de comandos.
;	sí	sí	sí	Separador de comandos.
* ? [ ]	sí	sí	sí	Expresiones para nombres de archivo y <b>case</b> .
* ? [ ]	no	sí	sí	Expresiones para el comando de test [[...]].
?() +() *() !() @()	no	sí	sí	Expresiones para nombres de archivo, [[ ]] y <b>case</b> .
\$ y \${ }	sí	sí	sí	Valor de una variable.
`...`	sí	sí	sí	Sustitución de comandos.
\$()	no	sí	sí	Sustitución de comandos.
'...' "..." \	sí	sí	sí	Caracteres de protección.
\$(( ))	no	sí	sí	Sustitución de expresión aritmética.
cmdo1 && cmdo2 cmdo1    cmdo2	sí	sí	sí	Operadores lógicos de shell.
#	sí	sí	sí	Comentario.
~	no	sí	sí	Directorio de inicio del usuario.

## Comandos internos de shell

Comando	sh ( Bourne)	ksh	bash	Significado
! cmdo	no	no	sí	Negación del código de retorno de un comando.
(( ))	no	sí	sí	Aritmética.
. script	sí	sí	sí	Provocar la ejecución de un script por el shell actual.
:	sí	sí	sí	Devuelve siempre verdadero.
[...]	sí	sí	sí	Evaluaciones.
[[..]]	no	sí	sí	Evaluaciones.
= (var=valor)	sí	sí	sí	Asignación de una variable.
alias	no	sí	sí	Creación, visualización de un alias.
bg	no	sí	sí	Pasar un proceso en segundo plano.
case	sí	sí	sí	Estructura de control.
cd	sí	sí	sí	Cambio de directorio.
echo	sí	sí	sí	Escritura en la salida.
env	sí	sí	sí	Lista de las variables exportadas.
eval	sí	sí	sí	Doble evaluación.
exec	sí	sí	sí	Redirecciones/remplazo del shell por un comando.
exit	sí	sí	sí	Salir del shell.
export	sí	sí	sí	Exportación de una variable.
expr	sí	sí	sí	Aritmética.
f1()	sí	sí	sí	Definición de una función.
fc	no	sí	sí	Historial de comandos.
fg	no	sí	sí	Pasar un proceso en primer plano.
for	sí	sí	sí	Estructura de control.
function f1	no	sí	sí	Definición de una función.
getopts	sí	sí	sí	Análisis de las opciones de un script.
if	sí	sí	sí	Estructura de control.
jobs	no	sí	sí	Lista de los procesos en segundo plano.
kill	sí	sí	sí	Envío de signal.
let	no	sí	sí	Aritmética.

newgrp	sí	sí	no	Cambio de grupo.
print	no	sí	no	Escritura en la salida.
pwd	sí	sí	sí	Directorio actual.
read	sí	sí	sí	Lectura.
readonly	sí	sí	sí	Definición de una variable de solo lectura.
return	sí	sí	sí	Salir de una función.
select	no	sí	sí	Estructura de control.
set	sí	sí	sí	Lista de variables.
set e1 e2 e3	sí	sí	sí	Asignación de parámetros posicionales.
set -o	no	sí	sí	Gestión de opciones.
shift	sí	sí	sí	Desplazamiento de argumentos.
test	sí	sí	sí	Evaluaciones.
time cmdo cmdo	no	sí	sí	Tiempo de ejecución de comandos conectados con una tubería.
times cmdo	sí	sí	sí	Tiempo de ejecución de un comando.
trap	sí	sí	sí	Gestión de signals.
typeset	no	sí	sí	Definición de propiedades de una variable.
ulimit	sí	sí	sí	Límites de recursos.
umask	sí	sí	sí	Gestión de umask.
unalias	no	sí	sí	Borrado de un alias.
unset	sí	sí	sí	Borrado de una variable.
until	sí	sí	sí	Estructura de control.
wait	sí	sí	sí	Esperar la muerte de un proceso.
whence	no	sí	no	Indica si un comando es interno o externo.
while	sí	sí	sí	Estructura de control.

➤ La "!" que invierte el código de retorno de un comando no se describe en ksh 88, pero en la práctica funciona.

## Orden de interpretación de un comando

Shell interpreta un comando en el orden siguiente:

- Aislamiento de palabras separadas por los caracteres espacio, tabulación o salto de línea.
- Tratamiento de los caracteres de protección (' ', " ", \).
- Sustitución de variables (\$).
- Sustitución de comandos (` ` \$()).
- Sustitución de caracteres de generación de nombres de archivo (\*, ?, [], etc.).
- Tratamiento de tuberías y redirecciones.
- La primera palabra resultante de las sustituciones anteriores se considera el comando que se ejecutará. El shell busca primero el comando en su lista de alias. Si no lo encuentra, lo busca entre sus comandos internos. Si tampoco lo encuentra, lo busca en la lista de funciones que se hayan definido. Finalmente, si tampoco lo encuentra, busca un comando externo usando la variable PATH.



## Programación shell en Unix/Linux sh, ksh, bash (con ejercicios corregidos)

Este libro de **programación shell** va dirigido a usuarios y administradores de sistemas Unix/Linux que desean aprender a programar **scripts shell**. Se detallan las funcionalidades de tres shells usados habitualmente (**Bourne Shell, ksh 88 y 93, bash**) y sus diferencias. Los conceptos se presentan de manera progresiva y pedagógica, convirtiendo este libro en un soporte ideal destinado tanto a la **formación profesional como a la autoformación**.

Los primeros capítulos se destinan al **funcionamiento del shell**: ejecución de un comando, caracteres especiales del shell usados habitualmente (**redirecciones, tuberías,...**), configuración del entorno de trabajo del usuario (variables y archivos de entorno,...). Los **mecanismos internos** se explican detalladamente y se ilustran con múltiples esquemas.

A continuación, el libro se centra en la programación propiamente dicha. Las **bases de la programación (variables, estructuras de control, comandos de verificación y cálculo,...**) se presentan e ilustran mediante una gran cantidad de ejemplos y, más adelante, se detallan los aspectos avanzados de la programación shell (**gestión de archivos, funciones,...**).

La última parte trata sobre las **utilidades anexas** indispensables para el tratamiento de cadenas de caracteres y de archivos de texto: **las expresiones regulares básicas y extendidas, el editor no interactivo sed**, una visión extendida **del lenguaje awk** y los principales comandos filtro de los sistemas unix.

Los **ejercicios** permitirán al lector practicar la escritura de scripts Shell a lo largo de todo el libro.

Los **ejemplos de scripts shells** incluidos en el libro se pueden descargar en [www.ediciones-eni.com](http://www.ediciones-eni.com).



En [www.ediciones-eni.com](http://www.ediciones-eni.com):

- Ejemplos de scripts del libro.
- Los archivos de ejercicios y su solución.



[www.ediciones-eni.com](http://www.ediciones-eni.com)

### Christine DEFFAIX RÉMY

Ingeniera informática en la empresa Ociensa Technologies, especialista en la administración y desarrollo en Unix y Linux, **Christine Deffaix Rémy** interviene en tareas de desarrollo y formación en grandes cuentas. Su sólida experiencia junto a sus cualidades pedagógicas proporcionan una obra realmente eficaz para el aprendizaje en la creación de scripts shell.

### Los capítulos del libro

Prólogo • Introducción • Mecanismos esenciales del shell • Configuración del entorno de trabajo • Las bases de la programación shell • Aspectos avanzados de la programación shell • Expresiones regulares • El comando sed • El lenguaje de programación awk • Los comandos filtro • Soluciones a los ejercicios • Anexos

Para más información:



ISBN : 978-2-7460-9615-8



9 782746 096158