ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

Técnicas de Programación de la Shell de Linux para Administradores

Málaga, Noviembre de 2005

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERO EN INFORMÁTICA

Técnicas de Programación de la Shell de Linux para Administradores

Realizado por

PEDRO LOPEZ SOTO

Dirigido por

FRANCISCO R. VILLATORO MACHUCA

y

LAWRENCE MANDOW ANDALUZ

Departamento

DEPARTAMENTO DE LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

UNIVERSIDAD DE MÁLAGA

MÁLAGA, NOVIEMBRE 2005

UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERO EN INFORMÁTICA

Reunido el tribunal examinador en el día de la fecha, constituido por:

Presidente Dº;/Dª._____

Secretario Dº;/Dª._____

Vocal Dº;/Dª. _____

para juzgar el proyecto Fin de Carrera titulado:

Técnicas de Programación de la Shell de Linux para Administradores

del alumno Dº;/Dª. **Pedro López Soto**
dirigido por Dº;/Dª. **Francisco R. Villatoro Machuca y Lawrence Mandow Andaluz**

ACORDÓ POR _____ OTORGAR LA CALIFICACIÓN DE

_____

Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS COMPARECIENTES DEL TRIBUNAL, LA PRESENTE DILIGENCIA.

Málaga, a _____de _____de _____

El Presidente _____El Secretario_____ El Vocal_____

Fdo: _____Fdo:_____ Fdo:_____

Índice de Contenidos

Capítulo 1. Introducción....1
1.1. Objetivos del Proyecto....1
1.2. Contenidos de la Memoria....3
Capítulo 2. Estudio Histórico y Práctico de la Evolución de las Shells....5
2.1. UNIX. Un Comienzo....5
2.2. Shell. Una Evolución....10
2.2.1 Shell Bourne. El Gran 'Daddy'....12
2.2.2 C Shell. El Valor de ser Diferente....13
2.2.3 Shell Korn. El Retorno a las Tradiciones....14
2.2.4 Shell POSIX. Adaptándose al Estándar....16
2.2.5 Shell Bash. En Busca de la Libertad....17
2.3. Conclusiones....19
Capítulo 3. Comparativa Funcional de las Shells de Unix....21
3.1. Criterios Técnicos....21
3.1.1 Histórico de Comandos....21
3.1.2 Edición de la Línea de Comandos....24
3.1.3 Seguimiento Invisible de Enlaces Simbólicos....25
3.1.4 Completado de Nombres de Ficheros, Comandos, Usuarios, Hosts y Variables....26
3.1.5 Facilidades para el uso del Prompt de Usuario....30
3.1.6 Evaluación de Expresiones Aritméticas y Lógicas....33
3.1.7 Manejo de Variables....40
3.1.8 Sentencias de Control de Flujo de la Ejecución....47
3.1.9 Redirección de Entrada y Salida. Pipes....50
3.1.10 Funciones....54
3.1.11 Shells Libres....55
3.1.12 Diferencias Entre Shell Posix y Shell Korn....56
3.2. Conclusiones....57
Capítulo 4. Técnicas Básicas de la Programación Shell....61
4.1. Comenzando a Crear un Script....61
4.1.1 Ejecutar un Script....63
4.1.2 Comentarios....66
4.1.3 Uso de Variables en un Script. Arrays. Variables Especiales....66
4.2. Comunicación con un Script....72
4.2.1 Argumentos de un Script....72
4.2.2 Sentencia `read`....73
4.2.3 Opciones y Argumentos con `getopts`....74
4.3. Variables. Aritmética, Lógica y Cadenas....78
4.3.1 Aritmética y Lógica....78
4.3.2 Manejo de Cadenas....78
4.4. Condiciones....79
4.4.1 Cortocircuitos....79
4.4.2 Condicional `if`....81
4.4.3 Condicional `case`....82
4.5. Bucles....84
4.5.1 `while` y `until`....84
4.5.2 `for`....85
4.5.3 `select`....87
4.5.4 `break` y `continue`....89
4.6. Funciones....89
4.6.1 Definición de Funciones....89
4.6.2 Argumentos y Entorno de Funciones....90
4.6.3 Finalización de Funciones....91
4.6.4 Librerías de Funciones....91
4.7. Flujo de Datos. Entrada y Salida....92

4.8. Traps y Señales....93
Capítulo 5. Expresiones Regulares, sed y awk....97
5.1. Introducción....97
5.2. Expresiones regulares. grep....97
5.3. Comando sed....100
5.3.1 sustituir....101
5.3.2 Borrar....103
5.3.3 Imprimir....104
5.3.4 Leer....105
5.3.5 Escribir....105
5.4. Comando awk....106
5.4.1 Procesamiento de líneas en awk....107
5.4.2 Campos definidos dentro de una línea....108
5.4.3 Operadores en los patrones....109
5.4.4 Procesamiento pre-entrada y post-entrada....109
5.4.5 Impresión de valores con awk. Salida formateada....110
5.4.6 Variables....111
5.4.7 Operadores y funciones....111
5.5. Programas sed y awk....112
Capítulo 6. Aplicaciones para administración con la Shell Bash....115
6.1. Análisis del Número de Argumentos de un Script: argumentos.sh....116
6.2. Bombardeo con Mails a un Servidor de Correo: mailbomb.sh....116
6.3. Backups del Sistema en Cdrom: cdbackup.sh....117
6.4. Grabación en cd de una Imagen de Cdrom en formato .iso: grabacd.sh....119
6.5. Representación de un Cursor en Pantalla: cursor.sh....120
6.6. Ejemplo de uso del Script cursor.sh: usacursor.sh....121
6.7. Escáner de red: escanea_ip.sh....121
6.8. Escáner de red: escanea_ip_2.sh....122
6.9. Backup y Eliminación de Ficheros log: logrotate.sh....123
6.10. Conversión de Nombres de Ficheros a Caracteres Minúsculas: minusculas.sh....125
6.11. Desglose de un Nombre Completo de directorio o Fichero en sus Componentes: path.sh 126
6.12. Difusión de un Correo a Múltiples Usuarios: broadmail.sh....127
6.13. Desglose de un Fichero de Base de Datos en sus Componentes: cortafichero.sh....128
6.14. Ejecución de un ftp a una Máquina Remota sin Intervención Manual : ftpremoto.sh....130
6.15. Renombrado de Ficheros de Imágenes en Formato .jpg: renombra.sh....131
6.16. Gestión de Paquetes de Instalación en Formato .rpm: maneja_rpm.sh....132
6.17. Petición de Usuario y Contraseña Para Scripts: login.sh....137
6.18. Eliminación de Segmentos de Memoria Compartida en el Sistema: rm_shmem.sh....138
6.19. Selección de la Shell Para una Sesión Interactiva de Trabajo: selecciona_shell.sh..139
6.20. Agenda Personal: agenda.sh....141
6.21. Creación de una Cuenta de Usuario: nuevousuario.sh....143
6.22. Listado de los Usuarios de un Sistema: usuarios.sh....145
6.23. Ejemplo de Sentencia getopts: opciones.sh....146
6.24. Distribución de Ficheros en Múltiples Servidores: copia_array_remoto.sh....147
6.25. Kit Para Implantación de Aplicaciones en Servidores en Cluster: clustertoolkit.sh.148
6.26. Conversión de Ficheros en Formato .lj a Postscript y a Formato Pdf: lj-ps-pdf.sh....150
6.27. Generación de un Informe de Totales: formatea.awk....151
Capítulo 7. Conclusiones....153
BIBLIOGRAFíA....155
Apéndice A. Contenidos del CD adjunto....159

CAPÍTULO 1. INTRODUCCIÓN

1.1. Objetivos del Proyecto

En la formación teórica y práctica en Sistemas Operativos en nuestra Escuela, se hace patente la necesidad de contemplar algunos aspectos relacionados con la administración de sistemas que son importantes desde un punto de vista aplicado en los entornos empresariales [1,2]. Uno de dichos aspectos es la programación de ficheros script en las shells de los sistemas operativos Unix [3,4]. Desde el punto de vista de un administrador, el conocimiento del lenguaje de programación shell es de gran importancia para entender los scripts que el sistema utiliza de forma estándar, así como para facilitar la programación de los suyos propios [5]. Generalmente, el administrador de dichos sistemas suele hacerse con una batería de dichos scripts de mayor o menor complejidad, hechos por él mismo, o basados en otras fuentes, que facilitarán las tareas más rutinarias en el trabajo diario. Es este sentido práctico el que nos sirve como eje principal para el desarrollo del presente proyecto.

Ante todo, hemos de indicar que la diversidad de implementaciones comerciales de entornos Unix ha hecho que difieran significativamente en cuanto a los tipos de shell que se usan en ellos [3,4]. Pongamos algunos ejemplos. La Shell Bourne, la más rudimentaria, primera de todas, sigue siendo la base de algunos sistemas Unix, como Solaris (Sun Microsystems), que la usa aún por defecto en sus últimas versiones. Históricamente, también tenemos la C Shell, que aunque no hay entornos que la usen habitualmente por defecto, sigue siendo aún muy usada por aquellos administradores más familiarizados con el lenguaje C. Otra shell muy extendida, por ejemplo en entornos Solaris, como alternativa a la Shell Bourne, es la Shell Korn, más avanzada en múltiples aspectos, y además, compatible con la anterior. En HP-UX (Hewlett Packard) se ha implementado una alternativa a la Shell Korn, aunque muy parecida, denominada Shell POSIX, cuyo nombre alude a los estándares POSIX (Portable Operating System Interface) promovidos por el IEEE con el fin de conseguir la portabilidad de los sistemas Unix. En Linux, donde están disponibles varias shells debido al ingente esfuerzo de la gente que colabora en dicho

proyecto, la más popular es la shell Bash. Hoy en día podemos considerar esta shell como estándar en Linux. En este proyecto de fin de carrera nos centraremos en Linux con esta última, aunque también se mencionarán, de forma comparativa, detalles de las demás.

Tres son las aportaciones básicas de este proyecto. En primer lugar se presentará un tutorial de las técnicas básicas de programación de las shells, incluyendo variables de entorno, tratamiento de cadenas, aritmética, estructuras de control, etc. Algunas shells no tienen alguna de estas características [3,4]. La novedad de este proyecto radica en que, aunque nos centraremos en la Shell Bash, también indicaremos las diferencias más significativas entre ésta y las demás (Bourne, C Shell, Korn y POSIX). Esta comparación, se hará desde un punto de vista histórico, observando la evolución que han tenido las shells desde los orígenes de los entornos Unix, y práctico, comparando las características que convierten a una shell en más ventajosa sobre otras en ciertos aspectos.

En este tutorial se presentarán ejemplos sacados de los scripts básicos que los sistemas operativos tienen, como los de arranque (boot). Algunos de estos scripts serán analizados en detalle como ejemplos de programación shell para administradores. Debemos tener en cuenta en esta fase del proyecto, que la shell es algo inherente y ligado al propio sistema operativo, por lo tanto, tendremos que hablar de temas como tratamiento de procesos, entorno de usuario, etc., que en principio no parecen tener nada que ver con el objetivo, pero que, como se verá en su momento, sí tienen su relación.

En segundo lugar se estudiará la aplicación de la shell para automatizar las tareas básicas de los administradores, que abarcan diferentes campos, como son los backup periódicos, la sincronización de sistemas de ficheros de forma básica, las actualizaciones de software, la gestión de conexiones remotas automáticas para realizar tareas en otros sistemas, y algunos ejemplos prácticos usados en hacking y seguridad. A lo largo del desarrollo del proyecto, también se hará una labor de investigación para ver cuáles son las tareas en las que hoy en día se usa más la programación shell, y se desarrollarán nuevos scripts para aportar soluciones.

En tercer y último lugar se estudiarán los procesadores *sed* y *awk* [6,7]. Aunque no son shells interactivas, merecen ser tenidos en cuenta en este proyecto, ya que estas herramientas son imprescindibles para los administradores, en especial para la generación de informes basándose en una salida de resultados. Se estudiarán sus características funcionales básicas y se presentarán múltiples ejemplos de sus aplicaciones en administración de sistemas.

1.2. Contenidos de la Memoria

Este proyecto se estructura en los siguientes capítulos:

Capítulo 2: Estudio histórico y práctico de la evolución de las shells a partir de la literatura, pasando por las principales implementaciones que se han realizado: Bourne, C shell, Korn, POSIX y Bash.

Capítulo 3: Comparativa a nivel funcional de las diferentes shells de Unix haciendo hincapié en las diferencias y mejoras que presentan cada una de ellas.

Capítulo 4: Estudio de scripts básicos de los sistemas operativos Unix y Linux, comentando en detalle las técnicas utilizadas a modo de tutorial, junto con su funcionalidad.

Capítulo 5: Estudio de los procesadores *sed* y *awk*, y desarrollo de una serie de ejemplos basados en ellos, útiles para administradores.

Capítulo 6: Desarrollo de una serie de aplicaciones "elementales" útiles para la administración de sistemas utilizando la Shell Bash.

Capítulo 7: Finalmente, se extraerán conclusiones a partir del estudio realizado, con énfasis en la comparación crítica de las diferentes shells estudiadas, de las ventajas

ofrecidas por los procesadores *sed* y *awk*, así como de las diferencias más significativas entre los scripts del sistema entre los entornos Unix de SUN, HP y Linux.

Bibliografía

Contenidos del cdrom: Descripción del material aportado en el cdrom adjunto

Los medios materiales que se han utilizado son tres máquinas UNIX, un equipo Sun Microsystems con Sistema operativo Solaris, uno Hewlett Packard con HPUX y un PC bajo Linux.

CAPÍTULO 2. ESTUDIO HISTÓRICO Y PRÁCTICO DE LA EVOLUCIÓN DE LAS SHELLS

A beginning is the time for taking the most delicate care that the balances are correct.
(Dune. Frank Herbert)

2.1. UNIX. Un Comienzo

No podemos empezar a estudiar la historia y la evolución de las shells sin tener en cuenta el marco en que se integran que es, por supuesto, los entornos operativos Unix. Las fuentes que hemos utilizado principalmente para componer esta pequeña historia son los archivos publicados en la web de Bell Labs: www.bell-labs.com, empresa y cuna de los entornos UNIX actuales.

Situamos los orígenes de Unix a finales de la década de los 60 en los laboratorios Bell. En 1966, un proyecto conjunto de General Electric (GE), Bell Laboratories y el Instituto Tecnológico de Massachusetts (MIT) pretendía desarrollar y mejorar el software de tiempo compartido. En este proyecto se situaba MULTICS (Información Multiplexada y Sistema de Computación), un prototipo de sistema operativo que pretendía compartir información entre usuarios con un sistema de seguridad. Fue implementado en una computadora GE 645. Este proyecto era una apuesta importante de Bell Labs, sin embargo, presentaba muchos problemas, como lentitud y muchos requisitos de memoria, y sufrió bastantes retrasos en el desarrollo, lo que le llevó a tener un futuro incierto.

A juicio de Dennis Ritchie [8], la decadencia y caída de MULTICS fue uno de los elementos determinantes para la creación de UNIX. Ken Thompson, Rud Canaday, Dough Mcllroy, Joe Ossana y Dennis Ritchie eran integrantes del equipo que trabajaba en dicho proyecto. Estos veían cómo sus esperanzas en conseguir un sistema operativo de tiempo compartido iban disminuyendo. Así, en 1969 empezaron a buscar alternativas a MULTICS. Ante el temor de que se produjera otro fracaso como el ocurrido con MULTICS, Bell Labs no se comprometía a aportar computadoras para otros desarrollos,

debido a esto, las tentativas que se hicieron acabaron todas frustradas. Finalmente, todo comenzó con un juego.

En 1969, K. Thompson creó "Space Travel". Una especie de divertimento que simulaba el sistema Solar y una nave viajando por él, que controlaba el jugador. Primero fue programado en MULTICS y luego en Fortran para GECOS (Sistema Operativo para GE 635, más tarde Honeywell). Estas dos versiones del juego fueron un fracaso por el coste de tiempo de procesador en una máquina cara, como la GE 635, y por la mala interactividad con la consola, ya que todo se hacía lanzando comandos a través de ésta. Finalmente, encontraron una pequeña computadora PDP 7 de DEC (Digital Equipment, posteriormente Compaq, y hoy parte de HP) con buenas capacidades gráficas, en la que K. Thompson y D. Ritchie reescribieron el código del juego. El desarrollo del juego en este caso se convirtió en algo más ambicioso de lo que esperaban, ya que descartaron todo el software del PDP 7 y crearon un paquete de punto flotante, otro para los gráficos de la pantalla, y un sistema de *debug*, todo ello, escrito en lenguaje ensamblador a través de un compilador cruzado en GECOS del GE 635 que producía cintas de papel posteriormente pasadas al PDP 7.

El éxito obtenido al implementar *Space Travel* para el PDP 7 les sirvió como reflexión sobre la tecnología necesaria para poder preparar programas en dicha computadora. Esto llevó a Thompson a pensar en el desarrollo de un sistema de ficheros básico, que aportara también las otras características para tener un sistema operativo funcional, como el concepto de procesos, utilidades para copiar, leer, escribir y borrar ficheros, y un interfaz básico de usuario, o *shell*. Todos estos conceptos fueron de nuevo implementados siguiendo la misma técnica que para el juego *Space Travel.* Utilizaron un compilador cruzado en GECOS para ensamblador del PDP 7, y pasaron el código a éste mediante cinta de papel. Pero, una vez acabado el sistema operativo, éste era autosuficiente como para soportar desarrollar sobre él. Finalmente, como prueba, se revisó "Space Travel" para que funcionara en el sistema operativo creado.

A este sistema le llamaron UNICS (Información Uniplexada y Sistema de Computación) ya que podía soportar dos usuarios a la vez y como juego de palabras con respecto a MULTICS. El nombre es atribuido a Brian Kernighan. En 1970 el nombre

fue cambiado a UNIX, aunque no queda del todo claro cómo se decidió hacer este cambio de nombre. Las características principales de este primer sistema eran las de no ser multiproceso, sólo un proceso podía estar en ejecución a la vez, el sistema de ficheros carecía de *pathnames*, es decir, caminos para localizar ficheros, por lo que todos los movimientos entre directorios debían hacerse a través de enlaces, y existían las redirecciones de entrada y salida > y <, pero no así el "pipe" |. Tampoco existían las actuales llamadas a sistema *fork* y *exec*, utilizadas para que un proceso genere a otro a través de una genealogía, por lo que la shell utilizaba un bucle de ejecución de comandos a base de saltos a punteros con el comienzo del comando y vuelta al código de la shell.

Fue en mayo de este mismo año, 1970, cuando, en uno de los últimos esfuerzos por encontrar nuevas computadoras para proyectos concretos, consiguieron que Bell Labs comprara un PDP 11 de DEC con el propósito de migrar el Unix de PDP 7 y desarrollar un sistema de edición y formateo de documentos, lo que conocemos hoy como *procesador de textos*. El sistema fue reescrito desde cero para el hardware del PDP 11 y transferido a través de terminales teletipo modelo 33. En esta versión sí se aportaron las llamadas a sistema *exec* y *fork*, los *pathnames* o nombres completos de directorios y ficheros. El equipo PDP 11 tenía 24Kb de memoria (16 para kernel y 8 para procesos de usuario) y disco de 512Kb con bloques de 1Kb. Los ficheros estaban limitados a 64Kb. Este pequeño tamaño para el sistema operativo fue quizá una de sus mejores bazas para alcanzar el éxito.

En esta versión de Unix se hizo la migración del código de un editor de textos ya existente, *roff* y convencieron al departamento de Patentes de Bell para usarlo como editor, ya que estaban evaluando otros sistemas comerciales para este propósito. Los alicientes que presentaban eran los de ser un *proyecto propio* de la empresa, soportar terminales teletipo modelo 37 que permitían caracteres matemáticos necesitados para dicho trabajo, y la posibilidad en *roff* de numerar las páginas y las líneas, características que los otros sistemas no soportaban. También fue un éxito esta prueba, y en 1972 ya había 10 computadoras con Unix. El editor *roff* evolucionó a *troff* que se sigue usando hoy en día.

Llegados a este punto, se planteó también la posibilidad de tener un lenguaje de alto nivel en el que programar. Barajaron la posibilidad de Fortran, pero presentaba serios problemas de portabilidad. Thompson desarrolló el lenguaje B [9], inspirado en un lenguaje ya existente, el BCPL (Basic Combined Programming Language) [10], pero más simple que éste. Este lenguaje derivó en 1971 en el conocido hasta hoy como C [11], escrito por Ritchie, con el que se volvió a codificar el sistema operativo en 1973, añadiéndose también el multiproceso, y los *pipes* (tuberías), sugeridos por Thompson. Es en este punto donde podemos decir que aquel Unix ya era similar a los sistemas Unix de la actualidad.

En 1974 apareció la primera publicación específica sobre el sistema Unix, ya escrito en lenguaje C [12]. Era la cuarta versión que se creaba. El código fue distribuido con licencias entre las universidades, ganando una gran popularidad en el mundo académico por varias razones:

– Era pequeño: como hemos dicho, las primeras versiones funcionaban con un disco de 512Kb, y usaban 24Kb de memoria, 16Kb para el sistema y 8Kb para programas de usuario.

– Era flexible: El código fuente estaba disponible, y escrito en lenguaje de alto nivel C, lo que facilitaba la portabilidad [13].

– Era barato: Por un precio muy bajo, las universidades tenían un sistema operativo con prestaciones que hasta entonces sólo estaban disponibles para equipos mucho más caros.

Por supuesto que también había inconvenientes: no había soporte, no había documentación y, como sigue sucediendo, no estaba exento de errores. Sin embargo, su popularidad en el mundo académico fue el mejor caldo de cultivo que pudo tener. Entre 1976 y 1977, Thompson se tomó unas vacaciones de Bell Labs para acudir a la universidad de Berkeley como profesor. Esto fue uno de los mejores empujones que tuvo dicha universidad para ser la que más desarrolló sobre Unix, hasta el punto de

sacar su propia distribución, la que se conocería como BSD (Berkeley Software Distribution). En 1977 salieron la quinta y sexta versiones y, para 1979 en que salió la séptima versión, ya había más de 600 máquinas con UNIX entre Bell labs y las universidades.

En 1982, AT&T sacó la primera versión comercial UNIX System III que fue difundido entre universidades y laboratorios de investigación, de forma que se desarrollaron múltiples herramientas. Este disparo en el desarrollo de aplicaciones influyó en una pérdida de compatibilidad. Para evitar este problema, AT&T sacó en 1983 el UNIX System V versión 1 (la versión IV fue interna y nunca salió al mercado). Esta fue la primera versión que aportaba mayor compatibilidad con todo lo existente hasta el momento. A partir de ahí salieron más versiones, hasta que a finales de los 80 salió UNIX System V release 4 (Unix SVR4), que aportaba herramientas de las versiones más populares hasta el momento: UNIX, BSD, XENIX, SUNOS y HP-UX. Hay que resaltar también que ha habido desarrollos que han aportado grandes avances a UNIX como el entorno gráfico X-Window a partir de 1984, o los protocolos TCP/IP para UNIX a partir de 1980.

A lo largo de la década de los 90 empezó la etapa que podríamos llamar de los UNIX propietarios. Fue en esta época cuando principalmente se empezaron a introducir los entornos operativos UNIX comerciales que predominan hoy en día, aunque realmente casi todos fueron desarrollados ya en los 80. En la tabla 2.1 podemos ver de forma muy resumida los principales UNIX para entorno servidor junto con su año de lanzamiento y la empresa que los comercializa.

Sistema Operativo	Año de lanzamiento	Empresa
AIX	1986	IBM
HPUX	1982	Hewlett Packard
Solaris (SUNOS)	1990 (Proviene de SUNOS en 1982)	Sun Microsystems
Tru64 (Digital Unix)	1999 (Proviene de Digital Unix en 1995)	Digital → Compaq → Hp

Figura 2.1: Principales entornos operativos UNIX comerciales.

Hoy en día estos entornos compiten principalmente por el mercado de servidores basados en UNIX y las empresas suelen vender soluciones completas que integran hardware y software con el sistema operativo desarrollado por ellas. La batalla por la cuota de mercado en este campo siempre ha sido muy dura.

A partir de 1995 se incorporaron otros jugadores en esta batalla. Los Linux comerciales, Red-Hat, Suse y Caldera, comenzaron a competir inicialmente en los servidores con procesadores Intel, aunque posteriormente también se han migrado a otro tipo de procesadores como los actuales Itanium (IA64). En realidad Linux apareció en 1991 como una idea concebida por Linus Torvalds, un estudiante universitario finlandés que usando como base MINIX, un entorno UNIX libre usado para experimentar en las universidades, y comunicando su proyecto a través de Internet, tuvo una acogida muy buena por parte de la comunidad de desarrolladores. El éxito de Linux fue precisamente éste, es decir, en lugar de trabajar un equipo organizado sobre él, todo se planteó como una comunidad de gente conectada a través de Internet que iban aportando ideas a este entorno operativo. El éxito de cada aplicación desarrollada para él dependía de una especie de sistema de *selección natural darwiniana* basado en la publicación casi continuada de nuevas versiones que la gente usaba y decidía si les convencía o no.

Hoy en día, podemos decir que los servidores basados en Linux pueden competir en igualdad de condiciones con el resto de los entornos UNIX, aunque los más puristas sigan insistiendo en que Linux nunca será UNIX.

2.2. Shell. Una Evolución

Una vez establecido el marco evolutivo en el que se integran las shells, podemos empezar a considerar la evolución misma de éstas. Como definición inicial, podemos decir que el concepto de shell se basa en la idea principal de facilitar al usuario la tarea

de lanzar comandos que son interpretados para que pueda ejecutarlos el sistema operativo Unix.

Existen dos modos posibles de trabajar con una shell. La primera forma es mostrando un cursor o *prompt* que permite al usuario escribir directamente los comandos, de tal manera que sean interpretados y ejecutados inmediatamente por el sistema operativo. Se implementa un bucle para poder ejecutar los comandos de forma inmediata según los siguientes pasos.

```
Cursor -> Introducir comando -> Interpretar comando
       -> Ejecutar comando -> Salida de comando -> Cursor ->…
```

La segunda forma es lanzando los comandos a través de un *script*. Básicamente, un script es un fichero de texto cuyo contenido son comandos del sistema operativo y estructuras de control similares a las usadas en lenguajes con programación estructurada como Pascal o C. Asignándole permisos de ejecución a este fichero, podemos lanzarlo como un comando más, ejecutando estos comandos en forma secuencial, o con bifurcaciones condicionales y bucles. Estas características, junto con otras más que posee la Programación Shell, le dan una potencia y utilidad muy superiores al modo de trabajo puramente interactivo. Todo esto será objeto de estudio en capítulos posteriores.

Teniendo en cuenta que una shell es el punto de acceso principal al sistema operativo UNIX, cualquier mejora en las características de éste será vital para que adquiera popularidad entre los usuarios y, por lo tanto, la clave de su éxito. Es por lo que hoy en día se tiende a conseguir, por ejemplo, una mayor rapidez para que el usuario acceda a los comandos (abreviaciones de teclado, histórico de comandos, etc.), facilitar más información (mejoras en el *prompt*), aportar compatibilidad hacia atrás con otras shells anteriores y también populares o, en el caso de la programación, aportar estructuras de control y manejo de variables cada vez más potentes, u optimizar el código para que la ejecución del script sea más rápida que en otras shells. Vamos a ver a continuación las shells que han sido más populares a lo largo de la historia de Unix, y cómo han ido evolucionando. No entraremos en este apartado en detalles técnicos, porque se cubrirá ese aspecto en los siguientes capítulos, pero sí citaremos las características que se han hecho más conocidas entre los usuarios.

2.2.1 Shell Bourne. El Gran 'Daddy'

La primera shell, que podemos considerar como tal, es la Shell Bourne, conocida como *sh*. Fue desarrollada por Steven Bourne hacia 1974 y apareció integrada en la versión 7 de UNIX en 1978. Previamente ya existían shells, como la de Thompson, que permitían ejecutar comandos, pero no tenían control de flujo ni variables, por lo que no existía aún el concepto de Programación Shell. La Shell Bourne ya aportaba las principales características tanto en modo interactivo, como en programación que se consideran hoy en día esenciales. De hecho, excepto la C Shell, podemos asegurar que el resto de las shells más populares, Ksh, POSIX y Bash, son extensiones y mejoras de la Shell Bourne. De la documentación que Bourne escribió para esta shell en la versión 7 de UNIX [14] podemos extraer las características que marcaron tanto a esta como a todas las sucesoras.

– Ejecución interactiva de comandos.
– Comandos en *background*.
– Redirección de entradas y salidas de comandos.
– *Pipelines* (comunicación entre comandos) y filtros.
– Generación de nombres de ficheros (caracteres comodines).
– Entrecomillado.
– Personalización de *prompt* (de forma estática como se explicará más adelante).
– Ficheros de perfil para personalización del entorno.
– Variables de Shell (locales y de entorno).
– Substitución de variables.
– Substitución de comandos.
– Debug de la shell.
– Agrupación de comandos.
– Condicionales.
– Control de flujo (*if-then-else, while-do, until-do, case*).
– Funciones (no estaban en UNIX V 7, la primera versión en la que apareció Bourne)
– Traps y manejo de errores.

Como podemos observar de la lista de características, las otras shell basadas en Bourne (Korn, POSIX y Bash) básicamente aportan una serie de mejoras que pueden suponer un porcentaje muy bajo comparativamente con lo que aportaba Bourne. Ya estaban definidas las principales líneas de la sintaxis de la Programación Shell, la redirección y los *pipes*, y la definición del entorno de usuario. Pero presentaba un gran problema. Casi todas las características aportadas iban orientadas a optimizar la ejecución de comandos, pero no presentaba apenas facilidades de uso para el usuario interactivo (tan sólo el uso de las funciones, que se implementó en versiones posteriores) [15]. Por esta razón, en cuanto aparecieron shells mejoradas, esta shell calló en desuso para el uso en modo interactivo.

Aún así, existen hoy en día versiones Unix, como Solaris que usan esta shell de forma predeterminada. Su éxito actual se basa en la simplicidad, ya que ocupa poca memoria y para scripts es óptimo en la rapidez de ejecución, pero sigue siendo la menos escogida a la hora de trabajar de forma interactiva. De hecho, otras distribuciones como HPUX, usan el ejecutable `/usr/bin/sh`, que habitualmente es una Shell Bourne, como una Shell POSIX, renombrando la Shell Bourne como `/usr/old/bin/sh`, o en el caso de AIX e IRIX, la renombran como `bsh`.

Como curiosidad, podemos también citar que la shell `/bin/sh` que aparece en las distribuciones BSD no es la Shell Bourne original, sino `ash` que fue escrita por Kenneth Almquist en 1989 y mejorada bastante desde entonces. En BSD la han renombrado como `/bin/sh` y se considera una ampliación de la Shell Bourne. Otra característica curiosa es que el lenguaje de programación es muy similar a un lenguaje existente hacia los 60 llamado ALGOL (*Algorithmic Language*).

2.2.2 C Shell. El Valor de ser Diferente

La primera alternativa a Bourne fue C Shell (`csh`), desarrollada por Bill Joy en Berkeley en el año 1978. Bill Joy fue también quien desarrolló el editor *vi*, además de ser uno de los principales impulsores de la distribución UNIX BSD de la universidad de Berkeley, y cofundador posteriormente de Sun Microsystems. Originalmente, C Shell se

integró en UNIX BSD. Tenía una serie de mejoras notables en modo interactivo, aunque la más destacable era la aparición de un histórico de comandos desde donde se podían recuperar los comandos ya ejecutados y para ser reeditados [16,17].

En el ámbito de la programación, se pretendió asimilarla a la sintaxis del lenguaje C, lo que por un lado era una ventaja para todos los programadores de este lenguaje, pero un inconveniente para la gente que ya trabajaba con la Shell Bourne, que por otra parte, usaba una sintaxis muy sencilla. Esto obligaba a conocer dos formas de Programación Shell totalmente diferentes y, además, incompatibles entre ellas. Quizá ha sido este detalle el que ha hecho que a pesar de que C Shell tiene un código muy optimizado y sintácticamente es muy avanzada, no haya sido una de las shells más populares entre todas las principales, ya que cuando C Shell apareció, ya había muchos scripts hechos en sintaxis Bourne, y la mayoría de los usuarios no estaban muy dispuestos a tener que reescribirlos.

En resumen, podemos destacar que las características nuevas aportadas con respecto a la Shell Bourne eran:

- Histórico de comandos.
- Sintaxis similar al lenguaje C.
- Alias de comandos (renombrar un comando con otro nombre más familiar).
- Control de *jobs* (para poder controlar múltiples procesos en una sola shell).

Una alternativa actual a los usuarios que prefieren `csh` es `tcsh` [18]. Esta es una shell mejorada, que ofrece compatibilidad completa con C Shell, y se centra en ofrecer mejoras en modo interactivo similares a las que ofrece la Shell Bash, como histórico de comandos mejorado, corrección sintáctica y completado de palabras.

2.2.3 Shell Korn. El Retorno a las Tradiciones

Como ya hemos visto, C Shell aportaba una serie de novedades muy suculentas para el usuario que trabajaba de forma interactiva, pero obligaba a reescribir el código de los

scripts ya hechos en la Shell Bourne. Con el fin de evitar este problema, David G. Korn desarrolló en Bell Labs el código de la Shell Korn (`ksh`) en 1983. La idea era aportar las mejoras que ya tenía C Shell, pero integrándolas en un lenguaje de programación y una sintaxis en modo interactivo similar al de la Shell Bourne, y además compatible hacia atrás con esta. La Shell Korn puede ejecutar perfectamente un script realizado para la Shell Bourne, ya que su lenguaje de programación y sintaxis son un superconjunto de esta.

Existen dos versiones lanzadas oficialmente, Korn Shell 88 que fue la oficialmente implantada en UNIX System V release 4 y Korn Shell 93 [19], que principalmente es una reescritura de la versión 88 centrándose en aportar mejoras de cara a la programación. La Shell Korn además cumple la norma estándar IEEE POSIX 1003.2 del PASC [20] (*Portable Application Standards Committee*), comúnmente conocido como *POSIX.2*.

Como mejoras, se asegura que a pesar de tener más código y por lo tanto ocupar más memoria [21], es capaz de ejecutar comandos de forma más rápida que Bourne y C Shell. De la documentación publicada tanto por el desarrollador como por otros referentes importantes [20,22] podemos apuntar que las principales características aportadas por la Shell Korn son:

- Compatibilidad hacia atrás con la Shell Bourne.
- Alias de comandos.
- Histórico de comandos.
- Sintaxis mejorada para las operaciones aritméticas.
- Edición de la línea de comandos compatible con vi y emacs.
- Control de *jobs*.
- Autocarga de funciones (pequeños procedimientos o agrupaciones de comandos).
- Completado de nombres de ficheros (abreviación de tecleo de comandos).
- Shell restringida (`rksh`).

Aunque básicamente algunas de estas características ya aparecían en C shell, se citan en este caso ya que se adaptaron para ser similares a la sintaxis de la Shell Bourne. En la versión 93, las mejoras vienen principalmente en la programación shell. Las más destacables son:

- En modo interactivo se aportan los atajos de teclado (*Keybinding*).
- Aritmética de punto flotante.
- Arrays asociativos (usan un *string* para ser generados).
- Arrays Indexados (usan una expresión aritmética para ser generados).
- Función `printf()` de ANSI C totalmente soportada.
- Funciones de búsqueda de subcadenas en variables mejoradas.

Aunque el código de Korn Shell 93 es público, pudiendo descargarse de Internet en http://www.research.att.com/sw/download su uso está sujeto a licencia por parte de AT&T y Lucent. No obstante, la descripción de la sintaxis sí es de dominio público y se puede reimplementar. Así han salido versiones de dominio público como `pdksh` (*Public Domain Korn Shell*) o se han integrado características como en el caso de Bash. En el caso de `pdksh`, soporta casi todo el estándar de la versión 88, y sólo algunas características de la versión 93, por lo que algunos scripts hechos para la Shell Korn pueden no ejecutarse en `pdksh`. Hoy en día, en muchas versiones UNIX se sigue aportando la versión 88, ya que venía con UNIX SVR4 que ha sido tomado como un referente y, además, la versión 93 necesita una licencia aparte.

También existen dos versiones con extensiones para entorno gráfico. La `dtksh` que viene junto con el entorno gráfico CDE (*Common Desktop Environment*, que es un estándar en Solaris, HPUX, AIX y TRU-64) y viene con un interfaz para *motif* (gestor de ventanas en Unix) que le permite ejecutar scripts en ventanas. También está `tksh` que es capaz de interactuar con la librería gráfica *Tk*, y trabajar con scripts *Tcl* (El lenguaje script de *Tk*) compartiendo espacio de nombres y comandos.

2.2.4 Shell POSIX. Adaptándose al Estándar

Hemos decidido añadir en este estudio la shell que viene incorporada en las distribuciones HP-UX de los servidores de la serie 9000 y recientemente en la serie Integrity de Hewlett Packard. Esta shell, denominada POSIX (`sh`), fue desarrollada por AT&T (Bell Labs), OSF (Open Group) y Hewlett Packard. Se utilizó como base la Shell Korn, por lo que, como se especifica en la documentación publicada para esta shell [23], las similitudes con `ksh` son muchísimas, hasta el punto de ser prácticamente compatibles en cuanto al lenguaje de programación y sintaxis. De hecho, esta shell cumple el mismo estándar que la Shell Korn, es decir, POSIX.2.

En la misma documentación citada anteriormente [23], también se justifica el hecho de que el estándar POSIX.2 requiere que cuando se lance una shell con el comando `sh`, ésta debe ser una Shell POSIX, por lo que en HPUX, como citamos anteriormente, `/usr/bin/sh` es por defecto POSIX, habiendo renombrado la Shell Bourne como `/usr/old/bin/sh`. Además, usando el comando `man sh`, la ayuda que es facilitada es la de la Shell POSIX. De hecho, en la versión 11i, la última de HP-UX, la Shell Bourne ha sido eliminada completamente. Esto contrasta con otras distribuciones de UNIX como Solaris, de Sun Microsystems, que sigue aún usando la Shell Bourne por defecto, a pesar de que aseguran cumplir los estándares POSIX.1 y POSIX.2.

No enumeraremos ahora las características aportadas por esta shell, ya que básicamente son las mismas que en el caso de la Shell Korn, aunque se hará una pequeña comparación entre ambas en el capítulo 3.

Como conclusión, podemos decir que la Shell POSIX surgió de la necesidad de establecer una shell estándar, con características muy aceptadas por los usuarios de las shells ya existentes, principalmente, de la Shell Korn.

2.2.5 Shell Bash. En Busca de la Libertad

Terminamos este repaso histórico con la Shell Bash. El nombre proviene de *Bourne Again Shell* (Otra Shell Bourne), y nació como uno de los proyectos GNU [24] de la

FSF (*Free Software Foundation*). La filosofía general de dicho organismo es desarrollar software completamente gratuito y de libre distribución, bajo licencia GNU, filosofía igualmente aplicada a la Shell Bash.

La Shell Bash está diseñada para cumplir el estándar POSIX.2 igual que la Shell POSIX y la Shell Korn, pero además presenta características especiales tomadas de C Shell. En cualquier caso, si se desea tener una shell que sólo sea POSIX, Bash proporciona un flag (`--Posix`) que permite ejecutar Bash ateniéndose exclusivamente a dicho estándar.

De la documentación publicada para Bash [25,26], podemos extraer algunas de las características aportadas:

– Edición de línea de comandos amigable, personalizable y con completado de nombres.
– Histórico de comandos con mejoras como selección de comandos guardados.
– Arrays indexados de tamaño ilimitado.
– Aritmética en bases desde 2 a 16.
– Entrecomillado estándar ANSI-C.
– Expansión de corchetes (similar a C Shell).
– Operaciones con subcadenas en variables.
– Opciones de comportamiento.
– Prompt mejorado con caracteres especiales.
– Ayuda integrada.
– Modo POSIX.
– Shell restringida.
– Temporización de comandos.
– Pila de directorios.
– Control de comandos internos.

Bash es una de las shells más usadas hoy en día, principalmente porque ha sido adoptada como shell predeterminada en las distribuciones de Linux. La característica de ser un proyecto GNU le da el atractivo de ser totalmente libre, por lo que también puede

ser usada en cualquier otra distribución UNIX, sólo es necesario bajar el código y compilarlo, o buscar el ejecutable ya compilado para nuestro sistema. Este hecho, junto con la gran cantidad de mejoras que se han ido aportando, la convierte, quizá, en una de las mejores implementaciones POSIX de las shells y, además, se pueden añadir características que aún no cumplen dicho estándar.

2.3. Conclusiones

En el presente capítulo hemos hecho un repaso histórico de las shells, buscando como punto de apoyo inicial los orígenes del sistema operativo Unix, y viendo tanto los aspectos históricos, como técnicos de cada una de las shells más destacables. No se ha entrado en detalles técnicos de las características de cada shell, ya que ese será el propósito de los siguientes capítulos. Podemos concluir este punto apreciando la importancia que han tenido siempre las shells en el entorno Unix, ya que es la herramienta principal para interactuar con el sistema operativo y, por lo tanto, los desarrolladores siempre han buscado facilitar cada vez más su uso a los usuarios finales.

CAPÍTULO 3. COMPARATIVA FUNCIONAL DE LAS SHELLS DE UNIX

Mientras que en el punto anterior nos centramos en hacer una comparación de las shells de Unix bajo una orientación histórica, en este caso, nos vamos a centrar en su funcionalidad. El objetivo final es establecer un marco comparativo que permita decidir cuál es la shell más adecuada para un administrador.

3.1. Criterios Técnicos

Debemos definir una serie de criterios que nos permitan cuantificar los aspectos más relevantes de las shells. Teniendo en cuenta lo que hemos ido exponiendo en el punto anterior, existen una serie de funcionalidades técnicas de una shell que podemos considerar relevantes. Vamos por lo tanto a exponerlas y a definir qué shells poseen dichas características. Se van a tener en cuenta aquellas en las que las principales shells son más diferentes unas de otras. Hay que aclarar además que, a todos los efectos, las funcionalidades expuestas, son idénticas en la Shell Korn y en la Shell POSIX, por lo que sólo haremos una pequeña referencia a ciertas diferencias relevantes. Una vez expuestas, definiremos algunos criterios más subjetivos, como facilidad de uso, portabilidad, etc.

3.1.1 Histórico de Comandos

Esta es una característica bastante deseada en una shell, cuando debe ser usada en modo interactivo. Básicamente, lo que va a permitir al usuario es recuperar comandos que ya hayan sido ejecutados anteriormente en la sesión actual o en las anteriores. Excepto la Shell Bourne, el resto poseen esta característica. Sin embargo, el funcionamiento es bastante diferente en casi todos los casos, por lo que merece la pena compararlos en detalle.

Shell Bourne [27]: Carece de esta característica.

C-shell [16]: Es necesario definir la variable `history` con un número que indica la cantidad de comandos que se almacenarán. Una vez definido, podemos recuperar comandos usando el carácter especial `!`. Algunas de las principales acciones que se pueden realizar son

`!!`	repite el último comando
`!n`	repite el comando número `n`
`!(cadena)`	repite el último comando que empiece por cadena
`history`	muestra la lista de comandos almacenados

Este último comando, `history`, no es un alias como sucede en Korn o Bash, sino que es una sentencia de la shell.

Al contrario que en la Shell Korn o en Bash, el histórico no se almacena en un fichero, sino que es guardado en memoria durante la sesión, lo que no permite recuperar los comandos ejecutados en sesiones anteriores.

Shell Korn [28]: Aunque no es necesario definirlas, ya que tienen valores por defecto, el histórico de comandos depende de la variable `HISTSIZE` (por defecto 128) para definir el número de comandos almacenados, y de la variable `HISTFILE` (por defecto `$HOME/.sh_history`) para indicar el fichero donde se almacenarán los comandos. Además es necesario activar el modo de edición `emacs` o `vi` de la shell para la línea de comandos, lo que se puede hacer asignando dicho valor a la variable `VISUAL`, o a la variable `EDITOR`, o bien con la sentencia `set`, de la forma

```
$ set -o vi
    ó
$ set -o emacs
```

En el caso de que tengamos definido el modo de edición como `vi`, recuperaremos comandos mediante la secuencia `ESCAPE+K`. En el modo `emacs` lo haremos con `CONTROL+P`.

La sentencia `fc` se puede usar para recuperar comandos anteriores y editarlos, aunque por simplicidad, también quedan definidos los alias `history` para recuperar una lista de histórico de comandos, y `r` para repetir uno de los últimos comandos ejecutados, que estás basados en dicha sentencia.

Al fichero `$HOME/.sh_history` se van añadiendo los comandos a medida que se van ejecutando, lo que no permite tanta flexibilidad como la que tiene Bash, que ahora comentaremos. Entre otras, una consecuencia desagradable de este hecho es que, a no ser que cambiemos la variable de valor, los comandos de diferentes sesiones simultáneas se mezclan unos con otros.

Shell Bash [29]: En Bash, las variables `HISTFILE` (por defecto `$HOME/.bash_history`) y `HISTSIZE` (por defecto 500) tienen el mismo significado que en Korn [28]. También está soportado el modo de llamada al histórico de la C Shell con el carácter `!`. Además, existen dos variables nuevas, `HISTCONTROL` que permite evitar que se añadan en el histórico de comandos, líneas comenzadas con espacio y comandos duplicados, y `HISTIGNORE` para indicar qué comandos no queremos que sean almacenados en el fichero histórico.

Al igual que en la Shell Korn [28], la variable `EDITOR`, o el comando `set -o`, definirá el modo de edición en que vamos a trabajar, aunque en Bash [29], por defecto está definido el modo `emacs`.

Un detalle bastante importante de Bash [29] es que, gracias a la librería *readline* que lleva incorporada, podemos acceder a los comandos del histórico simplemente usando las teclas de cursor del teclado extendido, lo que facilita mucho el trabajo. Esta librería, como iremos viendo, aporta muchísimas facilidades de este tipo de cara al usuario.

Además, el fichero `$HOME/.bash_history` no va creciendo a medida que vamos ejecutando comandos, sino que los comandos de la sesión actual se almacenan en memoria, lo que los hace más rápidos de recuperar, y se vuelcan al final de la sesión, para ser recuperados como parte de la sesión siguiente. Además, con esto se evita el problema que tenía Korn [28] de mezclar los comandos de diferentes sesiones que se abrieran simultáneamente.

3.1.2 Edición de la Línea de Comandos

Esta funcionalidad permite al usuario una vez recuperado un comando del histórico, o en el comando actual, poder usar funciones de edición como borrar caracteres, retroceder o avanzar el cursor, cortar, pegar, etc. En todas las shells se toma como base un editor para usar sus comandos, generalmente dicho editor es `vi` o `emacs`. No es factible decir que es mejor un editor u otro, eso depende del criterio del usuario, ya que cada uno prefiere aquel al que esté más acostumbrado.

Shell Bourne [27]: Carece de esta característica.

C Shell [16]: En el caso de C Shell, sólo disponemos de la sentencia `fc` para editar la línea de comandos. Esto obliga a ejecutar

```
% 'fc -e editor comando'
```

donde *`editor`* es el editor que vamos a usar, por defecto `/bin/ed` y *`comando`* es el último comando del histórico con ese nombre.

Shell Korn [28]: El modo de edición, como comentamos en la característica de histórico de comandos, depende de la variable `VISUAL`, o en su defecto, de `EDITOR`. La asignación debe hacerse de las dos formas

```
$ export EDITOR=vi          o          $ export EDITOR=emacs
```

Esto activa automáticamente la opción *vi* o *emacs* de la shell, que también se puede activar con

```
$ set -o vi          o          $ set -o emacs
```

Básicamente, una vez activado, podemos usar comandos estándar de `vi` o `emacs` cuando estamos escribiendo un comando, o recuperamos alguno del histórico.

Shell Bash [29]: Bash se inicia por defecto en modo `emacs`, sin necesidad de activarlo. Además, hace uso de la librería *readline* que aporta toda una serie de facilidades a la edición de la línea de comandos. Por ejemplo, podemos usar directamente las teclas *backspace*, y *supr* para borrar texto, e insertar texto sin necesidad de activar ningún modo especial. Esto lo hace muy atractivo para un usuario novel, que conozca el funcionamiento básico de cualquier editor como por ejemplo *notepad* de windows.

La librería *readline* también aporta una serie de personalizaciones del modo de trabajo de Bash [29], que pueden inicializarse mediante el fichero `$HOME/.inputrc`, o la sentencia `set`.

3.1.3 Seguimiento Invisible de Enlaces Simbólicos

Esta funcionalidad hace referencia a la propiedad que tienen algunas shells de tratar los enlaces simbólicos de directorios como directorios reales, algo que es muy deseable, en el caso de trabajar tanto en modo interactivo, como para un script shell que hace referencia a directorios.

C shell y Bourne [27,16] carecen de esta característica, que sí está presente en Korn [28], y Bash [29]. Básicamente, si tenemos un enlace simbólico, como por ejemplo:

```
/bin -> /usr/bin
```

En el caso de una Shell Bourne o una C shell [16], si hacemos la secuencia de comandos siguiente:

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

Es decir, al usar el enlace simbólico, para cambiar de directorio, trata directamente al directorio original, por lo que desconoce que, al volver hacia atrás, debe volver al directorio raíz, y vuelve a `/usr`. Esto puede dar problemas en determinadas condiciones para comandos o scripts que necesitan saber el directorio donde se encuentran. En cambio, en Korn [28] o Bash [29], la misma secuencia de comandos, daría como resultado:

```
$ pwd
 /
$ cd /bin
$ pwd
 /bin     (pero en realidad estaríamos en /usr/bin)
$ cd ..
$ pwd
 /
```

Es decir, hace ver al sistema que está en un directorio de verdad, que se llama `bin`, pero en realidad está usando `/usr/bin`, lo cual es bastante deseable en la mayoría de los casos, para que el uso de un directorio sea coherente, tanto si se usa directamente, como a través de enlaces.

3.1.4 Completado de Nombres de Ficheros, Comandos, Usuarios, Hosts y Variables

En general, en mayor o menor medida, casi todas las shells, excepto Bourne, tienen la capacidad de escribir por nosotros los nombres de ficheros, comandos, etc., lo que acelera muchísimo el proceso de introducir comandos de forma interactiva. Para poder

aprovechar esta característica, debemos conocer lo que puede hacer cada shell, y cómo lo hace, ya que no es igual en todos los casos. Vamos por lo tanto, a ver las diferencias.

Shell Bourne [27]: Carece de esta característica.

C shell [16]: Para poder usar esta característica, debemos activar la variable `filec` con el comando

```
% set filec
```

Con esto, ya podremos usar el completado de nombres de ficheros. Escribiendo en una línea de comandos, si escribimos parcialmente el nombre de un directorio o fichero que es reconocido por el sistema, al pulsar la tecla *ESC* se escribirá el nombre completo de este fichero o directorio.

Aunque algunas documentaciones [15] indican que existe también la posibilidad adicional de completar nombres de comandos, la única forma de hacerlo posible es escribiendo el path completo del comando, por ejemplo, `/usr/bin/passwd` lo que lo hace poco operativo para este caso. Para completar un nombre de usuario, debemos usar el carácter ~ antecediendo a dicho nombre. Por ejemplo, si el usuario *pedrolopez* existe en nuestro sistema, escribiendo

```
% ~ped
```

y pulsando la tecla *ESC*, obtendríamos

```
% ~pedrolopez
```

Esta notación a su vez, se utiliza para poder acceder al directorio personal del usuario, o referenciar a ficheros dentro de él. En cualquier caso, el carácter ~ no suele aparecer en los teclados estándar con una tecla propia, salvo excepciones, como los teclados de los equipos SUN, lo que lo convierte en una característica poco usada.

No se ha encontrado la posibilidad de completar nombres de hosts o de variables.

Shell Korn [28]: en este caso debemos tener activado el modo de edición de línea de comandos en `vi` o `emacs` como describimos anteriormente. Una vez activado, al escribir un nombre de fichero o directorio, usaremos la secuencia de teclas *ESCAPE*+\ para completarlo.

Nota: Según los faqs de la Shell Korn [30], es posible usar la tecla tabulador a partir de la versión *g* de dicha shell. No se ha podido probar esta característica, ya que la versión disponible en Solaris 9 que se ha probado, era anterior. No obstante, sí podemos indicar que `pdksh` [31] (Public Domain Korn Shell) puede usar la tecla tabulador activando la opción:

```
$ set -o vi-tabcomplete
```

También según estos mismos faqs, la secuencia para recuperar comandos no es *ESCAPE*+\, sino *ESCAPE*+*ESCAPE*, sin embargo, esta segunda secuencia de escape sólo se ha comprobado como cierta en la Shell POSIX de HP-UX, y no así en la Shell Korn de Solaris, como ya hemos dicho.

En la versión de la Shell Korn [28] analizada, la de Solaris 9, no se ha podido constatar que se pueda hacer completado de comandos, pero sí es posible hacerlo en `pdksh`. El completado de comandos en este caso, se realiza de igual forma. Escribiríamos el principio del comando, sin necesidad de poner el *path* completo de este, y usando la secuencia de escape, aparecería el comando completo.

En esta shell no es posible hacer completado de nombres de usuarios, ni de hosts, ni de variables.

Shell Bash[29]: Bash utiliza, como ya explicamos, la librería *readline*, que por defecto lleva activo el completado de nombres de ficheros, pero no sólo eso, sino que además podremos completar nombres de comando, variables, hosts y usuarios, de una forma bastante sencilla.

Para los nombres de ficheros y directorios, basta con escribir en una línea de comando los primeros caracteres y pulsar la tecla tabulador. Esto completará el nombre o si hay varios ficheros que coincidan, escribirá la cadena común, esperando que escribamos el resto del nombre. En caso de duda, un segundo toque de tabulador nos muestra la lista de posibilidades.

Para completar un comando, la secuencia es igual, escribimos los primeros caracteres del comando y con la tecla tabulador completamos el nombre, en caso de ambigüedad, un segundo toque de tabulador presenta la lista de posibilidades.

El completado de nombres de usuarios se hace comenzando con el carácter ~ y los primeros caracteres del nombre. De nuevo, esto completaría el nombre completo del usuario, exactamente igual a como se ha descrito para C shell [16].

Para los nombres de hosts, sería igual que en el caso anterior, pero empezando con el carácter `@`, por ejemplo

```
$ @bor [TAB]
```

se completaría como

```
$ @borabora
```

en el caso de que este fuera el nombre del host en el que trabajamos.

Para completar el nombre de variables, haríamos lo mismo, pero empezando por el carácter `$`.

3.1.5 Facilidades para el uso del Prompt de Usuario

Aunque esta parezca una característica un poco secundaria, es bastante útil a la hora de trabajar en modo interactivo, para aclarar al usuario en qué situación se encuentra. El prompt puede informarnos por ejemplo del directorio en el que estamos, el host al que estamos conectados, el usuario con que hemos hecho login, la fecha actual, etc.

En principio, todas las shells tienen un prompt que queda definido mediante una variable, y es mostrado cada vez que ejecutamos un comando, mientras se espera a introducir el siguiente. Aunque esto sea común a todas ellas, algunas como Bash [29] o Korn [28] tienen ciertas facilidades para poder presentar un prompt mucho más elaborado.

Shell Bourne [27]: La variable `PS1` define en el prompt, el valor que le asignemos, el cual será mostrado. A la hora de asignarla, podemos usar el valor de otras variables, o la sustitución de comandos, para que adopte cierta apariencia. La Tabla 3.1 presenta algunos ejemplos, asumiendo que el nombre de usuario es `pedrolopez`, y el host se llama `borabora`.

Asignación	Prompt presentado
`$ PS1="$LOGNAME $"`	`pedrolopez $`
`$ PS1="`hostname` $"`	`borabora $`
`$ PS1="$LOGNAME@`hostname` $"`	`pedrolopez@borabora $`
`$ PS1="`date +%D` #"`	`10/11/2004 #`

Tabla 3.1: Ejemplos de asignación de la variable `PS1`

Como aclaración hay que indicar que la notación de comillas inversas `` `comando` `` es lo que denominamos una sustitución de comando, donde se ejecuta el comando correspondiente, y éste es pasado a la variable en forma de cadena, para que pueda usarlo. En este caso, el comando `hostname` facilita el nombre del host actual, y `date` facilita la fecha con un formato determinado.

De forma predeterminada, el superusuario `root` usa como prompt `#`, y el resto de los usuarios usan `$`, aunque esto, como vemos, se puede cambiar.

Un problema que presenta `PS1` en Bourne [27], es que su interpretación es estática, es decir, si asignamos una variable como `PWD` que contiene el directorio actual de trabajo, ésta es interpretada sólo en el momento de la asignación, con lo que no cambia de valor a medida que cambiamos de directorio. Esto no sucede en Korn [28], ni en Bash [29], como indicaremos.

C shell [16]: La variable que se usa es `prompt`, que deberemos asignar de la siguiente forma

```
% set prompt=valor
```

Por ejemplo:

```
% set prompt="$user %"
```

mostraría

```
pedrolopez %
```

Al igual que en Bourne, C shell [16] interpreta esta variable de forma estática, por lo que no se puede usar para variables que cambian dinámicamente como `cwd` que es la que muestra en este caso el directorio actual de trabajo.

Shell Korn [28]: Al ser compatible con Bourne, también es la variable `PS1` la que usamos. Pero en este caso, su interpretación es dinámica, por lo que podemos hacer un prompt como el del siguiente ejemplo:

```
$ PS1="$LOGNAME@$(hostname):"'$PWD $'
```

que mostraría el siguiente prompt, si estuviéramos en el directorio `/usr/share`

```
pedrolopez@borabora:/usr/share
```

Hay que comentar un par de detalles de este ejemplo. En primer lugar, la notación `$(`*comando*`)` es una sustitución de comando, idéntica a la notación `` `comando` `` de Bourne [27], esto es una extensión nueva en Korn [28] y Bash [29].

En segundo lugar, la asignación de la variable `$PWD` debe hacerse entre comillas simples, para que sea pasada como literal a la variable, que se encargará de interpretarla de forma dinámica, en cada ejecución de comando, de esa forma, cuando cambiamos de directorio, `PS1` también cambia. Por esa razón se separa el entrecomillado de `'$PWD'` del resto de la asignación de `PS1`. Si pasáramos `"$PWD"` con doble entrecomillado, se interpretaría en el momento de asignarse a `PS1`, y nunca cambiaría.

Shell Bash [29]: También la variable `PS1` es la que usamos en este caso, y el comportamiento es idéntico a Korn [28], por lo que todo lo visto antes es válido aquí.

Además, Bash [29] aporta una serie de códigos especiales para `PS1` que nos permite simplificar la asignación de estos valores. Por ejemplo, compárese el ejemplo visto anteriormente en la Shell Korn [28], con una asignación en Bash, que daría el mismo resultado:

Korn: `PS1="$LOGNAME@$(hostname):"'$PWD $'`

Bash: `PS1="\u@\h:\w $"`

En ambos casos, el resultado es el mismo, pero en Bash es más simple la asignación, ya que no hay que recurrir a otras variables o comandos y no tenemos que tener en cuenta el doble o simple entrecomillado.

Esto es un extracto con algunos de los códigos que podemos usar en Bash [29]:

`\d`	Fecha en formato 'Semana Mes Día'
`\h`	Nombre del host
`\s`	Nombre de la shell
`\t`	Hora actual
`\u`	Nombre del usuario
`\v`	Versión de Bash
`\w`	Directorio actual

Además de `PS1`, existen otras variables prompt con diferentes propósitos que son:

– `PS2` Es el prompt secundario, y se activa cuando una línea de comando no está completa y pulsamos intro. Existe en Bourne, Korn y Bash [27, 28, 29].

– `PS3` Prompt presentado cuando usamos una sentencia de control `select` para pedir una opción al usuario. Sólo existe en Korn [28] y Bash [29].

– `PS4` Se usa cuando activamos la opción trace, y se antepone a cada línea de la traza de ejecución del script. Sólo existe en Korn [28] y Bash [29].

NOTA: Todas las diferencias que hemos comentado hasta el momento están relacionadas con el modo de trabajo interactivo, es decir, afectan al usuario cuando usa la shell introduciendo comandos directamente. A partir de este punto, comentaremos diferencias en características que tienen más que ver con la programación shell.

3.1.6 Evaluación de Expresiones Aritméticas y Lógicas

Una característica muy recomendable a la hora de programar una shell, es que tenga una forma sencilla de escribir expresiones aritméticas y lógicas, para poder hacer sumas, restas, divisiones, multiplicaciones, comparaciones y cualquier otra operación necesaria. En este sentido, en todas las shell podemos hacerlo, pero cada una de ellas tiene su propia notación. Vamos pues, a compararlas.

Shell Bourne [27]: En esta shell no tenemos integrada en realidad la posibilidad de evaluar expresiones aritméticas, pero aún así, podemos hacer uso del comando externo `expr` que nos permite evaluar una expresión y asignar su resultado a una variable. Así, para sumar dos números, tendríamos que hacer algo como:

```
$ NUM1="4"
$ NUM2="5"
$ NUM3=`expr $NUM1 "+" $NUM2`
$ echo $NUM3
 9
```

donde vemos que `expr` trabaja con argumentos que sean cadenas y para asignarlo a otra variable hay que hacerlo con una sustitución de comando, con la notación `` `comando` ``, lo que hace que sea bastante difícil de entender lo que se pretende conseguir con dicha operación, a pesar de ser una simple suma.

En cualquier caso, es necesario insistir en que esto no es algo integrado en la Shell Bourne [27], sino un comando externo. Podemos hacer con `expr` tanto expresiones aritméticas como lógicas.

C shell [16,32]: Las operaciones se pueden hacer en C shell con la sentencia `@`, que toma el formato:

```
@ nombre = expresion
```

En cierto modo, excepto por el hecho de que el comando `@` asigna a la variable *`nombre`* el resultado de una expresión aritmética o lógica, su funcionamiento es igual a la sentencia `set`, salvo que éste sólo asigna un valor constante a una variable. Por ejemplo, es igualmente válido hacer estos dos comandos, y el resultado sería el mismo.

```
% set valor = 1
% @ valor = 1
```

Sin embargo, para comprobar cómo son diferentes, podríamos hacer la siguiente prueba:

```
% @ x = (100 + 200)
% echo Resultado $x
Resultado 300
% set y = (100 + 200)
% echo Resultado $y
Resultado 100 + 200
```

Como vemos, en el caso de hacer `set` lo que obtenemos es una asignación de cadena, y no una operación aritmética. En el caso de las expresiones lógicas, el valor será `0` para verdadero, y `1` para falso:

```
% uno = 1
% dos = 2
% @ si = ($uno >= $dos)
% echo $si
 0
% @ no = ($dos >= $uno)
% echo $no
 1
```

Los operadores aritméticos y lógicos que se pueden usar son los mismos que para el resto de las shell. Además, podemos usar los operadores de lenguaje C `+=, -=, ++ y --`. Por ejemplo:

```
% @ uno = 1
% @ uno ++
% echo $uno
 2
% @ dos = 2
% @ uno += $dos
% echo $uno
 3
```

En resumen, en C shell se pretende usar una sintaxis similar a la del lenguaje C, pero aun así, tenemos que preceder estas operaciones con el comando `@`, por lo que la apariencia final no es la misma que en dicho lenguaje. Los operadores que tenemos disponibles en C shell [16] son:

Operadores lógicos

`== !=`	igualdad y desigualdad de literales
`=~ !~`	igualdad y desigualdad de expresiones regulares
`<= >= < >`	comparaciones
`&& ||`	*and* y *or*
`!`	*not*

Operadores aritméticos

`+ -`	suma y resta
`* / %`	producto, división y resto
`& ^`	and y or bit a bit
`<< >>`	desplazamiento a izquierda y derecha de bits

Shell Korn [28]: En esta shell, usamos la sentencia `let` para poner expresiones aritméticas y `test` para expresiones lógicas. Ambas sentencias tienen equivalentes en una notación un poco más clarificadora. Podemos usar dos formas en cada uno de ellos.

```
let expresion ó (( expresión ))
              y
test expresión ó [ expresión ]
```

La principal ventaja de usar la segunda notación en ambos casos es que en el caso de usar la palabra clave `let` o `test` debemos poner la expresión que continúa sin espacios, o usar comillas, con la segunda notación esto no es necesario. Por ejemplo, son válidas las siguientes ejecuciones, y todas dan el mismo resultado:

```
$ X=1
$ Y=2
$ let Z=X+Y
$ let ' Z  =   X  +  Y '
$ (( Z  = X + Y ))
$ echo $Z
 3
```

Las tres versiones darían el mismo resultado. Sin embargo

```
$ let Z  =  X  +  Y
 ksh: =: unexpected '='
```

Daría el error indicado, ya que al no usar comillas, se pasan argumentos separados al intérprete, en lugar de una sola cadena, que es lo que necesita. Todo lo comentado es igualmente válido para la sentencia `test`.

La sentencia `test` se utiliza para evaluar expresiones booleanas, y generalmente va incluido en una sentencia `if` o `while`. Existe infinidad de operadores que se pueden usar, ya que además, podemos comparar propiedades de ficheros. Exponemos a continuación algunos de estos operadores. Puede consultarse el resto en las páginas del manual de la Shell Korn [28].

Operaciones con ficheros

`-r` el fichero se puede leer

`-w` el fichero se puede escribir

`-x` el fichero se puede ejecutar

`-d` es un directorio

`-e` el fichero existe

Operaciones con cadenas

`cadena1` `=` *`cadena2`* las cadenas son iguales

`cadena1` `!=` *`cadena2`* las cadenas son diferentes

Operaciones con números

`número1` `-eq` *`número2`* los números son iguales

`número1` `-ne` *`número2`* los números no son iguales

`número1` `-lt` *`número2`* número1 menor que número2

`número1` `-le` *`número2`* número1 menor o igual que número2

`número1` `-gt` *`número2`* número1 mayor que número2

`número1` `-ge` *`número2`* número1 mayor o igual que número2

Es importante reseñar que también existe la notación `[[` *`expresion`* `]]` La principal característica de esta notación es que permite la expansión de nombres mediante caracteres comodín con los operadores == y !=, o el uso de variables sin asignar. Veamos un ejemplo de uso de comodines.

NOTA: La variable `$?` contiene el estado de terminación del último comando (1 (o algo diferente de 0) = error o falso, 0 = correcto o verdadero).

```
$ X=prueba
$ [ $X == pru* ]
$ echo $?
 1
$ [[ $X == pru* ]]
$ echo $?
 0
```

En el segundo caso, usando la notación `[[ ]]` ha dado el valor verdadero, ya que en el primer caso, la notación `[ ]` ha analizado el carácter `*` como un valor literal, y no como un comodín para indicar cualquier carácter. Ahora veamos un ejemplo de uso de variables sin asignar. Estas variables son muy problemáticas en la notación `[ ]`, ya que provocan un error.

```
$ X=prueba
$ unset Y
$ [ $X == $Y ]
 ksh: [: ==: missing second argument
$ [[ $X == $Y ]]
$ echo $?
 1
```

En el primer caso ha provocado un error de salida, ya que al convertir las variables antes de evaluar, lo que se evalúa finalmente es la expresión `[ prueba == ]` a la que le falta claramente el segundo argumento. Sin embargo, obsérvese cómo con la segunda notación, no es necesario. Simplemente evalúa a falso, que por otra parte, es la respuesta correcta. Un detalle importante es que no podemos entrecomillar los caracteres comodines, ya que los analizaría como literales. Por ejemplo:

```
$ X=prueba
$ [[ $X == "pru*" ]]
$ echo $?
 1
```

da como resultado falso, ya que analiza el literal `pru*`, que no coincide con la variable. La sentencia `let` es usada para cálculos de expresiones aritméticas. Indicamos a continuación algunos de los operandos que podemos usar:

variable++ *variable*--	incremento y decremento de *variable*
+ -	suma y resta
* / %	producto, división y resto
> >= < <=	comparaciones
== !=	igualdad y desigualdad
= -= +=	asignación
<< >>	desplazamiento a izquierda y derecha de bits

`&` `^` and y or de bits

Como vemos, se pueden realizar también operaciones de comparación, pero los operandos deben ser siempre números, jamás cadenas. Otra característica es que las variables no deben ser precedidas del carácter `$`, lo cual facilita mucho la legibilidad de las operaciones. Obsérvese esto en los siguientes ejemplos con la notación `(( ))`:

```
$ X=5
$ Y=3
$ (( Z = X + Y ))       # Suma de las variables X e Y
$ echo $Z
 8
$ V=2
$ (( Z -= V ))          # Z es el resultado de su valor menos el de V
$ echo $Z
 6
$ (( Z++ ))             # Z es incrementado una unidad
$ echo $Z
 7
$ (( Z = X & Y ))       # Z es la operación 'and' bit a bit entre X e Y
$ echo $Z
 1
$ (( Z = 1 << Z ))      # Z es desplazado 1 bit a la derecha
$ echo $Z
 2
```

En cuanto a las operaciones de comparación numérica podemos observar que se pueden hacer tanto con `test` como con `let`, pero en el segundo caso siempre será más legible. Compárese en el siguiente ejemplo. Ambas operaciones son equivalentes, pero la segunda tiene mejor legibilidad.

```
$ X=1
$ Y=2
$ [[ $X -le $Y ]]
$ echo $?
 0
$ (( X <= Y ))
$ echo $?
 0
```

Shell Bash [28]: Todo lo que hemos comentado de la Shell Korn es válido en Bash. El modo de funcionamiento de las operaciones aritméticas y lógicas han sido heredadas del anterior.

3.1.7 Manejo de Variables

Hay ciertas características con respecto al tratamiento y uso de variables en las shells que las diferencian significativamente a unas de otras. Vamos a comparar cuatro de estas características que son el uso de arrays, el manejo de cadenas asignadas a variables, la declaración de tipos, y el uso de variables locales y de entorno. Sólo trataremos los aspectos diferenciadores que son los que nos interesan para nuestro fin.

3.1.7.1 Arrays

Excepto la Shell Bourne, todas las shells comparadas admiten el uso de arrays de una sola dimensión. En la Tabla 3.2 podemos ver las diferencias de funcionamiento entre las distintas shells con respecto al manejo de arrays. No se incluye en dicha tabla la Shell Bourne puesto que no posee dicha característica.

Acción	**C shell**	**Korn**	**Bash**
Declarar una variable array	`-.............`	`-...........`	`$declare -a array`
Numeración de índices	`1,2,3,4...`	`0,1,2,3...`	`0,1,2,3...`

Asignar un array completo	`%set array=(v1 v2...)`	`$set –A array v1 v2...`	`$array=(v1 v2 v3...)`
			ó **(1)**
			`$array=([1]=v1 [2]=v2 ...)`

Asignar un valor independiente	`%set array[indice]=v1` **(2)**	`$array[indice]=v1`	`$array[indice]=v1`
Ver el array completo	`%echo $array`	`$echo ${array[*]}`	`$echo ${array[*]}`
Ver un valor del array	`% echo $array[indice]`	`$echo ${array[indice]}`	`$echo ${array[indice]}`
Ver un rango de valores	`% echo $array[ind1-ind2]`	-	-
Ver el número de valores	`%echo $#array`	`$echo ${#array[*]}`	`$echo ${#array[*]}`

Tabla 3.2: Ejemplos comparativos de operaciones con arrays en las diferentes shells

(1) La segunda notación admite la posibilidad de asignar cada valor a un subíndice concreto.

(2) Si asignamos un valor a un subíndice que no haya sido asignado al principio, se genera un error 'subíndice fuera de rango'.

En general, la asignación y uso de arrays en C shell [16] es más simple y legible, ya que se asume que todas las variables son arrays desde que se asignan, sin embargo, estos arrays quedan declarados y no se puede por ejemplo usar subíndices que no se hayan establecido desde el principio. En Korn [28] y Bash [29] es obligatorio el uso de llaves `{}` para delimitar la variable cuando es un array, en caso contrario, sólo se referencia al primer valor. Bash, además, permite declarar una variable como array y hacer una asignación completa indicando los números de subíndices.

3.1.7.2 Manejo de cadenas en variables

Korn [28] y Bash [29] poseen una serie de opciones muy interesantes para extraer partes de una cadena asignada a una variable. C shell [16] y Bourne carecen de estas funcionalidades. Además, Bash posee algunas adicionales. En la Tabla 3.3 se especifican, junto con las shell que tienen dichas opciones.

Función	Descripción	Ksh	Bash
`${variable#patron}`	Elimina la subcadena más corta por la izquierda que coincide con el *patron* en *variable* y la presenta a salida estándar	Sí	Sí
`${variable##patron}`	Elimina la subcadena más corta por la izquierda que coincide con el *patron* en *variable* y la presenta a salida estándar	Sí	Sí
`${variable%patron}`	Elimina la subcadena más corta por la derecha que coincide con el *patron* en *variable* y la presenta a salida estándar	Sí	Sí
`${variable%%patron}`	Elimina la subcadena más corta por la derecha que coincide con el *patron* en *variable* y la presenta a salida estándar	Sí	Sí
`${#variable}`	Longitud de la cadena en valor numérico de la variable.	Sí	Sí
`${variable:despl.:longit}`	Subcadena desde el carácter *despl* hasta el carácter *longit*. *longit* es opcional y si se omite se muestra la subcadena hasta el final.	No	Sí
`${variable/patrón/cadena}`	Las subcadenas que coinciden con *patron* son cambiadas por *cadena*. *cadena* es opcional, y si se omite se elimina las subcadenas.	No	Sí

Tabla 3.3: Tipos de operaciones con subcadenas

Veamos algunos ejemplos:

```
$ variable='/uno/dos/tres/cuatro'
$ echo ${variable#*/}
 uno/dos/tres/cuatro
$ echo ${variable##*/}
 cuatro
$ echo ${variable%/*}
 /uno/dos/tres
$ echo ${variable%%/*}
  (cadena vacía)
$ echo ${#variable}
 20
```

Los siguientes ejemplos sólo podríamos hacerlos en Bash:

```
$ echo ${variable:4:4}
 /dos
$ echo ${variable/dos/cinco}
 /uno/cinco/tres/cuatro
```

Como podemos ver, estas opciones de manejo de cadenas pueden ser muy interesantes en un programa shell. Bash [29] tiene dos posibilidades más que la Shell Korn [28] según se ve en la Tabla 3.3.

3.1.7.3 Declaración de tipos.

Aunque la filosofía de los lenguajes de programación de las shells es la de simplificar lo máximo posible, en algunos de ellos existe la posibilidad de poder declarar tipos de variables, para así convertirlas en constantes, o hacer que sólo contengan números, por ejemplo.

Shell Bourne [27]: Bourne da la posibilidad de declarar una variable de sólo lectura (una constante), pero no permite ningún otro tipo de declaración. Por ejemplo:

```
$ NOMBRE=juan
$ echo $NOMBRE
 juan
$ NOMBRE=jose
$ echo $NOMBRE
 jose
$ readonly NOMBRE
$ NOMBRE=antonio
 NOMBRE: es sólo lectura
```

C shell [16]: aunque no existe la posibilidad de declarar una variable de un determinado tipo, la distinción de variables se hace a nivel de los operadores que usemos: mientras las asignaciones de variables de cadena y booleanas se hacen con `set`, las asignaciones y operaciones aritméticas se realizan con `@`.

Shell Korn [28]: es compatible con la sentencia `readonly` de Bourne [27], pero además aporta la sentencia `typeset` que permite cambiar atributos de una variable, asemejándose así, a la característica de declaración de variables que tienen muchos lenguajes de programación de alto nivel. Veamos algunas posibilidades:

`typeset -L5` *VARIABLE*	Justifica a la izquierda con un ancho de 5 caracteres
`typeset -R5` *VARIABLE*	Justifica a la derecha con un ancho de 5 caracteres
`typeset -Z5` *VARIABLE*	Justifica a la derecha y rellena con ceros a la izquierda
`typeset -f`	Muestra las funciones declaradas en la shell
`typeset -l` *VARIABLE*	Cualquier letra mayúscula es convertida a minúscula
`typeset -u` *VARIABLE*	Cualquier letra minúscula es convertida a mayúscula
`typeset -i` *VARIABLE*	La variable sólo puede contener números enteros
`typeset -r` *VARIABLE*`=valor`	Se le asigna `valor` y se declara la variable de sólo lectura
`typeset -x` *VARIABLE*	La variable es exportada (equivale a `export`)

Aunque la documentación de la Shell POSIX publicada por HP [33] establece que `typeset` sólo existe para la Shell Korn [28], en las pruebas realizadas, es indiferente usar Korn o POSIX para esta sentencia.

Bash [29]: Aunque Bash también permite el uso de `typeset`, en realidad estaremos usando la sentencia `declare` a la cual está apuntando la anterior, esto se hace por compatibilidad con Korn. La sentencia `declare` no tiene tantas posibilidades como `typeset`:

`typeset -a VARIABLE`	Declara la variable como array

`typeset -r VARIABLE=valor`	Declara la variable de sólo lectura
`typeset -i VARIABLE`	Declara la variable para que sólo contenga enteros
`typeset -x VARIABLE`	Exporta la variable (equivale a `export`)
`typeset -f VARIABLE`	Muestra las funciones declaradas en la shell

Como podemos ver, Bash [29] carece de las posibilidades de alineación y ceros a la izquierda que tiene la Shell Korn [28], estas funciones son muy útiles en scripts que tengan que hacer numeraciones en nombres de ficheros y presentaciones encolumnadas de valores.

3.1.7.4 Variables locales y de entorno

Todas las shells estudiadas tienen un mecanismo de herencia para que los valores de las variables que asignamos puedan ser visibles por parte de los procesos que dichas shells lancen. En el caso de Bourne, Korn y Bash [27, 28, 29], una variable es asignada de la siguiente forma:

```
$ VARIABLE=valor
```

Para desasignar la misma variable, usaremos:

```
$ unset VARIABLE
```

Una vez asignada, dicha variable es visible sólo para la shell actual, y ningún subproceso lanzado por ésta tiene visibilidad sobre ella. Debemos exportar la variable:

```
$ export VARIABLE
```

lo cual permite que una subshell por ejemplo, pueda ver dicho valor. Si modificamos el valor de una variable exportada, no es necesario re-exportar la misma. Veamos un ejemplo de lo expuesto:

```
$ COLOR=rojo
$ echo $$
```

```
 346              # Process Id de la shell actual
$ echo $COLOR
 rojo
$ ksh
$ echo $$
 347              # Process Id de la subshell
$ echo $COLOR
                  # No se muestra nada en la salida estándar
$ exit
$ export COLOR
$ ksh
$ echo $$
 348              # Nuevo Process Id para otra nueva Subshell
$ echo $COLOR
 rojo
$ COLOR=amarillo
$ exit
$ echo $$
 346              # Estamos de nuevo en la shell original
$ echo $COLOR
 rojo             # La shell original conserva el valor que
                  # tenía para la variable
```

En el ejemplo anterior vemos cómo una variable no es visible por parte de los subprocesos de la shell hasta que esta no es exportada. Por otra parte, los cambios que hagan dichos subprocesos en dicha variable lo hacen sobre una copia local, con lo cual no influyen en el valor de la variable original.

Adicionalmente, Korn [28] y Bash [29] poseen la opción `allexport` que exporta automáticamente cualquier variable que sea definida de forma local, sin necesidad de hacerles `export`. Se define de la siguiente forma:

```
$ set -o allexport
```

C shell [16] tiene un tratamiento totalmente diferente para las variables de entorno y locales. En primer lugar, son totalmente independientes, es decir, una variable local no se convierte a variable de entorno, ni viceversa. Las variables locales se define y desasignan de la siguiente forma:

```
set VARIABLE=valor
unset VARIABLE
```

Las variables de entorno usan la sintaxis:

```
setenv VARIABLE valor
unsetenv VARIABLE
```

En el caso de que definiéramos una variable de entorno y una local con el mismo nombre, siempre prevalece la variable local:

```
% setenv COLOR rojo
% echo $COLOR
 rojo
% set COLOR=amarillo
% echo $COLOR
 amarillo
% unset COLOR
% echo $COLOR
 rojo
```

Sólo las variables de entorno son visibles por parte de los subprocesos:

```
% set COLOR=rojo
% setenv COLOR2 amarillo
% echo $COLOR
 rojo
% echo $COLOR2
 amarillo
% echo $$
 499                    # Process Id de la shell actual
% csh
% echo $COLOR
 COLOR: variable no definida
% echo $COLOR2
 amarillo
% echo $$
 500                    # Process Id de la shell actual
% setenv $COLOR2 naranja
% exit
% echo $COLOR2
 amarillo               # La shell original no pierde su valor
```

En este ejemplo vemos cómo las variables de entorno son las únicas visibles, por otra parte, cuando una variable no está definida, no se muestra la cadena vacía como en el caso de las otras shells, sino que se da un mensaje de error.

Otra característica también particular de C shell [16] es que determinadas variables predefinidas para la shell tienen una versión local y otra de entorno. Es el caso

de las variables `path` (local) y `PATH` (entorno). Siempre que modifiquemos la variable local, afectará también a la variable de entorno:

```
% echo $path
 /usr/bin /usr/sbin
% echo $PATH
 /usr/bin:/usr/sbin
% set path=($path /usr/local)
% echo $path
 /usr/bin /usr/sbin /usr/local
% echo $PATH
 /usr/bin:/usr/sbin:/usr/local
```

Podemos, además, observar en este ejemplo que la variable local debe asignarse en forma de array, ya que se comporta como tal. Las variables locales que tienen esta característica son `path`, `home`, `term` y `user`.

Otra característica de la variable `path` es que mantiene una tabla hash para buscar la localización de los ejecutables, esto provoca que aunque modifiquemos dicha variable, no podemos sacar provecho de los cambios hasta que no hagamos el comando:

```
% rehash
```

Como conclusión de esta característica de variables locales y de entorno, podemos ver que en el caso de Bourne, Korn y Bash [27, 28, 29], la sintaxis es mucho más simple de cara al usuario y, además, se mantiene una relación entre variables locales y de entorno mediante la exportación, que no mantiene C shell [16]. En el caso de esta última, la relación está implementada sólo para unas cuantas variables del entorno que están predefinidas.

3.1.8 Sentencias de Control de Flujo de la Ejecución

Este tipo de sentencias nos van a permitir alterar la ejecución secuencial de los comandos en un script. En todos los casos son muy similares a las que podemos encontrar en un lenguaje de alto nivel como C, y tendremos por lo tanto la posibilidad de crear condicionales y bucles. Puesto que todas las shells poseen dichas estructuras, nos centraremos en ver sólo las aportaciones adicionales de cada shell.

Shell Bourne [27]: Al ser la primera shell históricamente, aporta la mayoría de las sentencias de control, que son:

Condicionales:

```
if  'lista de comandos 1'
then
'lista de comandos 2'
[elif  'lista de comandos 3'
then
'lista de comandos 4' ....]
[else
'lista de comandos X']
fi
```

```
case $VARIABLE in
valor1)
 'lista de comandos 1'
;;
valor2)
'lista de comandos 2'
;; ...
esac
```

Bucles:

```
for VARIABLE [in 'lista de valores']
do
'lista de comandos'
done
```

```
while 'lista de comandos 1'
do
'lista de comandos 2'
done
```

```
until  'lista de comandos 1'
do
'lista de comandos 2'
done
```

En todos los casos excepto en el bucle `for` y `case`, la primera lista de comandos es determinante para decidir cómo sigue la ejecución, ya que se usa el valor de retorno del último comando (la variable `$?`) como valor de verdadero o falso. En el bucle `for`, si se omite la lista de valores, se aplican los valores dados como argumentos al script (`$1`, `$2`...).

Shell Korn [28]: Aporta como novedad la sentencia `select` que permite crear un menú de opciones con la posibilidad de tratar posteriormente la selección hecha a

través de entrada estándar. Generalmente dentro de la lista de comandos se pone una sentencia `case`.

```
select VARIABLE in 'lista de valores'; do 'lista de comandos' done
```

Shell Bash [29]: En esta shell tenemos algo que se echaba de menos hasta el momento, y es la posibilidad de usar un bucle `for` con la misma estructura que los que se usan en lenguaje C.

```
for((expresion1; expresion2; expresion3)); do 'lista de comandos';
done
```

Inicialmente se evalúa *expresion1*. A continuación se evalúa *expresion2*, y si da un resultado de 0 (Verdadero) entonces se para la ejecución. Si no es así, se evalúa *expresion3* y se ejecuta la lista de comandos. Al acabar, se evalúa de nuevo *expresion2*, y se continúa con el bucle de nuevo si no da un valor de 0. Esto permite por ejemplo hacer ejecuciones de la forma:

```
for (( I=1; I<=10; I++ )); do echo $I; done
```

Para conseguir algo parecido en la Shell Korn o Bourne, había que complicar mucho el script, por lo tanto, esto nos simplifica mucho el trabajo.

C shell [16]: Hemos dejado esta shell para el final, ya que es la que presenta mayores diferencias, por el hecho ya comentado de que usa una sintaxis muy similar a la del lenguaje C. En este caso tenemos las siguientes formas para las sentencias descritas antes:

Condicionales

```
if ( expresion1 )
   then 'lista de comandos 1'
   [ elseif ( expresion2 )
       then 'lista de comandos 2'...]
   [else 'lista de comandos X']
 endif

switch ( VARIABLE )
   case valor1:
      'lista de comandos1'
      breaksw
```

```
   case valor2:
      'lista de comandos2'
      breaksw
   default:
      'lista de comandos final'
      breaksw
endsw
```

Bucles

```
while (expresion)
     'lista de comandos'
end

foreach variable ('lista de valores')
     'lista de comandos'
end


loop:
    'lista de comandos'
    goto loop


repeat n comando
```

Básicamente, aunque la sintaxis es diferente, podemos ver que consiguen los mismos resultados que las que tenemos en la Shell Korn [28]. Cabe destacar que el bucle `for` de C shell [16] no sigue la sintaxis del lenguaje C, sino que es similar al de Korn. El bucle `for` más similar al de lenguaje C es el de Bash [29]. Una posibilidad que no tienen las otras shells es la de repetir varias veces un comando con la sentencia `repeat`.

Como conclusión de esta característica de las sentencias de control, vemos que todas las shells estudiadas poseen la mayoría de las posibilidades, aunque la que prácticamente permite más es Bash [29]. Algunos usuarios seleccionan C shell por su familiaridad con el lenguaje C. Esto puede ser un factor de decisión, pero como hemos visto, en la mayoría de los casos, obtenemos más ventajas seleccionando cualquier otra shell.

3.1.9 Redirección de Entrada y Salida. Pipes

Las operaciones de redirección nos permiten enviar la salida o tomar la entrada de un comando hacia, o desde un fichero. Los *pipes* o tuberías, canalizan la salida de un comando a la entrada del siguiente. Otra posibilidad es la de usar descriptores de ficheros, que actúan a modo de punteros, permitiendo que los asignemos de forma permanente a un fichero, para entrada o salida. Hay que tener en cuenta que cuando hacemos una redirección de salida, si el fichero existe, borra su contenido original, para evitarlo, tenemos la posibilidad de hacer un *append*, es decir, añadir texto al final del que hay actualmente. También para impedir machacar el contenido de un fichero ya existente por error, algunas shells pueden activar la opción `noclobber`.

Genéricamente, las operaciones de redirección las podemos resumir en la siguiente línea, donde el carácter | indica una posibilidad u otra,

`[exec] [`*comando* `|` *descriptor*`] [ < | >] [&] [`*fichero* `|` *descriptor*`] [-]`

A lo anterior tenemos que añadir las operaciones de *pipes*, *append*, *here-document* y *co-procesos*. Además, no todas las combinaciones son posibles, por lo que a continuación mostramos en la Tabla 3.4 las operaciones permitidas, las shells que las admiten, y comentaremos algunas excepciones.

Operación	**Comentarios**	**Shells que la permiten**
descriptores de ficheros	Punteros que permiten ser asignados de forma permanente a un fichero durante la ejecución del script. Los descriptores 0, 1 y 2 siempre son respectivamente entrada estándar, salida estándar, y salida de errores.	Bourne y Korn: 3 al 9 Bash: indefinidos C shell: no los permite
Noclobber	Opción que impide sobrescribir un fichero existente cuando se le redirecciona de salida. Provoca un mensaje de error	Bourne: no lo permite Korn y Bash: set -o noclobber C shell: set noclobber
Cmd [0]< file	El comando 'cmd' toma la entrada del fichero 'file'	Bourne, Korn, C shell, Bash
Cmd [1]> file	El comando 'cmd' envía su salida al fichero 'file'	Bourne, Korn, C shell, Bash

Tabla 3.4: Operaciones de redirección de datos

Operación	**Comentarios**	**Shells que la permiten**
Cmd 2> file	El comando 'cmd' envía su salida de errores al fichero 'file'	Bourne, Korn, C shell, Bash

Cmd [2]>> file	Redirige la salida estándar o la salida de errores del comando 'cmd' al fichero 'file', pero no borra el contenido adicional, sino que lo añade.	Bourne, Korn, C shell, Bash
Cmd >| file	Si noclobber está activo, ignora la opción y borra el contenido del fichero 'file' si este existe.	Korn, Bash
Cmd >! File	Si noclobber está activo, ignora la opción y borra el contenido del fichero 'file' si este existe. Si 'file' no existe, se crea.	C shell
Cmd >>! File	Si noclobber está activo, y 'file' no existe, provoca un error, y no se crea. Si 'file' existe, se añade la salida al contenido del fichero	C shell
Exec fd<file	Asigna el descriptor de fichero 'fd' al fichero 'file' para entrada de datos	Bourne, Korn, Bash
Exec fd>file	Asigna el descriptor de fichero 'fd' al fichero 'file' para salida de datos	Bourne, Korn, Bash
Cmd <&fd	Usa el fichero asignado al descriptor 'fd' como entrada del comando 'cmd'	Bourne, Korn, Bash
Cmd >&fd	Usa el fichero asignado al descriptor 'fd' como salida del comando 'cmd'	Bourne, Korn, Bash
Exec [fd]<&-	Cierra el descriptor 'fd' de entrada. Si se omite, asume 0	Bourne, Korn, Bash
Exec [fd]>&-	Cierra el descriptor 'fd' de salida. Si se omite, asume 1	Bourne, Korn, Bash
Exec [fd1]<&fd2	Si 'fd2' es un descriptor de entrada, 'fd1' se convierte en una copia de 'fd2'. Si se omite 'fd1', se asume 0.	Korn, Bash
Exec [fd1]>&[fd2]	El descriptor 'fd1' se convierte en una copia del descriptor 'fd2' para salida. Si se omite 'fd1', se asume 1. Si no se pone 'fd1' ni 'fd2', asume 2>&1.	Korn, Bash
Exec [fd1]<&fd2\-	Mueve el descriptor de entrada 'fd2' a 'fd1', y cierra 'fd2'. Omitido 'fd1', asume 0. NOTA: El carácter '\' se usa para romper el significado de '-' como carácter especial.	Bash (Sólo a partir de la versión 2.05b.0(1)-release)
Exec [fd1]>&fd2\-	Mueve el descriptor de salida 'fd2' a 'fd1' y cierra 'fd2'. Omitido 'fd1', asume 1. NOTA: El carácter '\' se usa para romper el significado de '-' como carácter especial.	Bash (Sólo a partir de la versión 2.05b.0(1)-release)
Exec fd<> file	Abre el fichero 'file' de entrada y salida con el descriptor 'fd'.	Bourne, Korn, Bash
cmd << tag texto texto ... Tag	Denominado 'here document' lee todo el contenido desde la primera etiqueta 'tag' hasta encontrar otra etiqueta igual, y lo pasa como entrada al comando cmd. Útil para pasar comandos a un comando interactivo como ftp.	Bourne, Korn, Bash, C shell
Cmd1 | cmd2	La salida estándar de 'cmd1' es conducida (pipe) a la entrada estándar de 'cmd2'.	Bourne, Korn, Bash, C shell

Tabla 3.4 (continuación): Operaciones de redirección de datos

Operación	**Comentarios**	**Shells que la permiten**

Cmd |&	Co-procesos, o pipelines de doble sentido. cmd se lanza en background, y el script o la shell actual puede comunicarse para salida con 'print -p' y para entrada con 'read -p' con cmd2.	Korn
Cmd1 |& cmd2	La salida estándar y la salida de errores del comando uno son redirigidas a la entrada del comando 2	C shell
Cmd1 2>&1 | cmd2	La salida estándar y la salida de errores del comando uno son redirigidas a la entrada del comando 2	Bourne, Korn, Bash

Tabla 3.4 (continuación): Operaciones de redirección de datos

Los comandos que van precedidos de la sentencia `exec` deben hacerse así si queremos que la asignación, cierre o cambio del descriptor se haga en la shell actual. Si no le ponemos `exec`, se ejecutará para el comando en el que esté especificada la operación de redirección.

Como podemos apreciar del cuadro anterior, C shell [16] es la más limitada en cuanto a posibilidades. Bourne, Korn y Bash [27, 28, 29] permiten casi todas las operaciones. La operación de mover descriptores de ficheros sólo la permite Bash en la última versión, y es algo que podemos hacer en cualquier caso copiando primero, y cerrando después el primer descriptor.

Sí cabe especial mención la posibilidad de hacer co-procesos en la Shell Korn [28]. Ninguna otra shell permite esta operación. Puede ser útil para lanzar un proceso en background al que solicitamos resultados. La clave está en usar la opción `-p` en los comandos `print` y `read`. Como ejemplo, podemos ver cómo lanzar la calculadora `bc` en background para hacer cálculos en un script:

```
# bc |&
 [1]     1025
# print -p "2*5"
# read -p var
# echo $var
 10
```

Esto puede simplificar las cosas mucho a la hora de comunicar procesos entre sí en los scripts. Nótese que las dos últimas expresiones de la Tabla 3.4 son muy similares a la anterior, pero difieren en que los dos comandos deben estar presentes en la línea,

mientras que los coprocesos implican a un comando lanzado en background, y a la shell interactiva o el script que lo lanza.

3.1.10Funciones

La posibilidad de agrupar una serie de comandos bajo la denominación de una función es algo que puede ser muy útil para simplificar el código de un script. Básicamente las funciones se declaran al principio de dicho script, y luego podremos usarla en cualquier punto como si fuera un comando. En cierto modo, podemos entender que una función es un *script creado en la memoria de la shell.* Su código se puede usar mientras la shell está activa, y en el momento que la cerramos, tendríamos que volver a declarar dicha función para usarla otra vez. Veamos lo que permite cada shell.

Shell Bourne [27]: La sintaxis para declarar una función en este caso es la siguiente.

```
nombre_funcion ()
{
comando1
comando2
...
}
```

A partir de la declaración de la función, podemos usar `nombre_funcion` como si fuera un comando más de la shell.

C shell [16]: No existe la posibilidad de declarar funciones. Lo más parecido es crear código delimitado por etiquetas y usar la sentencia `goto`. Esta opción no es muy elegante.

Shell Korn [28]: Permite la misma posibilidad de la Shell Bourne [27], pero además da la opción de declarar una función con una sintaxis un poco más clara.

```
function nombre_funcion
{
comando1
```

```
comando2
...
}
```

Al declararlas con la sentencia `function` queda más claro lo que estamos haciendo. Además, permite la posibilidad de declarar una variable de forma local a la función, lo que evita que su valor quede definido fuera de esta. Por ejemplo:

```
$ function variables
 {
 VARIABLE1=hola
 local VARIABLE2=adios
 echo $VARIABLE1
 echo $VARIABLE2
 }
$ variables
 hola
 adios
$ echo $VARIABLE1
 hola
$echo $VARIABLE2
 (variable 2 no está definida y no se muestra, es local a la función)
```

Shell Bash [29]: Totalmente compatible con Bourne y Korn. Tiene las mismas posibilidades que las dos anteriores.

3.1.11 Shells Libres

Finalmente, después de hacer una comparativa totalmente técnica, debemos de tener en cuenta si la shell que estamos usando está sujeta a licencia o no. En el caso de que nuestro sistema operativo sea comercial (HP-UX, Solaris, AIX, TRU 64,...) no será ningún problema, ya que la licencia de la shell irá incluida con dicho sistema, pero no ocurrirá lo mismo si estamos con un entorno libre (Linux, FreeBSD, NetBSD...).

De las shells estudiadas, la única que no está sujeta a licencia es Bash [29]. Sin embargo, existen alternativas. De la Shell Korn tenemos `pdksh` [31] (Public Domain Korn Shell) que es totalmente libre, pero que no presenta todas las funcionalidades de la Korn de AT&T.

Existe una alternativa libre a C shell [16] que es `tcsh` [18] (Tenex Shell). Esta es un superconjunto de C shell que principalmente aporta la posibilidad de edición de línea de comandos y el completado de nombres de ficheros con la tecla *tab*, de forma similar a Bash [29], lo que la hace una buena candidata a personas acostumbradas a C shell que quieran disfrutar de esas características interactivas.

3.1.12 Diferencias Entre Shell Posix y Shell Korn

En varias ocasiones hemos citado a la Shell POSIX. Como ya comentamos, esta shell es la que se utiliza por defecto en el entorno HP-UX de los servidores Hewlett Packard. Las diferencias entre la Shell Korn y la Shell POSIX, como ya dijimos, son muy pocas, y no afectan a lo estudiado por el momento, pero en esta comparativa no podíamos dejar de citarlas. Según la documentación oficial de HP [33], estas son las principales diferencias:

- La Shell POSIX busca los nombres de funciones antes que los nombres de las sentencias internas de la shell. La Shell Korn lo hace al contrario. Esto sólo influye en el caso de que definiéramos una función con algún nombre de sentencia, como por ejemplo `while`.

- Para tener el mismo comportamiento que la Shell Korn con respecto al punto anterior, la Shell POSIX aporta la sentencia `command` que permite ejecutar una sentencia ignorando si hay alguna función definida con el mismo nombre.

- La sentencia `time` deja de ser sentencia, y se convierte en el comando estándar POSIX '/bin/time'. Este comando permite medir los tiempos de ejecución de cualquier proceso.

- `readonly` y `export` proporcionan una salida con las variables definidas con esas opciones, que es posible redireccionar como entrada para definirlas en otra shell.

- `unalias` tiene una opción para poder eliminar de golpe todos los alias del entorno.

- `umask` aporta la opción `-s` que permite asignar máscaras para permisos en nuevos ficheros de forma simbólica.

3.2. Conclusiones

Los puntos anteriores se han sintetizado en la Tabla 3.5. Esta tabla es similar a la que ya se ha presentado en estudios previos [15]. De esta manera, y de un solo vistazo, podemos ver qué shell posee determinadas características.

En cualquier caso, es prácticamente imposible que esta tabla sea completa, ya que dos shells que poseen una misma característica se pueden diferenciar bastante en la forma y en las posibilidades que dan al usuario, por lo que debemos analizarla con más detalle, como hemos hecho en los puntos anteriores.

Un administrador de sistemas, a la hora de decidir cuál es la shell que más se adapta a sus necesidades, debe valorar en primer lugar si va a usarla para desarrollar scripts, o para usarla simplemente en modo interactivo. Como hemos visto en los puntos anteriores, la shell que mejores posibilidades de uso interactivo tiene es Bash [29], por el histórico de comandos con el uso de los cursores, o el completado de nombres de comandos, variables, hosts, etc.

Característica/Shell	Bourne	C shell	Korn	Bash
Histórico de comandos		X	X	X
Edición de línea de comandos		X	X	X
Seguimiento invisible de enlaces simbólicos			X	X
Completado de nombres de ficheros		X	X	X

Característica/Shell	Bourne	C shell	Korn	Bash
Completado de comandos			X (pdksh)	X
Completado de nombres de usuarios				X
Completado de nombres de hosts				X
Completado de variables				X
Facilidades para el uso del prompt de usuario				X
Evaluación de expresiones aritméticas y lógicas	(comando expr)	X	X	X
Arrays		X	X	X
Manejo de cadenas en variables			X	X
Declaración de tipos en variables	Sólo readonly		X	X
Variables locales y de entorno	X	X	X	X
Sentencias de control de flujo	X	X	X	X
Redirección de entrada y salida. Pipes	X	X	X	X
Funciones	X		X	X
Libre distribución			pdksh	X

Tabla 3.5: Resumen de las características de las shells

En cambio, si el uso que se va a dar es para desarrollar scripts, dependerá mucho de las funcionalidades que deseemos usar. Un programador de C se puede decantar por C shell por el hecho de la similitud con la sintaxis de dicho lenguaje. La Shell Korn [28] ofrece más posibilidades que las otras en las declaraciones de variables, ya que permite crear variables de cadena con restricciones de alineación. Sin embargo, las operaciones con cadenas más avanzadas las ofrece Bash [29]. También sucede lo mismo con las sentencias de control de flujo, ya que Bash [29] ofrece un bucle `for` con una sintaxis más avanzada.

En algunos casos, sin embargo, estaremos condicionados a lo que nos aporta el sistema operativo, por ejemplo, en HP-UX, es más cómodo usar la Shell POSIX que ya viene instalada, que tener que compilar e instalar Bash. En el caso de Solaris, la shell por defecto es Bourne, y para hacer scripts de sistema para arranque y parada, sería

conveniente usar ésta, ya que es la que tenemos accesible en el directorio de arranque /bin.

En cuanto a la portabilidad, las mayores posibilidades de migración se presentan de Bourne a Korn ya que la compatibilidad en Korn es completa hacia el código de Bourne. Entre Korn y Bash tendríamos que tener sólo especialmente en aquellos aspectos donde se diferencian, por ejemplo, el uso de la sentencia typeset para declaración de tipos, pero en general la compatibilidad también sería total. C shell sería la que presentaría mayores problemas para migrar su código a cualquier otra shell.

Si valoramos globalmente teniendo en cuenta qué shell posee más características de las estudiadas, vemos que es Bash. Además, salvo en algunos de los puntos vistos, como el caso de la declaración de variables, donde la Shell Korn aporta más ventajas, en líneas generales, Bash es la más completa de todas. Tenemos el valor añadido de que es una de las pocas shells que no está sujeta a ninguna licencia, motivo por el cual es la usada por defecto en Linux. Por lo tanto no es atrevido sugerir que en los casos donde tenemos plena libertad de elección, Bash será la más acertada.

CAPÍTULO 4. TÉCNICAS BÁSICAS DE LA PROGRAMACIÓN SHELL

Según lo expuesto en el capítulo anterior, por los motivos ya explicados, la shell más adecuada en la mayoría de los casos es Bash, por lo tanto, nos centraremos en dicha shell a partir de ahora, sin menosprecio de las demás. El propósito del presente capítulo es enumerar las diferentes características y técnicas de la programación shell, documentándolo con extractos de código de scripts realizados para aplicaciones prácticas, que se presentarán en el capítulo seis completos, junto con otros scripts ya existentes en diferentes sistemas operativos Linux/Unix para gestionar el arranque y parada de servicios y configuración del sistema.

Puesto que algunas de las características que citaremos en este capítulo ya han sido explicadas exhaustivamente, sólo se explicarán con detalle aquellos aspectos no vistos aún, haciendo referencia al capítulo donde se hayan comentado los ya vistos.

4.1. Comenzando a Crear un Script.

En principio, consideraremos como script a cualquier fichero de texto que contenga una secuencia de comandos de la shell que estemos usando, con una sintaxis correcta. En los sistemas Unix/Linux se pueden usar estos scripts de muy diversas formas:

Arranque y parada del sistema: El directorio `/etc/init.d` (o `/sbin/init.d` en HP-UX) contiene dichos scripts, que son llamados en el inicio y parada del sistema operativo, o cuando se produce un cambio de nivel de ejecución (se arrancan o paran servicios del sistema). La finalidad de estos scripts es principalmente la de arrancar procesos que actúen como servicios, como por ejemplo el demonio `cron` que programa tareas, o bien realizar configuraciones iniciales, como establecer las direcciones TCP/IP de las tarjetas de red.

Aunque la implementación de estos scripts varía de un sistema operativo a otro, la técnica usada en todos los casos es similar. Suelen usar una estructura `case` para analizar la variable `$1` que es asignada al primer argumento con el que se le llama. En función de que este argumento sea `start` o `stop` o cualquier otra opción, se arrancarán dichos servicios o se detendrán. Un ejemplo muy típico es el script que arranca el demonio `cron` para programación de tareas. El uso de dicho script es el siguiente en el caso de Red Hat Linux:

```
# /etc/init.d/crond start
 Iniciando crond:                                          [  OK  ]
# /etc/init.d/crond stop
 Parando crond:                                            [  OK  ]
#
```

Podemos ver un extracto de la estructura `case` que se usa. Como podemos ver, se utilizan además funciones predefinidas previamente:

```
…
case "$1" in
  start)
        start
        ;;
  stop)
        stop
        ;;
  restart)
        restart
        ;;
  reload)
        reload
        ;;
  status)
        rhstatus
        ;;
  condrestart)
        [ -f /var/lock/subsys/crond ] && restart || :
        ;;
  *)
     echo $"Usage: $0 {start|stop|status|reload|restart|condrestart}"
     exit 1
esac
…
```

Inicio de sesión de usuario: Cuando un usuario inicia una sesión con Bash [29], se ejecutan los comandos de `/etc/profile`, y posteriormente busca uno de los ficheros `$HOME/.bash_profile`, `$HOME/.bash.login` o `$HOME/.profile` en ese orden, y ejecuta los comandos del primero que encuentre. Nótese que la variable `HOME` contiene el directorio de trabajo del usuario.

En todos los casos, dichos scripts no tienen permisos de ejecución ya que en realidad son invocados por la propia shell de inicio de sesión y leídos sus comandos uno a uno. Si no se hiciera así, sino de la forma tradicional, se generaría un subproceso shell que sería el que se quedaría con las definiciones de variables de entorno que es lo que se suele definir en estos scripts de inicio. De esa forma, al acabar la ejecución, se perderían dichas definiciones.

Como comando: Podemos crearnos nuestra propia secuencia de comandos para realizar una tarea que siempre se repite de la misma forma, y evitando así tener que teclear todos los comandos manualmente. Además estos scripts pueden programarse para que se lancen a una determinada hora gracias a los demonios de programación de tareas (`cron` o `anacron`).

4.1.1 Ejecutar un Script

Para poder ejecutar un script, lo habitual es asignar permisos de ejecución a dicho fichero usando el comando `chmod`:

```
$chmod +x  mi_script
```

Con esto, lo único que tendremos que hacer es llamar al script como si se tratara de un comando más, teniendo en cuenta que el directorio donde se encuentre lo tengamos disponible como parte de la variable `PATH`, que contiene la lista de directorios donde podemos buscar ficheros ejecutables.

```
$ mi_script
```

Lo que sucede en este caso podemos verlo en el ejemplo de la Figura 4.1. El usuario ejecuta una shell Bash donde llama al script `Miapp`. Esto provoca la generación de una shell que a su vez llama a cada uno de los comandos del script.

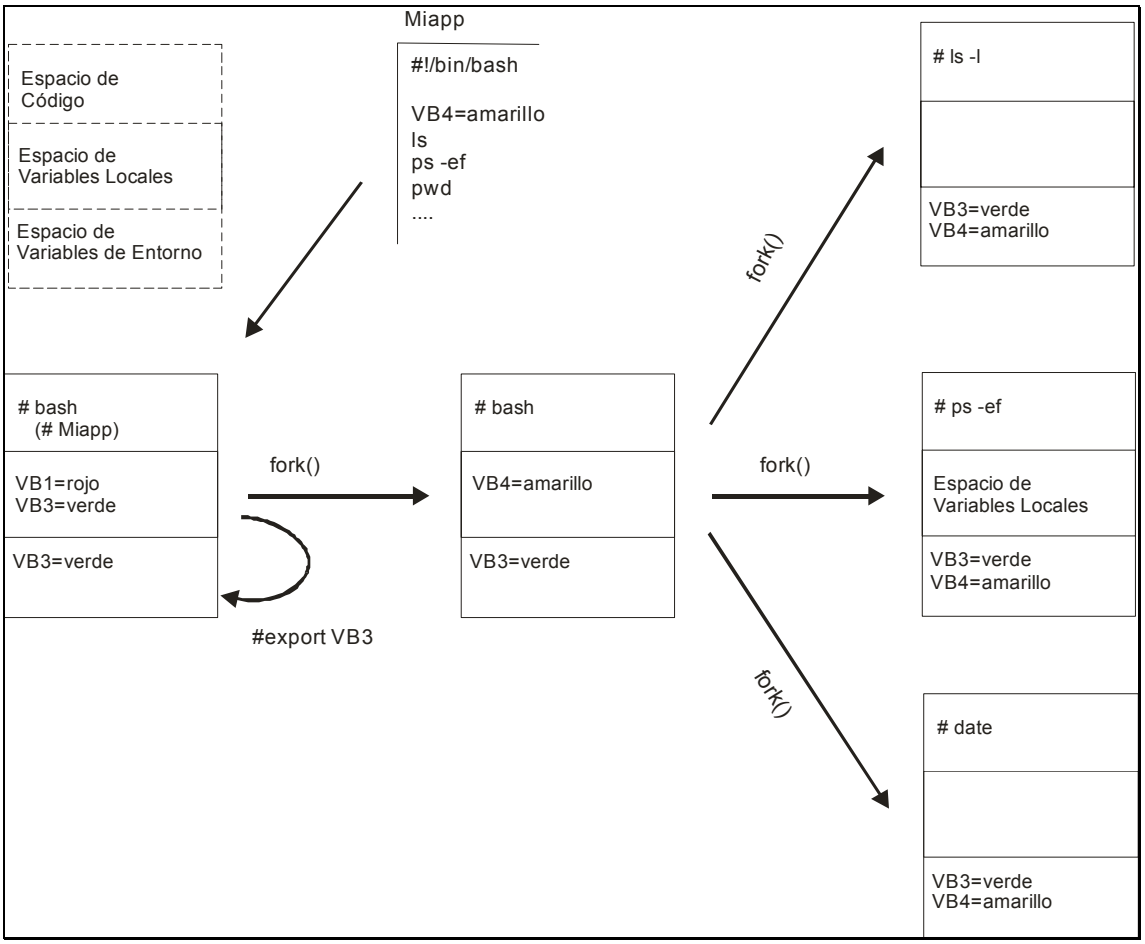

Figura 4.1: Ejecución de un script y sus diferentes comandos

Aunque el uso anterior es el habitual, no es obligatorio que un script tenga permisos de ejecución. En ese caso, podemos usarlo de la siguiente forma:

```
$bash mi_script
```

En realidad lo que hacemos en este caso es llamar a una shell bash, usando como argumento el fichero `mi_script`. Dicha shell ejecutará todos los comandos que se encuentran en dicho fichero.

Nótese que en el primer caso el subproceso shell se genera automáticamente gracias a los permisos de ejecución del script, pero en el segundo caso, somos nosotros los que explícitamente decimos con qué shell debe ejecutarse el fichero. Esto nos lleva a reflexionar qué pasaría si intentamos ejecutar un script hecho para C shell en una Shell Bash. Cuando un script con permisos de ejecución es llamado desde una shell, si no se indica lo contrario, se lanza un subproceso shell idéntico al llamante. Por lo tanto, si el script está hecho para una shell distinta, provocaría errores de sintaxis. Para evitar este problema en la primera línea del script podemos designar con qué shell se ejecuta:

```
#! path_de_la_shell
```

Por ejemplo, podemos ver las primeras líneas del script `lj-ps-pdf.sh`:

```
#!/bin/bash
…
# CONVERSION DE LJ A PS
for i in *.lj
do
…
```

O del script `renombra.sh` que usa Korn:

```
#!/bin/ksh
…
DIRECTORIO=$1
NOMBASE=$2
NUMBASE=$3
DIGITOS=$4
…
```

Esta debe ser la primera línea del script y el carácter # debe ser el primero de la línea. Además, podemos aprovechar esta línea para indicar argumentos de la shell que usamos. Por ejemplo, cuando lanzamos una shell con la opción `-x` muestra en pantalla cada línea de comandos antes de ser enviada al intérprete de comandos, lo cual es bastante útil para depurar el código de un script. Esta opción puede añadirse en la primera línea del script:

```
#! /bin/bash -x
```

4.1.2 Comentarios

El carácter # se utiliza para insertar comentarios. Por definición, cualquier cadena de caracteres a la derecha del carácter # es ignorada. La única excepción a dicho comportamiento es, como hemos dicho en el punto anterior, cuando está en la primera línea de un script, y con la sintaxis indicada. Podemos ver un ejemplo de comentarios en el script `cdbackup.sh`

```
…
# FICHERO DONDE SE HARA EL BACKUP
[ -z $CDBACKUP_DUMP ] && \
CDBACKUP_DUMP=${CDBACKUP_TARGET_DIR}/${FECHA}.$$.dump

# FICHERO CON LA IMAGEN ISO PARA GRABAR
[ -z $CDBACKUP_ISO ] && \
CDBACKUP_ISO=${CDBACKUP_TARGET_DIR}/${FECHA}.$$.iso
export CDBACKUP_DUMP CDBACKUP_ISO
…
```

4.1.3 Uso de Variables en un Script. Arrays. Variables Especiales

El uso de las variables es esencial en la programación shell, ya que son la clave para poder almacenar valores que podamos usar con posterioridad en un programa, o para poder pasar parámetros entre dos scripts cuando uno llama a otro. Las variables se asignan usando la siguiente sintaxis:

```
$ VARIABLE=valor
```

donde el valor será una cadena de caracteres, con o sin entrecomillado, que a partir de ese momento podremos recuperar convirtiendo a dicha variable en un string cuando la precedamos del carácter `$`:

```
$ $VARIABLE
```

Una forma especial de referenciar a la variable es usar la forma `${VARIABLE}`. En esta notación acotamos el nombre de la variable, esto tiene utilidad cuando queremos por ejemplo encadenar dos variables para que muestren su valor como una sola cadena. Si no lo hiciéramos de esta forma, sino sin las llaves, provocaría un error al no saberse dónde acaba una variable y empieza la siguiente. Podemos ver un ejemplo de esto en el script `renombra.sh`:

```
…
for i in  $DIRECTORIO/*.jpg  $DIRECTORIO/*.JPG
do
  mv "$i" "${DIRECTORIO}/${NOMBASE}${NUMBASE}.jpg" 2> /dev/null
  (( NUMBASE = NUMBASE + 1 ))
done
…
```

Esta notación con llaves es también utilizada para manejo de variables array y para operaciones de cadenas sobre variables como indicamos en el punto 3.1.7. Para terminar de completar lo ya expuesto, debemos añadir que también se utiliza dicha notación para asignar o mostrar valores por defecto o mensajes de error para variables que no tengan valor definido, según la siguiente sintaxis:

`${`*`VARIABLE`*`:-`*`valor`*`}` Si `VARIABLE` no tiene `valor` asignado, se muestra `valor`. En caso contrario, se muestra el valor asignado a `VARIABLE`.

`${`*`VARIABLE`*`:=`*`valor`*`}` Si `VARIABLE` no tiene `valor` asignado, se asigna a `VARIABLE` `valor` y a continuación es mostrado. En caso contrario, se muestra el valor de `VARIABLE`, y ésta conserva dicho valor.

`${`*`VARIABLE`*`:?`*`valor`*`}` Si `VARIABLE` no tiene `valor` asignado, se muestra `valor` en la salida estándar de errores, como si fuera un mensaje de error. En caso contrario, se muestra el valor de `VARIABLE` en la salida estándar.

`${`*`VARIABLE`*`:+`*`valor`*`}` Si `VARIABLE` no tiene `valor` asignado, no se hace nada. En caso contrario, se muestra `valor` en la salida estándar.

Podemos ver ejemplos de asignaciones por defecto en el script `cdbackup.sh`:

```
…
[ ! -d ${CDBACKUP_TARGET_DIR:=/tmp} ] && uso && exit 1
[ ! -x ${CDBACKUP_SCRIPT:=grabacd.sh} ] && uso && exit 1
…
echo "Nivel de la copia:   ${CDBACKUP_NIVEL:=0}"
…
echo "Tamano de imagen:    ${CDBACKUP_TAMANNO:=700000}"
…
```

Una variable mantendrá su valor en la shell actual mientras no volvamos a cambiar dicho valor con otra asignación, o la desasignemos usando la sentencia `unset`:

```
$unset VARIABLE
```

Para que una variable pase del espacio local al espacio de entorno, lo haremos exportando dicha variable:

```
$export VARIABLE
```

Podemos ver el proceso en la Figura 4.2:

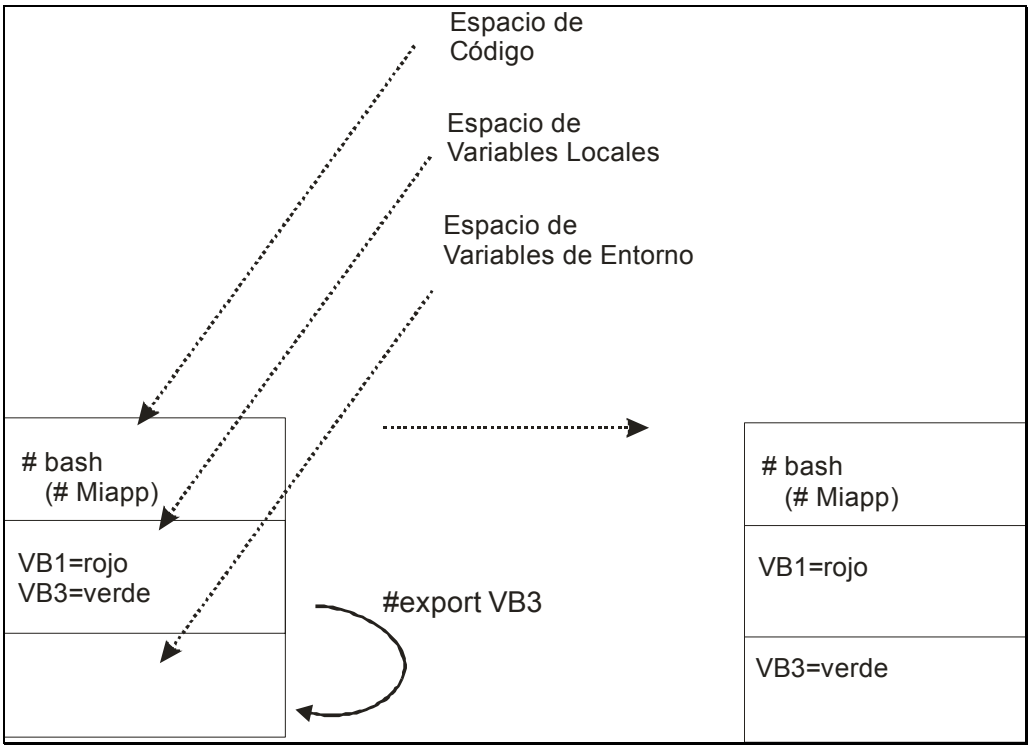

Figura 4.2: Exportación y visibilidad de variables exportadas.

Cuando iniciamos una sesión de shell, siempre se reservan dos espacios de memoria para almacenar variables, el de variables locales y el de variables de entorno. Una variable sólo existirá en el proceso actual, y en los subprocesos que este genere, si se ha exportado. Esto ya fue descrito con detalle en el segundo capítulo.

Para ver todas las variables que hay actualmente tanto locales como de entorno, usaremos la sentencia `set`. Con la sentencia `env` veremos sólo las variables que han sido pasadas al espacio de entorno y que por lo tanto, serán heredadas por los subprocesos.

Como se habrá observado, casi todas las variables de usan nombres en mayúsculas. Esto no es obligatorio, pero suele usarse como convención para distinguir las variables del resto del código, aunque si queremos podemos poner el nombre de una variable en minúsculas. Además, el único carácter especial permitido en los nombres de variables es el guión bajo _.

Otra posibilidad que ofrecen las variables en una shell es la de usarse como arrays. Sólo están permitidos los arrays de una dimensión. La notación para la asignación y uso de arrays ya se indicó en el punto 3.1.7. Podemos ver un ejemplo de uso de arrays en el script `path.sh` que hace uso de uno de ellos para segregar un nombre de fichero en sus componentes (subdirectorios y nombre):

```
…
X=0
until [[ $PATH1 != */* ]]
do
    NAME[$X]=${PATH1%%/*}
    PATH1=${PATH1#*/}
    ((X = X + 1))
done
…
```

Tenemos otro ejemplo en el script `logrotate.sh` que lo usa para extraer el nombre del primer fichero de una lista ordenada por fecha de modificación (extraerá el más antiguo):

```
…
LISTA_FICHEROS=$(echo $1.*) 2>/dev/null
ANTIGUO=${LISTA_FICHEROS%% *}
rm $ANTIGUO
…
```

En este extracto de código, a cada variable del array `NAME`, indexado por la variable `X` se le asigna un componente del nombre de fichero facilitado en la variable `PATH`, que es literalmente *cortada* en trozos usando operaciones de cadena y teniendo como referencia el carácter / para delimitar los nombres. Existen tres variables con un significado especial para la shell:

- `$?` Esta variable contiene el valor de retorno que generó el último comando ejecutado. Dicho valor tiene las siguientes interpretaciones:

0	Comando completado correctamente	(Verdadero)
> 0	Comando completado con errores	(Falso)
127	Comando no encontrado	(Este valor lo genera la propia shell)

Como se puede ver, este valor nos sirve tanto para saber si un comando ha terminado de forma correcta o no, como para tomarlo como valores de verdad o falsedad. De hecho, existen dos comandos que generan las constantes `true` y `false`.

```
$ ls
 backup boot dev home initrd.img media opt root srv tmp var bin cdrom
 etc initrd lib mnt proc sbin sys usr vmlinuz
$ echo $?
 0
$ true
$ echo $?
 0
$ cp
 cp: falta un fichero como argumento
 Pruebe `cp --help' para más información.
$ echo $?
 1
$ false
$ echo $?
 1
$ nocommand
 bash: nocommand: command not found
$ echo $?
 127
$
```

La sentencia `exit` dentro de un script permite devolver un valor de retorno en `$?` para la shell llamante del script.

```
$ ps -f
 UID        PID  PPID  C STIME TTY          TIME CMD
 1001      5594  5592  0 14:03 pts/2    00:00:00 /bin/bash
 1001     25443  5594  0 20:33 pts/2    00:00:00 ps -f
$ sh
$ ps -f
 UID        PID  PPID  C STIME TTY          TIME CMD
 1001      5594  5592  0 14:03 pts/2    00:00:00 /bin/bash
 1001     25444  5594  0 20:33 pts/2    00:00:00 sh
 1001     25445 25444  0 20:33 pts/2    00:00:00 ps -f
$ exit 33
 exit
$ echo $?
 33
$ ps -f
 UID        PID  PPID  C STIME TTY          TIME CMD
 1001      5594  5592  0 14:03 pts/2    00:00:00 /bin/bash
 1001     25446  5594  0 20:33 pts/2    00:00:00 ps -f
$
```

Esto nos puede servir para documentar los posibles casos de error dentro de dicho script, indicando cada valor de retorno diferente de 0 por qué motivo ha sido generado.

- `$$` Esta variable contiene el valor de la propia shell que estamos ejecutando. Una utilidad posible de esta variable es usar dicho valor para generar nombres de ficheros temporales que sean únicos para diferentes ejecuciones de un script. En el script `cortafichero.sh` que se utiliza para separar un fichero de texto ascii en columnas predefinidas, podemos ver el uso de dicha variable para generar ficheros temporales que no coincidan en nombre con los de cualquier otra ejecución paralela del mismo script.

```
…
### SE VAN REALIZANDO LOS DIFERENTES CORTES DE CADA COLUMNA Y SON
### ALMACENADOS EN FICHEROS TEMPORALES
for LONGITUD in $CORTES
do
(( CONTADOR = CONTADOR + 1 ))
(( PRINCIPIO = OFFSET + 1 ))
(( FINAL = OFFSET + LONGITUD ))
typeset -Z3 CONTADOR
cut -c $PRINCIPIO-$FINAL $FICHERO > /tmp/$FICHERO.$CONTADOR.$$
OFFSET=$FINAL
done
…
```

- `$!` Esta variable almacena el PID (process identificator) del último proceso lanzado en background. Gracias a ella, podemos lanzar de forma paralela a la línea de ejecución principal, un proceso que vaya haciendo otro trabajo adicional.

Podemos ver un ejemplo del uso de esta variable en el script `usacursor.sh` que lanza un proceso en background que genera un cursor giratorio en la terminal, mientras se está realizando otra función en la línea principal de ejecución, en este caso, un simple `sleep`:

```
…
echo espere 10 segundos...
./cursor &
sleep 10
kill $!
echo Ahora puede continuar
…
```

4.2. Comunicación con un Script

4.2.1 Argumentos de un Script

Cuando ejecutamos un script, cualquier cadena que añadamos en la misma línea de comando separada por espacios o tabuladores, será interpretada como una lista de argumentos. Estos serán asignados a variables de la siguiente según la Tabla 4.1:

Línea de comando	***comando***	***arg1***	***arg2***	***arg3***	***...***	***arg9***	***arg10***	***arg11***	***...***
Valor asignado	`$0`	`$1`	`$2`	`$3`	`...`	`$9`	`${10}`	`${11}`	`...`

Tabla 4.1: Argumentos de un script.

Además, se asignan dos variables adicionales:

`$#` = Número de argumentos

`$*` = Cadena con todos los argumentos separados por un carácter espacio

En el script `argumentos.sh` podemos ver un ejemplo del uso de esta variable para comprobar si el número de argumentos pasados es el correcto:

```
ARG_INSUF=51

if [ $# -ne 2 ]
then
  echo "Uso: $0 arg1 arg2"
  exit $ARG_INSUF
fi
```

Obsérvese que los argumentos a partir del número 10 deben ser acotados con los caracteres `{}` para referenciarlos. La Shell Bourne [27] sólo soporta hasta el argumento $9. La variable `$0` es asignada al nombre del script, esto puede aprovecharse para saber cómo hemos llamado a un script, en el caso de que tenga varios enlaces (nombres de fichero) apuntando al mismo fichero.

Generalmente, para un uso más coherente y claro de estos argumentos dentro del script, una vez pasados, los asignamos a otras variables con nombres más significativos. Podemos ver un ejemplo de esto al principio del script `renombra.sh:`

```
…
### SE ASIGNAN NOMBRES MAS RECONOCIBLES A LOS ARGUMENTOS
DIRECTORIO=$1
NOMBASE=$2
NUMBASE=$3
DIGITOS=$4
…
```

4.2.2 Sentencia `read`

El paso de argumentos sólo nos permite asignar valores a variables al principio de la ejecución de un script, pero a veces necesitamos tomar un dato durante dicha ejecución. En este caso podemos recurrir a la sentencia `read`. Esta sentencia lee el número de variables que le indiquemos en la línea de comandos, asumiendo que cada variable está separada por un número variable de espacios o tabuladores:

```
$read VB1 VB2 VB3
 uno    dos    tres     cuatro
$echo $VB1
 uno
$echo $VB2
 dos
$echo $VB3
 tres     cuatro
```

Obsérvese cómo la última variable se queda con el resto de la cadena resultante después de separar cada valor por espacios. Podemos ver el uso de dicha sentencia en el script `maneja_rpm.sh`, en el siguiente extracto:

```
…
# SE DA A ELEGIR ENTRE INSTALAR DE UN CD DE DISTRIBUCION LINUX O DE UN
# DIRECTORIO QUE CONTENGA PAQUETES RPM
echo -n "Los paquetes se instalarán desde el cdrom de Linux? (s/n): "
read RESPUESTA
if [[ $RESPUESTA = n* || $RESPUESTA = N* ]]
  then
   echo -n "Indique el directorio que contiene los paquetes: "
   read DIR
fi
cd $DIR
…
```

También podemos encontrar un uso peculiar de la sentencia `read` en el script `rm_shmem.sh`. En este caso, es usada dentro de un bucle que lee líneas de un fichero temporal. Cada línea le pasa dos variables nombradas como `ID` y `NUM`:

```
…
while read ID NUM
do
  if (( NUM == 0 ))
  then
     ipcrm shm $ID
  fi
done < /tmp/$$rm_sh_mem
…
```

4.2.3 Opciones y Argumentos con `getopts`

Es habitual que los comandos unix usen opciones y parámetros combinados en cualquier orden. A este tipo de parámetros se les denomina *parámetros posicionales*. Las opciones se indican con el carácter guión - precediéndolas. Una opción puede ir sola, en cuyo caso generalmente indicamos un tipo de comportamiento que esperamos

del comando, o acompañada de un parámetro posicional, que se suelen usar para pasar un dato al comando con el fin de realizar una acción concreta con este parámetro.

Los parámetros que no son acompañados de opciones van siempre al final de la línea de comandos. En el siguiente ejemplo del comando `useradd` que crea usuarios para trabajar en un sistema Unix, podemos ver opciones independientes, opciones con parámetros posicionales y parámetros independientes:

```
$useradd -u 101 -d /home/pedro -m -c "Pedro Lopez" pedro
```

opciones con parámetros:

`-u 101`	Número de usuario asignado
`-d /home/pedro`	Directorio de trabajo del usuario
`-c "Pedro López"`	Comentarios para el usuario

opciones independientes:

`-m`	Indica que el directorio del usuario debe ser creado

parámetros independientes:

`pedro`	Nombre del usuario para hacer login

Como podemos ver, este tipo de sintaxis sería muy complicada de gestionar con la lista de argumentos estándar que podemos pasar a un script (`$1 $2 $3`), ya que las opciones pueden ir en cualquier orden, e incluso pueden ponerse todas juntas con un sólo guión cuando no necesitan parámetros, como en el siguiente ejemplo:

```
$ls -laF /home
…
$ls -aFl /home
…
```

Ambas líneas de comandos darían exactamente el mismo resultado. La sentencia `getopts` nos permite gestionar este tipo de sintaxis. La sintaxis de `getopts` es la siguiente:

```
getopts lista_de_opciones VARIABLE
```

La lista de opciones es una cadena de caracteres comenzando con el carácter `:`. Cada uno de dichos caracteres se considera una opción. Si uno de los caracteres va precedido de nuevo del carácter "`:`", se asume que debe llevar un parámetro posicional a continuación. Por ejemplo:

```
getopts ":af:dm" vb
```

Admitiría comandos del tipo:

```
$cmd -a -f file -d
$cmd -f file -am
$cmd -m -f file -ad
…
```

En ningún caso se indica si una opción es obligatoria o no. Eso debe ser implementado en el código posterior. Cada vez que invocamos `getopts`, lee la línea de argumentos `$*` que ya indicamos que contiene todos los argumentos pasados al script, y lo procesa basándose en la cadena de opciones indicada. Si encuentra una opción, la deposita en la variable `vb`. Si además dicha opción lleva un parámetro, lo deposita en la variable especial `$OPTARG`. También se actualiza una variable `$OPTIND` que contiene el número de bloques de opciones y parámetros encontrados. En el momento que `getopts` encuentra alguna cadena que no cumple la lista de opciones indicada, deja de analizar la cadena de argumentos.

Generalmente `getopts` irá al principio de un script, dentro de un bucle `while` en el cual tome todas las opciones y parámetros encontrados al principio. Una vez capturados y realizadas las decisiones y acciones correspondientes, el resto de la cadena de argumentos es procesada de la forma habitual. La estructura una vez montada quedaría como en el ejemplo `opciones.sh` que se muestra completo en el siguiente punto. Mostramos aquí la estructura general.

```
while getopts ":abc:de:fg" OPCION
```

```
do
  case $OPCION in
    a)
     …
     ;;
    b)
     …
     ;;
    c)
      …
      …
     ;;
    g)
     …
     ;;
  esac
done
```

Podemos ver un ejemplo concreto de `getopts` en el script `cdbackup.sh`: en este caso el script puede ser invocado de la forma:

```
$cdbackup.sh –d "directorio temporal" –s "script de grabacion" –n
"nivel de backup" –t "tamaño del cd en kb"
```

Esto se implementa mediante la sentencia `getopts` dentro de un bucle `while` con una sentencia `case` para analizar cada opción y asignar las variables adecuadas antes de seguir con la ejecución del script:

```
…
while getopts ":d:s:n:t:" OPCION
do
  case $OPCION in
    d)
     # DIRECTORIO TEMPORAL PARA ALMACENAR FICHEROS
     CDBACKUP_TARGET_DIR=$OPTARG
     ;;
    s)
     # SCRIPT DE GRABACION
     CDBACKUP_SCRIPT=$OPTARG
     ;;
    n)
     # NIVEL DEL BACKUP
     CDBACKUP_NIVEL=$OPTARG
     ;;
    t)
     # TAMANIO DEL CD EN KB
     CDBACKUP_TAMANNO=$OPTARG
  esac
done
…
```

4.3. Variables. Aritmética, Lógica y Cadenas

4.3.1 Aritmética y Lógica

En el punto 3.1.6 comentamos exhaustivamente la sintaxis para el uso de aritmética y lógica en Bash [29] con la sentencia `let`. Podemos ver ejemplos en varios scripts de los expuestos en el siguiente punto, como `cortafichero.sh`, `escanea_ip.sh` o `renombra.sh`. En el caso de `escanea_ip.sh`, este script se hizo originalmente para la Shell Korn y no para Bash. Puesto que en Korn v88 no existe una implementación del bucle `for i in x to y`, hubo que ingeniar una forma de emularlo mediante un bucle while que generara la lista. En la línea donde se hace dicha emulación podemos ver un uso de sentencias 'let' tanto lógicas como aritméticas:

```
…
for i in $( X=1 ; while (( X <= 254 )) ; do echo $X ; (( X = X + 1 ))
; done )
do
   …
done
…
```

En todo caso, esta línea de código sólo muestra cómo emular un bucle `for` que no está soportado en dicha versión de la Shell Korn [28], y se puede simplificar:

```
…
X=1
While ((X<=254))
Do
   …
   ((X=X+1))
   …
Done
…
```

4.3.2 Manejo de Cadenas

En el punto 3.1.7 se comentó el manejo de cadenas en variables. Vimos cómo podíamos extraer parte de los caracteres que componían una variable, basándonos en ciertos criterios. Podemos ver ejemplos del uso de dichas funcionalidades en los scripts `minusculas.sh` `logrotate.sh` y `path.sh`. En este último, vemos cómo se

desglosan todos los componentes de un `path` o nombre completo de un fichero mediante un bucle que va extrayendo dichos componentes y los almacena en un array.

```
…
X=0
until [[ $PATH1 != */* ]]
do
    NAME[$X]=${PATH1%%/*}
    PATH1=${PATH1#*/}
    ((X = X + 1))
done
…
```

4.4. Condiciones

En ocasiones debemos romper la estructura de ejecución lineal de comandos con una toma de decisión. Existen diferentes estructuras que nos permiten hacer esto. En el punto 3.1.8 apuntamos simplemente la sintaxis. Ahora vamos a estudiarlas con detenimiento. Debemos indicar también, que en todos los casos, el principio básico es el de analizar la variable `$?` generada por el último comando ejecutado. De esa forma, un valor de `0` será interpretado como verdad, y un valor diferente de `0` será interpretado como falso.

4.4.1 Cortocircuitos

El cortocircuito (short circuit) es una forma simplificada de condicional. Se denomina así porque consiste en hacer una operación `and` o `or` lógicas con los valores de retorno depositados en `$?` por cada comando ejecutado. Existen dos tipos:

– && Este cortocircuito hace un *and* lógico entre los valores de retorno de los dos comandos entre los que está. Su sintaxis es:

comando1 && *comando2* && *comando3*…

Al hacerse esta operación lógica, si el comando de más a la izquierda termina con valor de falso, no se ejecuta el resto de comandos de la derecha, ya que esto sigue dando un valor de falso. Si el comando de la izquierda termina con valor de verdadero, se sigue ejecutando el comando de la derecha y así sucesivamente. Gracias a esto, lo que obtenemos es que cada comando depende del éxito de todos los anteriores.

Esta forma se suele utilizar para evitar mensajes de error en casos en que por ejemplo un fichero no existe, o que una variable tiene o no tiene el valor adecuado. Previamente analizamos la existencia de dicho fichero o evaluamos la variable correspondiente, y en caso de ser falso, no se ejecuta el comando de la derecha. Podemos ver un ejemplo de esto en el script `/etc/init.d/crond` de inicio del servicio de programación de tareas.

```
…
t=${CRON_VALIDATE_MAILRCPTS:-UNSET}
[ "$t" != "UNSET" ] && export CRON_VALIDATE_MAILRCPTS="$t"
…
```

En este caso, en la primera línea se le da a la variable 't' el valor de la variable `CRON_VALIDATE_MAILRCPTS`, y si dicha variable no está definida, se asigna el valor `UNSET`. En la segunda línea, se comprueba si el valor de `t` no es `UNSET`, en cuyo caso, se exporta la variable `CRON_VALIDATE_MAILRCPTS` asignándole el valor de `t`. Esto lo que hace es exportar dicha variable siempre y cuando tenga ya previamente un valor definido. Si no lo tuviera, no se llega a exportar.

– `||` Este cortocircuito hace un *or* lógico entre los valores de retorno de los dos comandos entre los que está. Su sintaxis es:

comando1 `||` *comando2* `||` *comando3* `...`

En este caso, si el primer comando termina con valor de verdadero, deja de evaluarse, ya que siempre será verdadero el resultado final. Como consecuencia de esto, el comando 2 sólo se ejecutará si el comando 1 termina con error, y así sucesivamente. Dependiendo de la lógica de lo que queramos analizar, será más conveniente usar un

cortocircuito u otro. Podemos ver un uso de `||` en el script de inicio y parada de los servicios DNS *named*:

```
[ -r ${ROOTDIR}/etc/named.conf ] || exit 1
```

En este caso se comprueba si existe el fichero `named.conf` en el directorio `/etc`. Si esto es verdad, entonces no se ejecuta el segundo comando, pero si es falso, no podemos lanzar al demonio `named` ya que no tenemos su fichero de configuración, en cuyo caso, ejecutamos `exit 1` y por lo tanto, el script termina con un valor de retorno de error.

4.4.2 Condicional `if`

Esta es una forma más elaborada de tomar decisiones. Su sintaxis general es la siguiente:

```
…
if    'lista de comandos A'
then
      'lista de comandos B'
[elif 'lista de comandos C'
 then 'lista de comandos D'...]
[ else
      'lista de comandos X' ]
fi
…
```

En su forma más sencilla, `if listaA then listaB fi`, se ejecutan todos los comandos de la lista A. Si el valor de retorno del último comando de esta lista es verdadero, entonces se ejecutará la lista de comandos B. En caso contrario, se seguirá ejecutando los comandos por debajo de `fi`. Cuando utilizamos `else`, en caso de que el último comando de la lista A termine con valor de falso, se ejecutará la lista de comandos X. Como vemos, entre ambas palabras reservadas podríamos anidar otro `if...then...` simplemente usando la palabra reservada `elif`.

Podemos ver un uso de condicional `if` en el script `nuevousuario.sh`

```
# AGREGA EL USUARIO Y DICE SI HA TENIDO ÉXITO O NO LA OPERACION

…
If useradd$ -u  $USER_UID -g $GID -G $GRUPOS -m -d /home/$LOGIN \
   -c $DESCRIPCION -s $SHELL $LOGIN 2> /dev/null
then
   echo "Usuario $LOGIN añadido con exito"
else
   echo "Error al añadir el usuario $LOGIN"
   exit 1
fi
…
```

En este caso, se lee una opción tecleada por el usuario, y dependiendo de que el valor sea uno u otro, se ejecutará un comando `rpm` (manejo de paquetes en Red Hat Linux) que muestre la información del paquete, o que muestre el listado completo de ficheros del paquete. Generalmente, la primera lista de comandos (la que se encuentra entre `if` y `then`) suele ser una sentencia `let` que realiza una evaluación, pero esto no es obligatorio, y podemos utilizar cualquier comando, y su valor de retorno será interpretado como verdadero o falso.

4.4.3 Condicional `case`

Este tipo de estructura condicional nos permite evaluar el valor de una variable, y dependiendo de dicho valor realizar una acción u otra. Su sintaxis es:

```
case $VARIABLE in
valor1)
 'lista de comandos 1'
;;
valor2)
 'lista de comandos 2'
;;
…
esac
```

Generalmente esta estructura se suele utilizar para decidir qué hacer en base a una opción seleccionada por un usuario como parte de un menú o pasada como argumento al propio script. Este segundo caso es exhaustivamente utilizado en los scripts de arranque y parada de servicios de los entornos Unix en general. Podemos ver un uso de esto en el script `crond`:

```
case "$1" in
  start)
        start
        ;;
  stop)
        stop
        ;;
  restart)
        restart
        ;;
  reload)
        reload
        ;;
  condrestart)
        [ -f /var/lock/subsys/crond ] && restart || :
        ;;
  *)
        echo $"Usage: $0 {start|stop|reload|restart|condrestart}"
        exit 1
esac
```

En este caso, la variable `$1` es pasada al script como argumento, y puede ser `start`, `stop`, `status`, etc... Dependiendo del valor que pasemos, el script ejecutará una función diferente que ha sido definida previamente.

Como apunte adicional al ejemplo anterior, vemos un ejemplo de cortocircuito `&&` en la opción `condrestart` donde si existe un determinado fichero, es ejecutada la función `restart`. A su vez, este cortocircuito está unido a otro cortocircuito `||` que en este caso lo que hace es que si todo lo anterior es ejecutado con éxito, no se ejecuta el comando : que es simplemente un comando nulo, y en caso contrario, sí es ejecutado. Esto se hace para evitar un error, ya que si toda la ejecución fuera falsa, la estructura fallaría al no haber ninguna ejecución válida de comando.

También podemos ver un ejemplo muy ilustrativo de estructura `case` en el script `agenda.sh`:

```
…
DATOS="/$HOME/agenda.txt"
OPCIONES_MENU="Ver Alta Baja Modificacion Salir"
PS3="Selecciona una Opcion"
…
select OPCION in $OPCIONES_MENU
do
  case $OPCION in
    Ver)
…
        ;;
    Alta)
```

```
…
      ;;
    Baja)
…
      ;;

    Modificacion)
…
      ;;

    Salir)
…
      ;;
  esac
done
…
```

En este caso la opción es leída a través de la línea de comandos mediante una sentencia `read`.

4.5. Bucles

Con las estructuras condicionales podemos tomar decisiones, pero también tenemos la necesidad en ocasiones de ejecutar tareas repetitivas. Es en ese caso cuando debemos utilizar los bucles. Bash [29] nos ofrece varias estructuras que nos permiten hacer esto.

4.5.1 `while` y `until`

La sintaxis de estos bucles es la siguiente:

```
while                                   until
      'lista de comandos A'                   'lista de comandos A'
do                                      do
      'lista de comandos B'                   'lista de comandos B'
done                                    done
```

En el caso de `while` se ejecuta la lista de comandos A, y si el último comando de dicha lista devuelve un valor de 0 (verdad) entonces se ejecuta la lista de comandos B, y cuando esta acaba, vuelve a ejecutarse la lista de comandos A. En caso contrario, se ejecutan los comandos que van a continuación de la palabra reservada 'done'. En el caso de `until` el comportamiento es idéntico, pero sólo se ejecuta la lista de comandos B, si el último comando de la lista A devuelve un valor diferente de 0 (falso). Generalmente la lista de comandos A no es más que una evaluación de tipo test, o un

comando simple. En estos casos, es indiferente usar `while` o `until` ya que basta con negar la condición de evaluación. Podemos ver en el siguiente ejemplo que el resultado final es el mismo:

```
$X=1
$while (( X <= 8 ))
do
      echo X=$X
      (( X = X + 1 ))
done
```

```
X=1
X=2
X=3
X=4
X=5
X=6
X=7
X=8
```

```
$X=1
      $until (( X > 8 ))
do
      echo X=$X
      (( X = X + 1 ))
done
```

```
X=1
X=2
X=3
X=4
X=5
X=6
X=7
X=8
```

Podemos ver un ejemplo práctico similar al anterior en el script `copia_array_remoto.sh`:

```
…
typeset -Z2 NUM
NUM=1
while (( NUM <= MAQUINAS ))
do
        HOST[$NUM]=$SALA$NUM
        (( NUM = NUM + 1 ))
done
…
```

En este script se asume que los ordenadores de cada sala tienen como nombre el nombre de sala y un número ordinal de dos dígitos. Para generar una lista con los nombres de hosts previamente hemos almacenado en la variable `SALA` el nombre de la sala y en la variable `MAQUINAS` el número de ordenadores que tiene. Con este bucle, generamos un array con los nombres de todos los ordenadores, que después se usará para hacer la copia remota.

4.5.2 `for`

En Bash [29] existen dos formas de bucle `for` cuyas sintaxis son:

```
for VARIABLE [ in lista ]
do
      'lista de comandos'
done
```

```
for (( expr1 ; expr2 ; expr3 ))
do
            'lista de comandos'
done
```

En la primera forma, `lista de comandos` es ejecutada tantas veces como valores haya en `lista`. Por cada ejecución, un valor de `lista` es asignado a `VARIABLE`. Si `lista` no es facilitada, `variable` es asignada a cada uno de los parámetros posicionales pasados al script. Existen muchas formas de generar `lista`. Puede ser una lista de valores explícitos, pueden ser generados a su vez por la ejecución de un comando, o pueden ser generados mediante patrones que se expandan a nombres de ficheros entre otras. En el script `copia_array_remoto.sh` vemos cómo la lista de valores es facilitada con un array que contiene el nombre de los hosts a los que copiar:

```
…
for i in ${HOST[*]}
do
       rcp $FICHERO $i:$DIR/$FICHERO && echo "Copia correcta en $i"
done
…
```

En el script `minusculas.sh` vemos un ejemplo en el que la lista es generada por un patrón que se expande a los nombres de ficheros del directorio en el que queremos hacer el cambio:

```
…
for FICHERO in $DIRECTORIO/*
do
…
done
…
```

Un problema que tiene esta forma de bucle `for` es que no podemos dar una lista de elementos numéricos como valor de principio, incremento y fin, por ello, en el ejemplo del script `escanea_ip.sh` que fue originalmente diseñado para `ksh`, como ya comentamos anteriormente, vemos una forma ingeniosa de generar una lista de números de 1 a 254 sin tener que expresarla explícitamente:

```
…
for i in $( X=1 ; while (( X <= 254 )) ; do echo $X ; (( X = X + 1 ))
; done )
do
…
done
…
```

Como vemos, la lista es generada a su vez por la ejecución de un bucle `while` cuyos valores son convertidos a una cadena que se pasa al bucle `for`. Esto es necesario en la Shell Korn v88, donde no existe la segunda forma de bucle `for` que hemos apuntado al principio. Esto ya fue comentado cuando hablábamos de expresiones aritméticas y lógicas. En el caso de Bash [29] y de Korn v93, bastaría con haberlo expresado de la forma que aparece en `escanea_ip_2.sh`:

```
…
for (( i=1 ; i <= 254 ; i = i + 1 ))
do
…
done
…
```

Las tres expresiones se evalúan según la sintaxis de expresiones aritméticas indicadas en el punto 3.1.6. La primera expresión sólo es evaluada al principio del bucle. Después, en cada repetición, se evalúa la segunda expresión, y mientras esté retornando valores diferentes de 0 (falso) se ejecutará la tercera expresión y se repetirá el bucle de comandos entre `do` y `done`.

4.5.3 `select`

La sentencia `select` implementa una forma particular de bucle que combina un menú de opciones, la petición de una opción y la acción repetitiva de comandos. La sintaxis es la siguiente:

```
select VARIABLE [ in lista ]
do
      'lista de comandos'
done
```

Select muestra los valores facilitados en `lista` como un menú de opciones cada una de ellas precedida de un número en la salida de errores estándar. A continuación, el valor de la variable de entorno `PS3` es mostrado, y se espera a que el usuario escriba uno de los números de las opciones. Dependiendo del número usado, `VARIABLE` será asignada a la opción correspondiente y se ejecutará `lista de comandos`.

Una vez terminada dicha ejecución, se volverá al menú de opciones. La ejecución del bucle finaliza al pulsar `EOF` (`crtl + d`) o al encontrarse en el bucle con una sentencia `break` o `exit.`

Generalmente, la lista de comandos del interior del bucle suele ser una estructura `case` que analiza cada uno de los valores posibles del menú. Podemos ver un ejemplo de ello en el script `agenda.sh`:

```
…
OPCIONES_MENU="Ver Alta Baja Modificacion Salir"
PS3="Selecciona una Opcion"

select OPCION in $OPCIONES_MENU; do
  case $OPCION in
    Ver)
      …
        ;;
    Alta)
      …
      ;;
    Baja)
      …
      ;;
    Modificacion)
      …
      ;;
    Salir)
      …
      exit 0
  esac
done
…
```

En este caso, dependiendo de la opción seleccionada, se hará una acción u otra sobre la agenda del usuario. Sólo se sale del script cuando se pulsa la opción correspondiente a `Salir`.

4.5.4 `break` y `continue`

Si queremos romper de forma fortuita la ejecución de un bucle sin acabar completamente el script, no podemos usar la sentencia `exit`. En esos casos tenemos dos opciones cuya sintaxis es la siguiente:

```
while                                   while
      …                                       …
do                                      do
      …                                       …
      break                                         continue
      …                                       …
done                                    done
```

Aunque se han indicado para un bucle `while`, ambas sentencias son válidas en cualquier tipo de bucle. La sentencia `break` rompe completamente la ejecución del bucle, y sigue ejecutando los comandos que están a continuación de la sentencia `done`. En el caso de `continue`, sólo rompe la ejecución actual del bucle, y vuelve a ejecutar los comandos existentes desde el principio, en este caso, debajo de la sentencia `while`.

Por ser sentencias que rompen todas la lógica de la programación estructurada, no es recomendable el uso de éstas, salvo en casos donde es absolutamente necesario como por ejemplo el de las sentencias `select`.

4.6. Funciones

4.6.1 Definición de Funciones

Las funciones son agrupaciones de código que son ejecutadas dentro de la shell actual. Son útiles cuando tenemos una cantidad de instrucciones que queremos que sean invocadas en varios puntos de nuestro script de forma idéntica y repetida, en ese caso, son la clave para ahorrar la reescritura de código.

En términos generales, la sintaxis de una función es la siguiente:

```
function nombre_de_funcion
{
      comandos
      …
}
```

ó bien

```
nombre_de_funcion ()
{
      comandos
      …
}
```

En ambos casos, a partir del punto donde la función queda definida, puede ser ejecutada en la instancia de la shell o del script actual. En caso de terminar dicha shell, la función debe ser de nuevo definida, por lo que es habitual hacerlo al principio de los scripts, o de los ficheros de perfil de inicio de sesión de las shells.

Para ver la lista de funciones que tiene la shell definida podemos hacerlo con el comando `typeset`

```
$typeset -f
funcion_1 ()
{
 …
}
…
```

Esto nos mostrará dicha lista. Con la opción `-F` obtendríamos sólo la lista de nombres de funciones sin el código que contienen.

4.6.2 Argumentos y Entorno de Funciones

Las funciones están definidas en el espacio de memoria de la shell y son ejecutadas en su ámbito. Esto implica que cualquier definición de variables o entorno que se haga dentro de una función, será visible fuera de ella. Las únicas variables que sólo son visibles en el espacio local de una función son los argumentos posicionales, cuyo tratamiento es idéntico al de los scripts de la shell. El siguiente ejemplo que puede ser ejecutado en una shell lo demuestra:

```
$function prueba
```

```
 {
      echo $1
      VISIBLE='hola'
      echo $VISIBLE
 }
$
$prueba adios
 adios
 hola
$
$echo $1
 (No se muestra nada, el argumento posicional no está definido fuera)
$echo $VISIBLE
 hola
```

4.6.3 Finalización de Funciones

Por el mismo motivo expresado anteriormente, si dentro de una función usamos la sentencia `exit`, esto acabaría la ejecución completa de la shell, y no sólo de la función. Para terminar una función sin acabar la shell donde está definida, tenemos la sentencia `return` cuyo uso para pasar valores de retorno es idéntico al explicado anteriormente para `exit`. Podemos verlo en el siguiente ejemplo:

```
$function termina
 {
      echo 'Esta función termina con un valor de retorno de 35'
      return 35
 }
$
$termina
 Esta función termina con un valor de retorno de 35
$echo $?
 35
```

4.6.4 Librerías de Funciones

Esta es una característica no implementada en Bash, pero sí en la Shell Korn [28], y por su uso e importancia merece una mención. Consiste en mantener un directorio donde cada fichero contenga la definición de una única función. El fichero a su vez debe tener el mismo nombre que la propia función. Además de esto debemos tener una variable de entorno `FPATH` que contenga la lista de dichos directorios:

```
$ echo $FPATH
 /usr/functions:/home/pedrolopezs/funciones
$ ls -l /home/pedrolopezs/funciones
 -rw-------   1 root       sys            55 Apr  5 11:19 funcion1
 -rw-------   1 root       sys             74 Apr  4 18:27 funcion2
 -rw-------   1 root       sys             49 Apr  4 18:27 funcion3
$ cat /home/pedrolopezs/funciones/funcion1
 function funcion1
 {
  …
 }
$
```

Cumpliendo estos requisitos, cada vez que se invoque una función que aún no esté definida en la shell, ésta será buscada a lo largo de los directorios que aparecen en la variable `FPATH` y si es encontrada, será definida en la shell y ejecutada, quedando cargada en el espacio de memoria de ésta hasta que acabe. Esta técnica permite al administrador ahorrarse la reescritura de código para ser usado en varios scripts.

Podemos ver ejemplos de usos de funciones en el script `maneja_rpm.sh` para cada una de las funcionalidades permitidas en él, y en el script `ftpremoto.sh` para definir una función que muestre un cursor giratorio mientras se realiza la operación remota

4.7. Flujo de Datos. Entrada y Salida

En el punto 3.1.9 comentamos cómo los comandos en entornos UNIX pueden comunicarse *conectando* su entrada o salida de datos mediante redirectores y tuberías *pipes*. También es posible asignar de forma permanente un descriptor de entrada o salida para manejar un fichero dentro de un script. Podemos ver un ejemplo de redirección a ficheros temporales en el script `cortafichero.sh`:

```
…
cut -c $PRINCIPIO-$FINAL $FICHERO > /tmp/$FICHERO.$CONTADOR.$$
…
paste -d: /tmp/$FICHERO.*.$$ > $FICHERO.cortado
…
```

En el script `ftpremoto.sh` vemos un ejemplo de estructura *here document*, ya descrita en la tabla 3.4. que se utiliza para insertar en el propio código los comandos que se lanzarán contra el servidor ftp:

```
…
ftp $1 > /dev/null <<COMANDOS
prompt n
hash
ascii
cd scripts
mget *
bye
COMANDOS
…
```

También podemos ver un ejemplo de *here document* para un ftp en el script mailbomb.sh:

```
…
telnet smtp.terra.es 25 <<COMANDOS > /dev/null 2>&1
helo origen.es
mail from: <prueba@origen.es>
rcpt to: <pringaillo1@terra.es>
data
From: prueba@origen.es
To: pringaillo1@terra.es
Subject: mensaje de prueba
Hola
.
quit
COMANDOS
…
```

4.8. Traps y Señales

Un proceso puede ser controlado mediante señales. Estas señales indican al proceso de forma no determinista (en cualquier momento durante la ejecución) que deben realizar una acción, normalmente relacionada con finalizar la ejecución, interrumpirla o reanudarla. Las señales pueden ser enviadas explícitamente a un proceso mediante el comando `kill`, pero debido a que éste está en primer plano (*foreground*) no podemos

teclear dicho comando. También existen combinaciones de tecla equivalentes a determinadas señales. Algunas de las señales más habituales podemos verlas en la Tabla 4.2.:

Señal	Nombre	Combinación	Descripción
0	EOF	Ctrl+d	No es exactamente una señal, pero equivale a fin de sesión
1	HUP		El proceso debe reiniciarse sin interrumpir su ejecución
2	INT	Ctrl+c	El proceso debe interrumpirse
3	QUIT	Ctrl+\	El proceso debe interrumpirse generando un core (volcado de memoria)
9	KILL		El proceso debe terminar inmediatamente (Parada fortuita)
15	TERM		El proceso debe terminar cuando pueda (Parada ordenada)
18	CONT		El proceso debe continuar en el punto que hizo stop
19	STOP	Ctrl+z	El proceso debe hacer stop (pausa en la ejecución)

Tabla 4.2: Señales para control de procesos.

Las señales 18 y 19 corresponden a las 26 y 24 respectivamente en HP-UX. El resto de las señales suelen ser iguales en todos los entornos Unix.

Durante la ejecución de un script puede darse el caso de recibir una de estas combinaciones de teclas o un comando kill desde otra sesión por parte del usuario, con intención de interrumpir el proceso del script. Si esto pasa, tenemos la posibilidad de controlar la secuencia de acciones a realizar una vez recibida la señal correspondiente. Esto lo haremos con el comando `trap`. La sintaxis de `trap` es la siguiente:

`trap` *`señal1 señal2 ... señaln 'lista de comandos'`*

Ejecuta los comandos entre comillas al recibir una de las señales

`trap` *`señal1 señal2 ... señaln`* `''`

Ignora las señales no realizando ninguna acción al recibirlas (ni siquiera parar)

`trap` *`señal1 señal2 ... señaln`*

Las señales vuelven a tener su comportamiento habitual (acción por defecto)

Con este comando podríamos por ejemplo blindar un trozo de código que no queremos que sea interrumpido por parte del usuario. Esto se puede ver en el fichero `/etc/profile` de HP-UX. Dicho fichero es ejecutado en el inicio de sesión de cualquier usuario, y no hace caso de las combinaciones `ctrl+c` o `ctrl+\`:

```
…
trap 1 2 3 15 ''
…
…     (Zona de código blindado a estas cuatro señales)
…
trap 1 2 3 15
…
```

También podemos ver esto en el script `login.sh` para evitar que el usuario rompa la ejecución del script.

Otra de las utilidades típicas del uso de traps es el de realizar una acción antes de acabar la ejecución del script en caso de que se lance por ejemplo una combinación `CTRL+C` para interrumpirlo. Por ejemplo, podríamos borrar ficheros temporales generados durante la ejecución. Esto debería declararse al principio del script para que tenga vigencia a lo largo de toda su ejecución. Podemos ver un ejemplo en el script `ftpremoto.sh`. Se evita que el fichero `.netrc` se quede en el directorio del usuario sin ser borrado en caso de cortar la ejecución del script con `CTRL+C` por ejemplo:

```
…
trap "rm $HOME/.netrc ; exit 1" INT QUIT TERM
…
```

También podemos ver un ejemplo en el script `login.sh` para evitar que el cursor quede invisible si se interrumpe la ejecución del script:

```
…
# PONE EL CURSOR INVISIBLE
tput civis
# CUANDO EL PROCESO ACABE, PONE DE NUEVO EL CURSOR NORMAL
trap 'tput cnorm; exit' INT TERM
…
```

Nótese en los ejemplos anteriores que dentro de los comandos debemos poner explícitamente el comando `exit` de lo contrario, la sentencia `trap` devolvería el control de la ejecución al punto donde nos quedamos.

Existe también una pseudo-señal `ERR` que puede usarse para el tratamiento de errores. Cuando un comando termina con error, si existe un `trap` con dicha señal ejecutarán los comandos correspondientes:

```
…
trap ERR 'echo Se ha producido un error en la ejecución ; exit'
```

CAPÍTULO 5. EXPRESIONES REGULARES, SED Y AWK

5.1. Introducción

Aunque sed y awk son implementaciones independientes del código de las diferentes shells que hemos estudiado, su uso está plenamente extendido como parte propia de muchos de los script que se utilizan en administración de sistemas. La razón viene dada por la flexibilidad que ambas herramientas aportan para realizar filtros y procesamiento de ficheros de texto. Es por ello que dedicaremos un punto especial a estudiar las principales claves de dichas herramientas.

Tanto sed como awk hacen uso para el análisis de cadenas de las expresiones regulares, por lo que antes de pasar a dichas herramientas debemos hacer un repaso a las reglas que soportan dichas expresiones.

5.2. Expresiones regulares. grep

Las expresiones regulares (e.r.) se utilizan para buscar secuencias de texto en una cadena que sigan una serie de reglas. Para ello se utilizan una serie de caracteres con significado especial. Una expresión regular puede contener por lo tanto:

- Caracteres literales: `abce`
- Metacaracteres: `^ $ . * [] () \`
- Combinaciones de ambos: `^abcd$`

El significado de los Metacaracteres lo tenemos en la Tabla 5.1

Metacarácter	Significado	Ejemplo
^	Hace referencia a que la e.r. debe estar al principio de la línea	e.r.: ^hola.* hola hola que tal hola ana …
$	Hace referencia a que la e.r. debe estar al final de la línea	e.r.: .*fin$ fin esto es el fin llegó el fin …
.	Es el carácter comodín y equivale a cualquier carácter individual excepto la cadena vacía.	e.r.: . a b / …
*	Equivale a la cadena vacía o cualquier repetición una o más veces de la e.r. que le precede.	e.r.: a* Ø a aa aaa …
[]	Establece una categoría de caracteres, donde el resultado es uno cualquiera de ellos. Usando el guión – tenemos como resultado la categoría que va del carácter a la izquierda del guión hasta el carácter a la derecha de este. Usando ^ al principio, invertimos la categoría.	e.r.: [a-f] a b … F
()	Establece los límites de una e.r. de esta forma podemos usar otros metacaracteres sin que afecten globalmente a toda la e.r. Si usamos (e.r.1 | e.r.2 | …) equivale a una cualquiera de las e.r. expresadas.	e.r.: a(.*) a ab a/ …
\	En realidad este carácter no tiene significado como expresión regular, pero se usa para convertir en literal a cualquiera de los caracteres anteriores.	e.r.: \.. .a .b …

Tabla 5.1 Significado de los metacaracteres en las expresiones regulares

Las expresiones regulares se utilizarán tanto en sed como en awk para buscar cadenas de caracteres que coincidan con dichas expresiones y procesarlas posteriormente según la sintaxis que hayamos usado en el comando. Un comando que también hace uso de las expresiones regulares y que también tiene mucha utilidad en scripts es grep. Este comando busca en un fichero la expresión regular que le pasamos y extrae aquellas líneas que cumplen dicha expresión regular. La sintaxis es la siguiente:

```
grep expresión-regular fichero
```

Podemos ver unos ejemplos usando el fichero /etc/group. En este fichero, la palabra root aparece en diferentes líneas y diferentes posiciones (principio de línea, final de línea…). El fichero tiene el siguiente contenido en un equipo HP-UX:

```
$ cat /etc/group
 root::0:root
 other::1:root,hpdb
 bin::2:root,bin
 sys::3:root,uucp
 adm::4:root,adm
 daemon::5:root,daemon
 mail::6:root
 lp::7:root,lp
 tty::10:
 nuucp::11:nuucp
 users::20:root
 nogroup:*:-2:
 smbnull::101:
 mysql::102:
```

Algunos ejemplos de grep con expresiones regulares aplicadas a este fichero:

```
$ grep 'root' /etc/group      #La palabra root en cualquier posicion
 root::0:root
 other::1:root,hpdb
 bin::2:root,bin
 sys::3:root,uucp
 adm::4:root,adm
 daemon::5:root,daemon
 mail::6:root
 lp::7:root,lp
 users::20:root
$
$ grep '^root' /etc/group     # La palabra root al principio de linea
 root::0:root
$
$ grep 'root$' /etc/group     # La palabra root al final de linea
 root::0:root
 mail::6:root
```

```
 users::20:root
$
$ grep '^d.*n$' /etc/group    # Lineas que empiecen en d y acaben en n
 daemon::5:root,daemon
$
$ grep 'a.m' /etc/group       #cualquier a seguida de otro caracter y m
 adm::4:root,adm
 daemon::5:root,daemon
$
$ grep 'a[aeiou]m' /etc/group       # cualquier a seguida de vocal y m
 daemon::5:root,daemon
```

5.3. Comando sed

El comando sed se utiliza principalmente para hacer substitución de cadenas, aunque hay muchos otros comandos que puede realizar. La sintaxis genérica es:

```
sed –opciones 'comando sed' fichero(s) → stdout
```

O bien también podemos lanzar múltiples comandos en una sola línea:

```
sed –opciones –e 'comando sed' –e 'comando sed' fichero(s) → stdout
```

O como un programa en un fichero de comandos:

```
sed –opciones –f fichero-de-comandos fichero(s) → stdout
```

El comportamiento estándar de sed es el de admitir un fichero o varios como argumentos, o en su defecto, leer la entrada estándar *stdin* y lanzar por salida estándar *sdtout* el resultado de los comandos sed que hayamos programado [7].

```
stdin → sed 'comando sed' → stdout
```

Hay que tener en cuenta que sed nunca trabaja directamente sobre el fichero de entrada, por lo que para tener una salida permanente, ésta debe ser redirigida hacia otro fichero.

Los principales comandos sed que podemos usar son s, d, p, r, w (sustituir, borrar, imprimir, leer, escribir). Cada comando a su vez, puede llevar un ámbito de acción que podemos ver resumido en la Tabla 5.2 [7]:

Ambito	Interpretación	Ejemplo
Línea	El comando sed sólo actúa en la línea linea	sed '4d' file (borra la linea 4 del fichero)
Linea1,linea2	El comando sólo actúa entre las líneas linea1 y linea2 ambas incluidas	sed '4,7s/hola/adios/g' file (cambia la palabra hola por adios desde la línea 4 a la 7 inclusive)
Linea,$	El comando actúa desde la línea linea1 hasta el final del fichero	sed '6,$d' file (borra desde la línea 6 al final del fichero)
/e. regular/	El comando sólo actúa en aquellas líneas que tengan una cadena que coincida con la expresión regular e.regular	sed '/Madrid/d' file (borra las líneas donde aparece la palabra Madrid)
/e.regular1/,/e.regular2/	El comando busca la primera linea que contenga una cadena que coincida con e.regular1 y empieza a actuar hasta encontrar una línea que contenga otra cadena coincidente con e.regular2. En ese momento deja de actuar y sigue buscando de nuevo e.regular1 para volver a actuar.	sed -n '/capitulo/,/fin/p' file (imprime las líneas que existen entre aquellas que contienen la palabra capítulo y la palabra fin. La Opción -n ignora el resto de las líneas y es necesario en los comandos de tipo p.)

Tabla 5.2: Ámbito de actuación de los comandos sed.

Veremos a continuación cómo funciona cada uno de los comandos sed que hemos citado:

5.3.1 sustituir

La sintaxis para este comando es la siguiente [7]:

```
sed 'ambito s/e.regular/cadena/[ng]' fichero(s)
```

El comando `s` substituye en las líneas de ámbito indicadas, las cadenas que coincidan con la `e.regular`, por `cadena` que se pasa como segundo argumento. La opción final `g` indica que deben sustituirse todas las ocurrencias de la línea, si ponemos un número n en su lugar, sólo substituye la ocurrencia número *n*. Por defecto, si no se indica esta opción final, se asume que se substituye sólo la ocurrencia número `1`. Por ejemplo, en el fichero `/etc/group` cuyo contenido vimos anteriormente, podríamos hacer las siguientes substituciones.

En el primer caso substituimos la expresión regular `root` por `pedro`, sólo entre las líneas 3 a la 6. Esto se hace en todas las ocurrencias de la línea gracias a la `g` del final del comando:

```
$ sed '3,6s/root/pedro/g' /etc/group
 root::0:root
 other::1:root,hpdb
 bin::2:pedro,bin
 sys::3:pedro,uucp
 adm::4:pedro,adm
 daemon::5:pedro,daemon
 mail::6:root
 lp::7:root,lp
 tty::10:
 nuucp::11:nuucp
 users::20:root
 nogroup:*:-2:
 smbnull::101:
 mysql::102:
```

En este segundo ejemplo substituimos la expresión regular `root` por `pedro`, sólo en aquellas líneas donde aparezca la expresión regular `bin`:

```
$ sed '/bin/s/root/pedro/g' /etc/group
 root::0:root
 other::1:root,hpdb
 bin::2:pedro,bin
 sys::3:root,uucp
 adm::4:root,adm
 daemon::5:root,daemon
 mail::6:root
 lp::7:root,lp
 tty::10:
 nuucp::11:nuucp
 users::20:root
 nogroup:*:-2:
 smbnull::101:
 mysql::102:
```

La expresión regular puede ponerse entera en la cadena a sustituir usando el carácter especial `&`. En este caso, se añade el texto a la expresión encontrada.

En este último ejemplo, añadimos detrás de la expresión regular `root`, la cadena `pedro`. Se consigue por el carácter especial `&` que apunta a la expresión regular original. Esto se hará en todo el fichero, por el ámbito `1,$,` pero sólo se sustituirá la primera ocurrencia ya que no se ha puesto `g` al final del comando:

```
$ sed '1,$s/root/&pedro/' /etc/group
 rootpedro::0:root      #En esta linea solo se hace una  substitución
 other::1:rootpedro,hpdb
 bin::2:rootpedro,bin
 sys::3:rootpedro,uucp
 adm::4:rootpedro,adm
 daemon::5:rootpedro,daemon
 mail::6:rootpedro
 lp::7:rootpedro,lp
 tty::10:
 nuucp::11:nuucp
 users::20:rootpedro
 nogroup:*:-2:
 smbnull::101:
 mysql::102:
```

5.3.2 Borrar

La sintaxis de este comando es [7]:

```
sed 'ambito d' fichero(s)
```

El comando `d` eliminará las líneas que se encuentren en el ámbito indicado, cada uno de los ficheros. En el siguiente ejemplo, se borran las líneas donde aparece la expresión regular `root`:

```
$ sed '/root/d' /etc/group
 tty::10:
 nuucp::11:nuucp
 nogroup:*:-2:
 smbnull::101:
 mysql::102:
```

Podemos encontrar un ejemplo concreto de uso de este comando en el script `lj-ps-pdf.sh` donde se usa para eliminar las tres primeras líneas de un fichero:

```
for i in *.lj
do
# EL COMANDO BASENAME CORTA UNA CADENA HASTA LA SECUENCIA INDICADA
 nombre=$(basename $i .lj).ps
 echo -n "convirtiendo $i en $nombre... "
# SE ELIMINAN LAS TRES PRIMERAS LINEAS CON UN COMANDO SED
 sed '1,3d' $i > $nombre
 echo "hecho"
done
```

5.3.3 Imprimir

La sintaxis es la siguiente [7]:

```
sed –n 'ambito p' fichero(s)
```

El comando `p` imprime las líneas indicadas en el ámbito para cada uno de los ficheros. La opción `–n` debe ponerse ya que si no se hace, cada línea que cumpla el ámbito es impresa dos veces y el resto de las líneas también serían presentadas. Esto es debido a que el comportamiento normal de `sed` es imprimir literalmente aquellas líneas que no cumplan el ámbito, pero eso es algo que en este caso concreto no nos interesa.

En este ejemplo, se imprimen sólo aquellas líneas donde aparece la expresión regular `root`:

```
$ sed -n '/root/p' /etc/group
 root::0:root
 other::1:root,hpdb
 bin::2:root,bin
 sys::3:root,uucp
 adm::4:root,adm
 daemon::5:root,daemon
 mail::6:root
 lp::7:root,lp
 users::20:root
```

5.3.4 Leer

La sintaxis para este comando es [7]:

```
sed 'ambito r fichero-r' fichero(s)
```

El comando `r` inserta el contenido de *`fichero-r`* debajo de cada línea que entre dentro del ámbito expresado, en cada uno de los ficheros. En el siguiente ejemplo, se busca las líneas donde exista una d, cualquier secuencia de caracteres incluida la cadena vacía, y luego una m. A continuación se inserta el fichero que se ha creado previamente después de cada línea con esa condición:

```
$ echo "ESTA LINEA CONTIENE LA EXPRESION adm" > /tmp/file1
$
$ sed '/adm/r /tmp/file1' /etc/group
 root::0:root
 other::1:root,hpdb
 bin::2:root,bin
 sys::3:root,uucp
 adm::4:root,adm
 ESTA LINEA CONTIENE LA EXPRESION adm
 daemon::5:root,daemon
 mail::6:root
 lp::7:root,lp
 tty::10:
 nuucp::11:nuucp
 users::20:root
 nogroup:*:-2:
 smbnull::101:
 mysql::102:
```

5.3.5 Escribir

La sintaxis es la siguiente [7]:

```
sed –n 'ambito w fichero-w' fichero(s)
```

El comando `w` escribe en *`fichero-w`* las líneas que entran dentro del ámbito en cada uno de los ficheros. También se usa en este caso la opción `–n` para que no genere salida por pantalla ya que el objetivo del comando es el de generar un fichero. En el siguiente ejemplo, se genera el fichero /tmp/rootgrp que contiene sólo aquellas líneas donde aparece la expresión regular root:

```
$ sed –n '/root/w /tmp/rootgrp' /etc/group
$ cat /tmp/rootgrp
 root::0:root
 other::1:root,hpdb
 bin::2:root,bin
 sys::3:root,uucp
 adm::4:root,adm
 daemon::5:root,daemon
 mail::6:root
 lp::7:root,lp
 users::20:root
```

5.4. Comando awk

El comando awk tiene un comportamiento similar al de sed en cuanto a que admite como argumento un fichero, o en su defecto la entrada estándar, lo procesa a través de un programa awk, y finalmente produce una salida estándar que puede ser redirigida a otro fichero. Las principales formas de sintaxis son [6,7]:

```
stdin → awk 'programa awk' → stdout

awk –f fichero-de-programa fichero(s)

awk 'programa awk' fichero(s) → stdout
```

La forma de procesar la entrada es diferente a sed. De hecho se considera a awk prácticamente un lenguaje de programación basado en C. Es capaz de utilizar funciones y estructuras de control de flujo similares a las de dicho lenguaje. En esta exposición nos centraremos sólo en los modos básicos de funcionamiento de awk que son los más utilizados dentro de los scripts. Para ello, nos basaremos en un ejemplo sencillo pero que representa todas las características citadas. El script completo formatea.awk se encuentra en el capítulo 6. Este script admite un fichero de entrada con el siguiente formato:

```
# cat /tmp/listado.txt
 Pedro Fernandez Perez Madrid 23000 8000
 Jaime Martin Leal Malaga 24500 6000
 Juana Luque Garcia Malaga 23200 4500
 Antonio Bernal Mendez Madrid 24500 3200
 Maria Luque Moran Madrid 30000 8000
 Juan Atienza Martinez Malaga 24000 4000
 Antonio Lopez Marques Malaga 23500 3000
```

Y genera una salida formateada en la que asume que la penúltima columna es el salario de cada persona y la última los beneficios obtenidos en el año. La salida sería de la siguiente forma:

```
# ./formatea.awk listado.txt
 Nombre     Apellido1  Apellido2   Salario Beneficios

 Pedro      Fernandez  Perez         23000       8000
 Jaime      Martin     Leal          24500       6000
 Juana      Luque      Garcia        23200       4500
 Antonio    Bernal     Mendez        24500       3200
 Maria      Luque      Moran         30000       8000
 Juan       Atienza    Martinez      24000       4000
 Antonio    Lopez      Marques       23500       3000
 ------------------------------------------------
 Total Salarios: 172700
 Total Beneficios
 Madrid: 19200 Malaga: 17500
```

5.4.1 Procesamiento de líneas en `awk`

En general el programa `awk` leerá línea a línea la entrada del fichero argumento o la entrada estándar. Cada línea será procesada por todas las acciones de que consta el programa, aunque ciertas acciones pueden ir condicionadas por una expresión regular o patrón que hará que dicha acción procese determinadas líneas del fichero de entrada o no.

Las acciones incluidas en el programa pueden constar de uno o más de los siguientes elementos [6,7]:

patron {*acción*}	El patrón es habitualmente una expresión regular pero puede adoptar otras formas como ahora veremos.
/e.r./ {*acción*}	La acción procesará sólo aquellas líneas que contengan la expresión regular.
{*acción ; acción*}	Podemos incluir varias acciones consecutivas en la misma línea condicionada o no, por un patrón.
{*acción* *acción*}	Las acciones se pueden poner en diferentes líneas físicas sin necesidad del carácter ;

5.4.2 Campos definidos dentro de una línea

Cada línea de entrada es procesada a modo de registro. El separador de campo por defecto es cualquier combinación de espacios y tabuladores. Cada campo es designado mediante una variable de la siguiente forma [6,7]:

Línea: `en un lugar de la mancha`

Variables:
```
$0: en un lugar de la mancha
$1: en
$2: un
$3: lugar
$4: de
$5: la
$6: mancha
NF: 6
NR: 1
```

Cada variable `$n`, toma como valor una palabra de la línea, la variable `NF` apunta al número total de campos en cada línea y la variable NR contiene el número de registro actual. `$0` contiene la línea completa. Estas variables son las más utilizadas posteriormente en las acciones, sobre todo para los comandos `print` y `printf`.

También se pueden usar estas variables como parte de un patrón para condicionar una línea de acciones. Por ejemplo en el siguiente caso:

```
$4 > 5 { print $3 }
```

En este caso la acción es imprimir el tercer campo, y esto sólo se hace si el cuarto campo tiene un valor mayor que cinco. Podemos ver un ejemplo concreto en el script `formatea.awk`. El principio de las líneas principales de procesado es:

```
$4 ~ /Madrid/ {printf …
$4 ~ /Malaga/ {printf …
```

En este caso se procesa la línea si el campo `$4` coincide con la expresión regular `Madrid` o `Malaga`. En el siguiente punto veremos todos los operadores que se pueden usar.

5.4.3 Operadores en los patrones

Los patrones usados para condicionar las líneas con acciones pueden usar también operadores de expresiones regulares, relacionales o aritméticas, como el del ejemplo anterior. Los operadores que podemos usar en estos patrones son [6,7]:

Operador		Ejemplo
Expresiones Regulares:		
~	Coincide con la expresión regular	`$1 ~ /Madrid/ { acción }`
! ~	No Coincide con la expresión regular	`$2 ! ~ /Malaga/ { acción }`
Expresiones Aritméticas:		
==	Igual que	`$2 == 5 { acción }`
!=	No igual que	`NF != 10 { acción }`
>=	Mayor o igual que	`$4 >= 10 { acción }`
<=	Menor o igual que	`$4 <= 10 { acción }`
>	Mayor que	`$4 > 10 { acción }`
<	Menor que	`$4 < 10 { acción }`
Conjunción de operaciones:		
!	Not	`! $3 < 4 { acción }`
&&	And	`$3 ~ /Madrid/ && $2 < 3 { acción }`
||	Or	`$3 ~ /Te/ || $3 ~ /Cafe/ { acción }`

5.4.4 Procesamiento pre-entrada y post-entrada

Los patrones `BEGIN` y `END` tienen significados especiales. Todas las líneas precedidas por `BEGIN` son procesadas antes de las líneas de cada fichero. Todas las líneas precedidas por `END` son procesadas al final del proceso completo de cada fichero. En el caso de que sean varios los ficheros de argumentos de entrada, este proceso se hace por cada uno de ellos [6,7]:

```
BEGIN { acciones previas al procesamiento de la entrada }
patron { acciones durante el procesamiento de la entrada}
END { acciones posteriores al procesamiento de la entrada }
```

Podemos ver ejemplos concretos en el script `formatea.awk` para imprimir la cabecera y el pie del informe:

```
BEGIN {print "Nombre     Apellido1  Apellido2   Salario Beneficios"}
BEGIN {print "" ; TSAL=0 ; BENMD=0 ; BENMA=0 }
…
END {print "-------------------------------------------------"}
END {print "Total Salarios: " TSAL }
END {print "Total Beneficios"}
END {print "Madrid: " BENMD " Malaga: " BENMA }
```

5.4.5 Impresión de valores con `awk`. Salida formateada

Los comandos que se usan habitualmente en `awk` para mostrar información son `print` y `printf`. El primero funciona de forma similar al comando `echo` de unix, permitiendo mostrar variables y valores literales. El segundo produce una salida formateada y la sintaxis es muy similar a la de `printf` del lenguaje `C`.

Podemos ver ejemplos de ambos comandos en el script `formatea.awk`:

```
BEGIN {print "Nombre     Apellido1  Apellido2   Salario Beneficios"}
…
$4 ~ /Madrid/ {printf "%-11s%-11s%-11s%8d%11d\n",$1,$2,$3,$5,$6
TSAL+=$5
BENMD+=$6}
…
END {print "Total Salarios: " TSAL }
```

En el caso del comando `printf` indicamos primero una cadena que actúa como patrón, donde se especifica la posición y el formato (número de caracteres y alineación) de los argumentos que vamos a presentar. En el ejemplo anterior, `printf` imprime cinco argumentos, tres son cadenas, y dos decimales. Siempre en dicha cadena debemos indicar `\n` en el caso de que queramos que se haga un salto de línea.

5.4.6 Variables

Dentro de un script awk podemos utilizar tanto las variables predefinidas que citamos en el punto 5.4.2, como variables definidas según nuestras necesidades. No es necesario declarar dichas variables, simplemente inicializarlas [6,7].

En el script `formatea.awk` podemos ver cómo se usan tres variables para sumar el total de los salarios que tiene cada persona y los totales de beneficios por provincia. Dichas variables son inicializadas a cero en las líneas `BEGIN`, luego se van sumando en las líneas principales de procesamiento y finalmente son presentadas en las líneas `END`:

```
…
BEGIN {print "" ; TSAL=0 ; BENMD=0 ; BENMA=0 }
$4 ~ /Madrid/ {… ; TSAL=TSAL+$5 ; BENMD=BENMD+$6}
$4 ~ /Malaga/ {… ; TSAL=TSAL+$5 ; BENMA=BENMA+$6}
…
END {print "Total Salarios: " TSAL }
END {print "Total Beneficios"}
END {print "Madrid: " BENMD " Malaga: " BENMA }
```

Como podemos ver, las variables definidas por el usuario no se anteponen con el carácter `$`, al contrario de lo que se hace en la programación shell.

5.4.7 Operadores y funciones

En un programa `awk` podemos usar operaciones aritméticas y funciones. La sintaxis es similar a la de `c`. En la Tabla 5.3 podemos ver los principales operadores [6,7].

suma, resta, producto, división, resto	`+ - * / %`
Exponenciación	`^`
incremento, decremento	`++ --`
Asignación	`= += -= *= /= %= ^=`
agrupación de operaciones	`( )`
Relacionales	`<= < >= > != ==`

Tabla 5.3: Operaciones permitidas con variables en `awk`

En el script formatea.awk vemos ejemplos de asignación con suma en las líneas principales de proceso, para obtener así al final los totales de beneficios de Madrid, Málaga y el total de sueldos:

```
…
$4 ~ /Madrid/     {printf "%-11s%-11s%-11s%8d%11d\n",$1,$2,$3,$5,$6
      TSAL+=$5
BENMD+=$6}
$4 ~ /Malaga/     {printf "%-11s%-11s%-11s%8d%11d\n",$1,$2,$3,$5,$6
TSAL+=$5
BENMA+=$6}
…
```

En cuanto a las funciones, podemos decir que el uso y sintaxis es idéntico al del lenguaje C. La lista es tan extensa como la que permite dicho lenguaje, por lo que simplemente citamos algunos ejemplos. Para más referencias, puede consultarse en el manual online de awk:

```
int(num)          parte entera truncada de un número
sqrt(num)         raiz cuadrada de un número
sin(num)          seno de un número
cos(num)          coseno de un número
rand()            devuelve un número aleatorio
```

5.5. Programas sed y awk

Tanto en sed como en awk podemos utilizar la misma notación que en programación shell para indicar en la primera línea cuál es el programa con el que un fichero de comandos se invoca. En este caso indicaremos sed, o awk y además la opción –f que indica que es un fichero de comandos [6,7]:

```
#!/usr/bin/awk     -f              #!/usr/bin/sed -f
…                             …
```

Podemos ver en el script formatea.awk cómo comienza en la primera línea indicando que se ejecutará con awk:

```
#!/usr/bin/awk -f
BEGIN {print "Nombre     Apellido1  Apellido2   Salario Beneficios"}
```

...

Al fichero tenemos que darle además permisos de ejecución, con ello bastará con invocarlo de la forma habitual para cualquier script:

```
$ formatea.awk listado.txt
```

CAPÍTULO 6. APLICACIONES PARA ADMINISTRACIÓN CON LA SHELL BASH

Exponemos a continuación en este capítulo, el código completo de cada uno de los scripts comentados en el capítulo 4. Estas aplicaciones han sido desarrolladas basándose en necesidades concretas de un administrador de sistemas en el desempeño diario de su trabajo. Existen aplicaciones mucho más complejas cuyo código ocuparía una cantidad considerable de páginas, por lo que se han seleccionado aquellas que son más reducidas en código, pero que han servido para exponer las características de la programación shell expuestas en los capítulos anteriores.

6.1. Análisis del Número de Argumentos de un Script: `argumentos.sh`

```
#!/bin/bash
#####################################################################
# ESTE CONJUNTO DE COMANDOS ANALIZA SI EL NUMERO DE ARGUMENTOS ES EL
# ADECUADO, QUE EN ESTE SCRIPT ES 2. SI LA CANTIDAD DE ARGUMENTOS ES
# INFERIOR O SUPERIOR, SE MANDA A CONSOLA UN MENSAJE CON EL USO DEL
# PROGRAMA, Y SE TERMINA EL SCRIPT CON UN CODIGO DE ERROR ALMACENADO
# EN LA VARIABLE ARG_INSUF.
# ESTAS LINEAS PUEDEN SER INCLUIDAS EN CUALQUIER SCRIPT QUE NECESITE
# COMPROBAR EL NUMERO DE ARGUMENTOS PASADOS.
#
# EJEMPLO DE USO:
#
#    # # (Uso correcto de argumentos con el script)
#    # ./argumentos.sh hola mundo
#    # echo $?
#    0
#    #
#    # # (Uso incorrecto de argumentos con el script. Muestra error)
#    # ./argumentos.sh hola mundo cruel
#    Uso: ./argumentos.sh arg1 arg2
#    # echo $?
#    51
#
#####################################################################

ARG_INSUF=51

if [ $# -ne 2 ]
then
  echo "Uso: $0 arg1 arg2"
  exit $ARG_INSUF
fi
```

6.2. Bombardeo con Mails a un Servidor de Correo: `mailbomb.sh`

```
#!/bin/bash
#####################################################################
# ESTE SCRIPT DEMUESTRA COMO PODEMOS CONECTAR CON UN SERVIDOR SMTP
# REMOTO PARA ENVIAR UN MAIL. EN REALIDAD REALIZA LO QUE SE DENOMINA
# UN "MAIL BOMBING". EL ENVIO DE CORREO SE REALIZA EN UN BUCLE
# INFINITO HASTA QUE LO DETENGAMOS. ESTO PROVOCA UN BOMBARDEO DE MAIL
# SOBRE EL CORREO REMOTO, SATURANDOLO DE MENSAJES. SI EL MENSAJE QUE
# ENVIAMOS ES SUFICIENTEMENTE GRANDE, PODEMOS SATURAR LA CUENTA DE
# CORREO.
#
# EJEMPLO DE USO:
#
#    # ./mailbomb.sh
#    # # (No se genera salida pero se enviaran correos al servidor)
#
#####################################################################
```

```
while true
do

# LA SECUENCIA DE COMANDOS ESTA PREPARADA PARA ENVIAR UN CORREO A
# UNA DIRECCION DE TERRA PREVIAMENTE DADA DE ALTA PARA HACER
# PRUEBAS. EL CORREO SE ENVIA A TRAVES DEL PUERTO DE SMTP (25)
telnet smtp.terra.es 25 <<COMANDOS > /dev/null 2>&1
helo origen.es
mail from: <prueba@origen.es>
rcpt to: <pringaillo1@terra.es>
data
From: prueba@origen.es
To: pringaillo1@terra.es
Subject: mensaje de prueba
Hola
.
quit
COMANDOS

# SE ENVIA UN CORREO CADA SEGUNDO
sleep 1
done
```

6.3. Backups del Sistema en Cdrom: `cdbackup.sh`

```
#!/bin/bash
#########################################################################
# ESTE SCRIPT PERMITE HACER UN BACKUP DE UN SISTEMA DE FICHEROS EN CD,
# DE ESA FORMA PODEMOS VOLCAR UNA COPIA DE SEGURIDAD POR EJEMPLO EN
# REGRABABLES. PARA ELLO SE USAN LOS COMANDOS "DUMP" QUE GENERA UN
# FICHERO DE VOLCADO, "MKISOFS" QUE GENERA UN FICHERO ISO GRABABLE EN
# CD Y "CDRECORD" QUE GRABA EN CD. LA PARTE DE GRABACION ESTA SEPARADA
# EN UN SCRIPT COMPLEMENTARIO POR SI SE DESEA CAMBIAR EL METODO DE
# GRABACION.
#
# EJEMPLO DE USO:
# EN ESTE EJEMPLO SE HACE UNA COPIA DE SEGURIDAD DEL SISTEMA /usr
# USANDO EL DIRECTORIO TEMPORAL /prueba CON NIVEL 0 (BACKUP COMPLETO)
# CON CDS DE 700MB Y CON SCRIPT DE BACKUP "/scripts/grabacd.sh"
#
# cdbackup.sh -d /prueba -s /scripts/grabacd.sh -n -t 700000 /usr
# Fecha del sistema:   0603281800
# Sistema de Ficheros: /usr
# Nivel de la copia:   0
# Directorio temporal: /prueba
# Ficheros temporales: /prueba/0603281800.15058.dump y
#    /prueba/0603281800.15058.iso
# Tamano de imagen:    700000
# Script de Volcado:   /prueba/grabacd.sh
# Numero de CD:        7.8
#   DUMP: Date of this level 0 dump: Tue Mar 28 18:00:32 2006
#   DUMP: Dumping /dev/hda3 (/usr) to /prueba/0603281800.15058.dump
#   ...
#   DUMP: 3.40% done at 780 kB/s, finished in 2:22
#   DUMP: Closing /prueba/0603281800.15058.dump
#   DUMP: Volume 1 completed at: Tue Mar 28 18:08:18 2006
#   ...
#   DUMP: Launching /prueba/grabacd.sh
# Generando la imagen de /prueba/0603281800.15058.dump sobre
#      /prueba/0603281800.15058.iso
# Inserte un CD para grabar
```

```
# Grabando el cd sobre /dev/hdc
#   DUMP: Volume 2 started with block 700001 at: Tue Mar 28 18:14:47
#      2006
#   DUMP: Volume 2 begins with blocks from inode 120258
#   DUMP: 10.16% done at 0 kB/s, finished in 2:03
#   DUMP: Closing /prueba/0603281800.15058.dump
#   ...
#   DUMP: Launching /prueba/grabacd2.sh
# Generando la imagen de /prueba/0603281800.15058.dump sobre
#    /prueba/0603281800.15058.iso
# Inserte un CD para grabar
# Grabando el cd sobre /dev/hdc
#  DUMP: Volume 3 started with block 1400001 at: Tue Mar 28 18:24:57
#      2006
#  DUMP: Volume 3 begins with blocks from inode 230706
#  DUMP: 20.33% done at 0 kB/s, finished in 1:34
#  ...
#
#########################################################################

# FUNCION PARA MOSTRAR EL USO DEL SCRIPT

function uso
{
 echo "$0 Uso:"
 echo "$0 -d <dirbase> -s <script> -n <nivel> -t <tamano> \
 <sistema de ficheros>"
 }

# BUCLE PARA ASIGNAR LAS VARIABLES DE LAS OPCIONES

while getopts ":d:s:n:t:" OPCION
do
  case $OPCION in
    d)
     # DIRECTORIO TEMPORAL PARA ALMACENAR FICHEROS
     CDBACKUP_TARGET_DIR=$OPTARG
     ;;
    s)
     # SCRIPT DE GRABACION
     CDBACKUP_SCRIPT=$OPTARG
     ;;
    n)
     # NIVEL DEL BACKUP
     CDBACKUP_NIVEL=$OPTARG
     ;;
    t)
     # TAMANIO DEL CD EN KB
     CDBACKUP_TAMANNO=$OPTARG
  esac
done

# UNA VEZ PASADAS TODAS LAS OPCIONES, EL RESTO DE LA CADENA SE ASUME
# QUE SON ARGUMENTOS. SOLO SE ESPERA UN ARGUMENTO QUE ES EL SISTEMA DE
# FICHEROS AL QUE HACER EL BACKUP. SI HAY MAS ARGUMENTOS SE PRESENTA
# UN MENSAJE DE ERROR.
shift $(($OPTIND - 1))

# SISTEMA AL QUE SE LE HARA EL BACKUP
if [ $# != 1 ]
 then
  uso
  exit 1
 else
```

```
  SISTEMA=$1
fi

# VARIABLE CON LA FECHA ACTUAL PARA NOMBRAR LOS FICHEROS
FECHA=$(date +%y%m%d%H%M)

# COMPROBANDO QUE EXISTE EL DIRECTORIO TEMPORAL
# Y EL SCRIPT DE GRABACION
[ ! -d ${CDBACKUP_TARGET_DIR:=/tmp} ] && uso && exit 1
[ ! -x ${CDBACKUP_SCRIPT:=grabacd.sh} ] && uso && exit 1

# FICHERO DONDE SE HARA EL BACKUP
[ -z $CDBACKUP_DUMP ] &&
CDBACKUP_DUMP=${CDBACKUP_TARGET_DIR}/${FECHA}.$$.dump

# FICHERO CON LA IMAGEN ISO PARA GRABAR
[ -z $CDBACKUP_ISO ] &&
CDBACKUP_ISO=${CDBACKUP_TARGET_DIR}/${FECHA}.$$.iso
export CDBACKUP_DUMP CDBACKUP_ISO

# LINEAS DE INFORMACION EN PANTALLA Y VALORES POR DEFECTO
echo "Fecha del sistema:   $FECHA"
echo "Sistema de Ficheros: $SISTEMA"
echo "Nivel de la copia:   ${CDBACKUP_NIVEL:=0}"
echo "Directorio temporal: $CDBACKUP_TARGET_DIR"
echo "Ficheros temporales: $CDBACKUP_DUMP y $CDBACKUP_ISO"
echo "Tamano de imagen:    ${CDBACKUP_TAMANNO:=700000}"
echo "Script de Volcado:   $CDBACKUP_SCRIPT"

# CALCULO DEL NUMERO DE CDS QUE SE NECESITAN
TAMANNO_DUMP=$(/sbin/dump -$CDBACKUP_NIVEL -S -L $FECHA $1 2> \
/dev/null)
NUM_CD=$(dc -e "1 k $TAMANNO_DUMP 1024 / $CDBACKUP_TAMANNO / p")

echo "Numero de CD:        $NUM_CD"

# DUMP DEL SISTEMA DE FICHEROS
/sbin/dump -$CDBACKUP_NIVEL -B $CDBACKUP_TAMANNO -F $CDBACKUP_SCRIPT \
-L $FECHA -f $CDBACKUP_DUMP -u $SISTEMA
[ $? != 0 ] && echo "ERROR EN EL DUMP" && exit 1

# SE LANZA EL SCRIPT DE BACKUP POR ULTIMA VEZ
. $CDBACKUP_SCRIPT

# BORRADO DE LOS FICHEROS TEMPORALES GENERADOS
echo "Borrando $CDBACKUP_DUMP y $CDBACKUP_ISO"
rm $CDBACKUP_DUMP $CDBACKUP_ISO
```

6.4. Grabación en cd de una Imagen de Cdrom en formato .iso: `grabacd.sh`

```
#!/bin/bash
#########################################################################
#
# ESTE ES EL FICHERO QUE COMPLEMENTA AL SCRIPT cdbackup.sh. SE ENCARGA
# DE HACER LA IMAGEN ISO Y LA GRABACION EN CD.
#
# EJEMPLO DE USO:
# (ESTE SCRIPT SOLO NECESITA SER INVOCADO. TODAS LAS VARIABLES QUE USA
# ASUME QUE ESTAN DEFINIDAS EN EL SCRIPT QUE LO INVOCA)
```

```
#
# #./grabacd.sh
#
#########################################################################

# LOS NOMBRES DE LOS FICHEROS ESTAN ALMACENADOS EN LAS VARIABLES
# CDBACKUP_DUMP y CDBACKUP_ISO

# SE INICIALIZA EL DISPOSITIVO DE BACKUP
CDBACKUP_DEV="/dev/hdc"

echo "Generando la imagen de $CDBACKUP_DUMP sobre $CDBACKUP_ISO"
echo "Inserte un CD para grabar"
mkisofs -r -o $CDBACKUP_ISO $CDBACKUP_DUMP > /dev/null 2>&1

 [ $? != 0 ] && echo "Error en la imagen ISO" && exit 1

echo "Grabando el cd sobre $CDBACKUP_DEV"
cdrecord -v -eject speed=4 dev=$CDBACKUP_DEV $CDBACKUP_ISO > \
/dev/null 2>&1
CDBACKUP_ERROR=$?

if [ $CDBACKUP_ERROR != 0 ]
then
  echo "Ha fallado cdrecord con error : " $CDBACKUP_ERROR
  echo Escriba exit para continuar, cuando haya resuelto el error.
  bash
fi
```

6.5. Representación de un Cursor en Pantalla:

cursor.sh

```
#!/bin/bash
#########################################################################
# ESTE SCRIPT PUEDE SER UTIL CUANDO UN PROCESO ESTA REALIZANDO UNA
# TAREA QUE REQUIERE LA ESPERA DEL USUARIO. SIMPLEMENTE MUESTRA UN
# CURSOR GIRANDO. GENERALMENTE SE LANZARA ESTE PROCESO EN BACKGROUND
# DESDE OTRO QUE LO NECESITE, Y CUANDO ACABE DE REALIZAR LAS TAREAS
# NECESARIAS, MATARA AL PROCESO CURSOR UTILIZANDO LA VARIABLE $!
#########################################################################

# PONE EL CURSOR INVISIBLE

tput civis

# CUANDO EL PROCESO ACABE, PONE DE NUEVO EL CURSOR NORMAL

trap 'tput cnorm; exit' INT TERM

# DIBUJO DEL CURSOR

while true; do
echo -e "-\b\c"
sleep 1
echo -e "/\b\c"
sleep 1
echo -e "|\b\c"
sleep 1
echo -e "\\\\\b\c"
sleep 1
```

```
done
```

6.6. Ejemplo de uso del Script cursor.sh:

usacursor.sh

```
#!/bin/bash
#########################################################################
# ESTE SCRIPT TIENE COMO UNICA FINALIDAD EL MOSTRAR EL USO DEL SCRIPT
# CURSOR. DICHO SCRIPT ES LANZADO EN BACKGROUND, SE ESPERA DURANTE 10
# SEGUNDOS, Y FINALMENTE SE MATA EL PROCESO CURSOR.
#
# EJEMPLO DE USO:
#
# # ./usacursor.sh
#  espere 10 segundos...
#  /                          (Cursor girando 10 segundos)
#  Ahora puede continuar
#
#########################################################################

echo espere 10 segundos...
./cursor.sh &
sleep 10
kill $!
echo Ahora puede continuar
```

6.7. Escáner de red: escanea_ip.sh

```
#!/bin/bash
#########################################################################
# ESTE SCRIPT ESCANEA UN RANGO DE DIRECCIONES IP DE 1 A 254 MEDIANTE
# EL COMANDO PING. LA RED A ESCANEAR SE DA EN LA FORMA X.Y.Z COMO
# PARAMETRO DEL SCRIPT.
#
# EJEMPLO DE USO:
#
# # ./escanea_ip.sh 192.168.1
#  Probando 192.168.1.1
#  Ok
#  Probando 192.168.1.2
#
#  Probando 192.168.1.3
#  Ok
#  Probando 192.168.1.4
#  ...
#
#
# # FICHERO GENERADO
# # more 192.168.1.0.out
#  192.168.1.1
#  192.168.1.3
#  192.168.1.6
#  ...
#  192.168.1.210
#  192.168.1.230
#  192.168.1.231
```

```
#
#####################################################################

for i in $(X=1; while(( X<=254 ));do echo $X; (( X=X+1 )); done )
do
   echo "Probando $1.$i"

# EL COMANDO PING ENVIA UN UNICO PAQUETE Y ESPERA UN SEGUNDO. SE PUEDE
# CAMBIAR ESTOS VALORES EN FUNCION DEL TIEMPO DE RESPUESTA DE LA RED.

  ping -w 1 -c 1 "$1.$i" > /dev/null
   if [[ $? = "0" ]]
   then

# LA SALIDA DE LAS IPS QUE RESPONDEN CORRECTAMENTE SE DA EN FORMA DE
# FICHERO ?X.Y.Z.0.out?.

      echo "$1.$i" >> "$1.0.out"
      echo " Ok"
   else
      echo
   fi
done
```

6.8. Escáner de red: `escanea_ip_2.sh`

```
#!/bin/bash
#####################################################################
# ESTE SCRIPT ES IDENTICO A escanea_ip.sh PERO USANDO EL TIPO DE
# BUCLE FOR QUE APORTA LA SHELL BASH, LO QUE SINPLIFICA LA SINTAXIS.
#
# AL IGUAL QUE escanea_ip.sh, ESTE SCRIPT ESCANEA UN RANGO DE
# DIRECCIONES IP DE 1 A 254 MEDIANTE EL COMANDO PING. LA RED A
# ESCANEAR SE DA EN LA FORMA X.Y.Z COMO PARAMETRO DEL SCRIPT.
#
# EJEMPLO DE USO:
#
# # ./escanea_ip.sh 192.168.1
#  Probando 192.168.1.1
#  Ok
#  Probando 192.168.1.2
#
#  Probando 192.168.1.3
#  Ok
#  Probando 192.168.1.4
#  ...
#
#
# # FICHERO GENERADO
# # more 192.168.1.0.out
#  192.168.1.1
#  192.168.1.3
#  192.168.1.6
#  ...
#  192.168.1.210
#  192.168.1.230
#  192.168.1.231
#
#####################################################################

for (( i=1 ;  i<=254 ; i=i+1 ))
```

```
do
   echo "Probando $1.$i"

# EL COMANDO PING ENVIA UN UNICO PAQUETE Y ESPERA UN SEGUNDO. SE PUEDE
# CAMBIAR ESTOS VALORES EN FUNCION DEL TIEMPO DE RESPUESTA DE LA RED.

  ping -w 1 -c 1 "$1.$i" > /dev/null
   if [[ $? = "0" ]]
   then

# LA SALIDA DE LAS IPS QUE RESPONDEN CORRECTAMENTE SE DA EN FORMA DE
# FICHERO ?X.Y.Z.0.out?.

      echo "$1.$i" >> "$1.0.out"
      echo " Ok"
   else
      echo
   fi
done
```

6.9. Backup y Eliminación de Ficheros log:

logrotate.sh

```
#!/bin/bash
#########################################################################
# ESTE SCRIPT IMPLEMENTA UNA OPERACION QUE LINUX REALIZA CON CIERTOS
# FICHEROS LOGS, CONSISTENTE EN ALMACENAR EN FORMATO GZIP DICHO
# FICHERO PARA QUE NO OCUPE TANTO ESPACIO, Y DEJAR AL LOG ORIGINAL CON
# CERO BYTES. DE ESTA MANERA, EL FICHERO LOG ORIGINAL NO PIERDE SUS
# PROPIEDADES (PERMISOS, PROPIETARIO, ETC) A LA COPIA DEL LOG SE LE
# RENOMBRA CON LA FECHA Y LA HORA DE LA OPERACIÃ POR SI
# POSTERIORMENTE SE QUIERE RECUPERAR PARA VER SU CONTENIDO. ADEMAS, SI
# SE SUPERA EL NUMERO DE COPIAS QUE SE GUARDA EN EL FICHERO DE
# CONFIGURACION ?LOGROTATE.CONF? SE BORRA LA COPIA MAS ANTIGUA QUE
# EXISTA. SE PUEDE PROGRAMAR UNA TAREA EN CRON QUE EJECUTE ESTE SCRIPT
# PERIODICAMENTE SOBRE UN DETERMINADO FICHERO. POR EJEMPLO UNA VEZ AL
# MES, EL 1 DE CADA MES A LAS 23:59:
#
#     59  23  31 *  *  /usr/local/almacenalog /var/log/milog
#
# EN EL NOMBRE DEL FICHERO COPIA SE HA USADO GUION COMO SEPARADOR DE
# FECHA YA QUE LOS CARACTERES / Y : TIENEN SIGNIFICADO PARA EL
# INTERPRETE DE COMANDOS.
#
# EJEMPLO DE USO (PARA UNA ROTACION DE CINCO LOGS):
# # ll prueba.*
#  -rw-r--r--  1 root root 15 Mar 27 13:11 prueba.txt
# # ./logrotate.sh prueba.txt
# # ll prueba.*
#  -rw-r--r--  1 root root  0 Mar 27 13:12 prueba.txt
#  -rw-r--r--  1 root root 56 Mar 27 13:11 prueba.txt.27-03-06-13-12-
#      11.gz
# # ./logrotate.sh prueba.txt
# # ll prueba.*
#  -rw-r--r--  1 root root  0 Mar 27 13:12 prueba.txt
#  -rw-r--r--  1 root root 56 Mar 27 13:11 prueba.txt.27-03-06-13-12-
#      11.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#      16.gz
# # ./logrotate.sh prueba.txt
# # ll prueba.*
```

```
#  -rw-r--r--  1 root root  0 Mar 27 13:12 prueba.txt
#  -rw-r--r--  1 root root 56 Mar 27 13:11 prueba.txt.27-03-06-13-12-
#      11.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#      16.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#      20.gz
# # ./logrotate.sh prueba.txt
# # ll prueba.*
#  -rw-r--r--  1 root root  0 Mar 27 13:12 prueba.txt
#  -rw-r--r--  1 root root 56 Mar 27 13:11 prueba.txt.27-03-06-13-12-
#      11.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#      16.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#      20.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#      25.gz
# # ./logrotate.sh prueba.txt
# # ll prueba.*
#  -rw-r--r--  1 root root  0 Mar 27 13:12 prueba.txt
#  -rw-r--r--  1 root root 56 Mar 27 13:11 prueba.txt.27-03-06-13-12-
#      11.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#      16.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#      20.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#      25.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#      29.gz
# # ./logrotate.sh prueba.txt
# # ll prueba.*
#  -rw-r--r--  1 root root  0 Mar 27 13:12 prueba.txt
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#     16.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#     20.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#     25.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#     29.gz
#  -rw-r--r--  1 root root 49 Mar 27 13:12 prueba.txt.27-03-06-13-12-
#     33.gz
#
#######################################################################

# SE LEE UN FICHERO DE CONFIGURACION PARA VER EL NUMERO MAXIMO DE LOG
# QUE PERMITIMOS.

. logrotate.conf
if [ -f $1 ]
then
      NUMLOG=$( ls -1 $1.* 2> /dev/null | wc -l)
      if [ $NUMLOG -eq $MAXLOG ]
      then
        LISTA_FICHEROS=$(echo $1.*) 2>/dev/null
        ANTIGUO=${LISTA_FICHEROS%% *}
        rm $ANTIGUO
      fi
       FECHA=$(date +%d-%m-%y-%k-%M-%S)

# SE SACA LA COPIA DEL FICHERO ANADIENDO FECHA Y HORA AL NOMBRE
```

```
       cp  -pf $1 "${1}.$FECHA"

# SE COMPRIME EL FICHERO COPIADO

       gzip "${1}.$FECHA"

# SE DEJA EL FICHERO ORIGINAL CON CERO BYTES

       > $1
else
  echo "El fichero $1 no existe"
  exit 1
fi
```

logrotate.conf

```
############################################################
#
# FICHERO DE CONFIGURACION PARA EL SCRIPT logrotate.sh
#
############################################################

MAXLOG=5
```

6.10. Conversión de Nombres de Ficheros a Caracteres Minúsculas: `minusculas.sh`

```
#!/bin/ksh
#########################################################################
# ATENCION, ESTE SCRIPT USA KSH YA QUE CIERTAS CARACTERISTICAS DE
# TYPESET SON ESPECIFICAS DE ESTA SHELL
#########################################################################

#########################################################################
# ESTE SCRIPT RENOMBRA LOS FICHEROS DE UN DIRECTORIO PARA QUE TODOS
# SUS CARACTERES SEAN MINUSCULAS. A VECES, AL DESCARGAR FICHEROS DESDE
# UN SITIO REMOTO O AL TRASFERIR ENTRE LINUX Y WINDOWS, CIERTOS
# FICHEROS PONEN SU EXTENSION O SU NOMBRE A MAYUSCULAS, LO QUE PUEDE
# DAR PROBLEMAS DE COMPATIBILIDAD CON LAS APLICACIONES. CON ESTE
# SENCILLO SCRIPT, SE COMPRUEBAN LOS NOMBRES Y SE CONVIERTEN TODOS A
# MINUSCULAS
#
# EJEMPLO DE USO:
#
#  # ll prueba
#  total 8
#  drwxr-xr-x  2 root root 4096 Mar 27 13:52 .
#  drwxr-xr-x  3 root root 4096 Mar 29 08:12 ..
#  -rw-r--r--  1 root root    0 Mar 27 13:16 FILE1
#  -rw-r--r--  1 root root    0 Mar 27 13:16 FILE2
#  -rw-r--r--  1 root root    0 Mar 27 13:16 FILE3
#  # ./minusculas.sh prueba
#  Moviendo prueba/FILE1 a prueba/file1
#  Moviendo prueba/FILE2 a prueba/file2
#  Moviendo prueba/FILE3 a prueba/file3
#  # ll prueba
#  total 8
#  drwxr-xr-x  2 root root 4096 Mar 29 08:12 .
```

```
#  drwxr-xr-x  3 root root 4096 Mar 29 08:12 ..
#  -rw-r--r--  1 root root    0 Mar 27 13:16 file1
#  -rw-r--r--  1 root root    0 Mar 27 13:16 file2
#  -rw-r--r--  1 root root    0 Mar 27 13:16 file3
#
######################################################################

# SE ASIGNAN NOMBRES MAS RECONOCIBLES A LAS VARIABLES PASADAS COMO
# ARGUMENTOS

DIRECTORIO=$1

# LA VARIABLE NOMMINUSCULAS SIEMPRE VA A TENER VALORES EN MINUSCULAS,
# AUNQUE SE LE ASIGNEN  EN MAYUSCULAS SE APROVECHA ESTA CARACTERISTICA
# PARA HACER UN SENCILLO MOVE

typeset -l NOMMINUSCULAS

# SE TOMAN TODOS LOS FICHEROS DEL DIRECTORIO EN ESTE BUCLE Y SE VAN
# PROCESANDO

for FICHERO in $DIRECTORIO/*
do

# LA VARIABLE NOMNUEVO SERA IGUAL QUE LA VARIABLE FICHERO, PERO CON EL
# NOMBRE DEL FICHERO EN MINUSCULAS

  NOMMINUSCULAS=${FICHERO##*/}
  NOMNUEVO=${DIRECTORIO}/${NOMMINUSCULAS}

# SE COMPRUEBA QUE EL NUEVO NOMBRE NO EXISTE COMO FICHERO, Y EN CASO
# CONTRARIO, SE PREGUNTA SI SE QUIERE SOBREESCRIBIR

  if [[ -f ${NOMNUEVO} && ${NOMNUEVO} != ${FICHERO} ]]
  then
    echo -n "El fichero ${NOMNUEVO} existe. Sobreescribir? (S/N) "
    read RESPUESTA
    if [[ $RESPUESTA = S* || $RESPUESTA = s* ]]
    then
       mv "${FICHERO}" "${NOMNUEVO}" 2> /dev/null && echo  \
          "Moviendo ${FICHERO} a ${NOMNUEVO}"
    else
       echo "Ignorado el fichero ${NOMNUEVO}"
    fi

# PARA LOS FICHEROS QUE NO TENGAN CONFLICTO, SIMPLEMENTE SE MUEVEN AL
# NUEVO NOMBRE

  else
    mv "${FICHERO}" "${NOMNUEVO}" 2> /dev/null && echo \
       "Moviendo ${FICHERO} a ${NOMNUEVO}"
  fi
done
```

6.11. Desglose de un Nombre Completo de directorio o Fichero en sus Componentes: `path.sh`

```
#!/bin/bash
######################################################################
```

```
#
# ESTE SCRIPT TOMA COMO ARGUMENTO UN PATH O DIRECTORIO DE LA FORMA
# /dir1/dir2/dir3 Y LO DESGLOSA EN TODOS SUS SUBDIRECTORIOS,
# ALMACENANDO CADA DIRECTORIO EN UNA VARIABLE ARRAY LLAMADA NAME.
# PUEDE USARSE EL RESULTADO DE ESTE SCRIPT PARA CUALQUIER OPERACIÃ
# QUE REQUIERA DESGLOSAR UN PATH DE DIRECTORIO EN TODOS SUS
# COMPONENTES.
#
# EJEMPLO DE USO:
#
# # ./path.sh /uno/dos/tres
# El path facilitado es: /uno/dos/tres
# El path se compone de: uno dos tres
#
# NAME[0]=uno
# NAME[1]=dos
# NAME[2]=tres
#
#########################################################################

# SE USA UNA VARIABLE TEMPORAL path1

PATH=$1
PATH1=${PATH#/}

# SE USAN OPERACIONES DE CADENA SOBRE LA VARIABLE PATH1 PARA IR
# EXTRAYENDO LOS COMPONENTES Y ALMACENARLOS EN NAME

X=0
until [[ $PATH1 != */* ]]
do
    NAME[$X]=${PATH1%%/*}
    PATH1=${PATH1#*/}
    ((X = X + 1))
done

NAME[$X]=$PATH1

# ESTAS ULTIMAS LINEAS DE CODIGO SIMPLEMENTE MUESTRAN EL RESULTADO
# OBTENIDO. SE PUEDEN ELIMINAR Y USAR EL SCRIPT COMO FUNCION DENTRO DE
# OTRO.

echo "El path facilitado es: $PATH"
echo "El path se compone de: ${NAME[*]}"
echo

X=0
for i in ${NAME[*]}
do
  echo "NAME[$X]=$i"
  ((X = X + 1))
Done
```

6.12. Difusión de un Correo a Múltiples Usuarios:

broadmail.sh

```
#!/bin/bash
#########################################################################
# ESTE SCRIPT ENVIA UN MAIL A TODOS LOS USUARIOS QUE ESTAN CONECTADOS
# AL SISTEMA EN ESTE MISMO INSTANTE.
#
```

```
# EJEMPLO DE USO:
#
# # ./broadmail.sh prueba.txt
# Enviando mail a pedrolopezs
# Enviando mail a root
# # mail
# mailx version nail 11.4 8/29/04.  Type ? for help.
# "/var/mail/pedrolopezs": 1 messages 1 new
# >N  1 root@neoborabora.h Mon Mar 27 13:27   17/598
# ?
# Message  1:
# From root@neoborabora.home  Mon Mar 27 13:27:07 2006
# X-Original-To: root
# Delivered-To: root@neoborabora.home
# Date: Mon, 27 Mar 2006 13:27:06 +0200
# To: root@neoborabora.home
# User-Agent: nail 11.4 8/29/04
# MIME-Version: 1.0
# Content-Type: text/plain; charset=us-ascii
# Content-Transfer-Encoding: 7bit
# From: root@neoborabora.home (root)
#
# Mensaje enviado por mail a todos los usuarios conectados
#
# ?
# At EOF
# ? q
# # (Todos los usuarios que tengan sesion abierta tendran mail)
#########################################################################

# SE USA awk Y sort SOBRE EL COMANDO who PARA OBTENER LA LISTA DE
# USUARIOS QUE ESTAN ACTUALMENTE CONECTADOS.

for i in $(who | awk '{print $1}' | sort -u)
do
echo "Enviando mail a $i"

# EL MENSAJE SE ENVIA USANDO UNA ESTRUCTURA "HERE DOCUMENT" EL TEXTO A
# ENVIAR DEBE PONERSE ENTRE LAS DOS ETIQUETAS "TEXTO".

mail $i <<TEXTO
Mensaje enviado por mail a todos los usuarios conectados
TEXTO

done
```

6.13. Desglose de un Fichero de Base de Datos en sus Componentes: `cortafichero.sh`

```
#!/bin/ksh
#########################################################################
# ATENCION, ESTE SCRIPT USA KSH YA QUE CIERTAS CARACTERISTICAS DE
# TYPESET SON ESPECIFICAS DE ESTA SHELL
#########################################################################
```

```
#####################################################################
#
# ESTE SCRIPT SURGE ANTE LA NECESIDAD DE CONVERTIR UN FICHERO DE BASES
# DE DATOS USADO EN UN SISTEMA MAINFRAME. EL FICHERO CONTIENE EN CADA
# LINEA UN REGISTRO. CADA REGISTRO CONTIENE UNA SERIE DE CAMPOS QUE NO
# ESTAN SEPARADOS POR NINGUN CARACTER ESPECIAL, PERO ES CONOCIDO DONDE
# EMPIEZA Y ACABA CADA UNO DE ELLOS.
#
# EL SCRIPT ACEPTA COMO ENTRADA EL NOMBRE DEL FICHERO A CONVERTIR, Y
# LAS LONGITUDES DE LOS CAMPOS QUE SE QUIEREN SEPARAR. USANDO FICHEROS
# TEMPORALES Y COMANDOS CUT Y PASTE, SE GENERA UN FICHERO IDENTICO AL
# ORIGINAL, PERO CON CADA CAMPO DE CADA REGISTRO SEPARADO POR EL
# CARACTER : LO CUAL PERMITIRA SER TRATADO POR CUALQUIER HERRAMIENTA
# QUE PERMITA MIGRAR A LA BASE DE DATOS, O PARA ORDENAR LINEAS POR
# CAMPOS.
#
# EJEMPLO DE USO:
#
# # cat datos
# JOSE   PEREZ    RODRIGUEZCOSLADA MADRID
# ANA    FERNANDEZPEREZ    VILANOVABARCELONA
# JUAN   MARTIN   VERA     ALHAURINMALAGA
# ANTONIOVERA     GOMEZ    VILLALBAMADRID
# MARIA  PUEYO    SAGOLS   GABA    BARCELONA
# JUAN   FERNANDEZLOPEZ    CABRA   CORDOBA
# # ./cortafichero.sh datos 7 9 9 8 9
# # cat datos.cortado
# JOSE   :PEREZ    :RODRIGUEZ:COSLADA :MADRID
# ANA    :FERNANDEZ:PEREZ    :VILANOVA:BARCELONA
# JUAN   :MARTIN   :VERA     :ALHAURIN:MALAGA
# ANTONIO:VERA     :GOMEZ    :VILLALBA:MADRID
# MARIA  :PUEYO    :SAGOLS   :GABA    :BARCELONA
# JUAN   :FERNANDEZ:LOPEZ    :CABRA   :CORDOBA
#
#####################################################################

# SE ASIGNAN NOMBRES MAS RECONOCIBLES A LAS VARIABLES PASADAS COMO
# ARGUMENTOS

FICHERO=$1
shift
CORTES=$*
OFFSET=0
CONTADOR=0

# SE VAN REALIZANDO LOS DIFERENTES CORTES DE CADA COLUMNA Y SON
# ALMACENADOS EN FICHEROS TEMPORALES

for LONGITUD in $CORTES
do
(( CONTADOR = CONTADOR + 1 ))
(( PRINCIPIO = OFFSET + 1 ))
(( FINAL = OFFSET + LONGITUD ))
typeset -Z3 CONTADOR
cut -c $PRINCIPIO-$FINAL $FICHERO > /tmp/$FICHERO.$CONTADOR.$$
OFFSET=$FINAL
done

#SE REUNEN TODOS LOS CORTES EN UN SOLO FICHERO Y SE ELIMINAN TODOS LOS
# FICHEROS TEMPORALES

paste -d: /tmp/$FICHERO.*.$$ > $FICHERO.cortado
rm /tmp/$FICHERO.*.$$
```

6.14. Ejecución de un ftp a una Máquina Remota sin Intervención Manual : `ftpremoto.sh`

```
#!/bin/bash
#########################################################################
# ESTE SCRIPT DEMUESTRA EL USO DE UN FTP DE FORMA DESATENDIDA POR
# PARTE DEL USUARIO. ESTE SIMPLEMENTE TIENE QUE PONER LA LISTA DE
# COMANDOS A REALIZAR, Y LA PASSWORD DEL USUARIO CON EL QUE VA A
# ENTRAR. PUEDE USARSE EN CUALQUIER TAREA QUE REQUIERA LA DESCARGA DE
# FICHEROS DESDE OTRA MAQUINA, COMO POR EJEMPLO, ACTUALIZACION DE
# SOFTWARE O PARCHES. EN ESTE CASO CONCRETO, SE USA PARA DESCARGAR EL
# CONTENIDO DE UN FICHERO README.txt EN EL DIRECTORIO DE TRABAJO DEL
# DEL USUARIO QUE SE CONECTA, DESDE EL SITIO FTP DE DESCARGAS DE LINUX
# SUSE, PERO SERIA IGUAL DE VALIDO CON CUALQUIER TIPO DE DESCARGA.
#
# EJEMPLO DE USO:
#
# neoborabora:/prueba # ll README.txt
# /bin/ls: README.txt: No such file or directory
# neoborabora:/prueba # ./ftpremoto.sh
# Transfiriendo ficheros...
# /
# Transferencia finalizada
# neoborabora:/prueba # ll README.txt
# -rw-r--r--  1 root root 353 Mar 14 11:30 README.txt
#
#########################################################################

# EN CASO DE INTERRUPCION FORTUITA DEL SCRIPT SE BORRA EL FICHERO
# .netrc QUE CONTIENE LA PASSWORD DEL USUARIO.

trap "rm $HOME/.netrc ; exit 1" INT QUIT TERM

# SE USA UN ?HERE DOCUMENT? PARA CREAR EL FICHERO .netrc. ESTE FICHERO
# PERMITE A FTP CONECTARSE A LA MAQUINA INDICADA EN machine CON EL
# USUARIO Y LA PASSWORD INDICADOS EN LA MISMA LINEA SIN INTERVENCION
# DEL USUARIO.

cat > $HOME/.netrc <<NETRC
machine ftp.suse.com login anonymous password pepe@hotmail.com
NETRC
chmod 600 $HOME/.netrc

# SE LANZA EL SCRIPT CURSOR.SH MIENTRAS SE EJECUTA EL FTP REMOTO

echo "Transfiriendo ficheros..."
./cursor.sh &

# SE UTILIZA TAMBIEN LA ESTRUCTURA "HERE DOCUMENT" PARA LANZAR LOS
# COMANDOS EN EL HOST REMOTO. LA LISTA DE COMANDOS VAN ENTRE LAS DOS
# ETIQUETAS "COMANDOS"

ftp ftp.suse.com > /dev/null <<COMANDOS
prompt n
hash
ascii
cd /pub/suse/i386/10.0
get README.txt
bye
COMANDOS
```

```
# ACABA CON EL SCRIPT CURSOR.SH EN EL MOMENTO DE FINALIZAR EL SCRIPT

kill $!
echo "Transferencia finalizada"

# SE BORRA EL FICHERO .netrc

rm $HOME/.netrc
```

6.15. Renombrado de Ficheros de Imágenes en Formato .jpg: `renombra.sh`

```
#!/bin/ksh
#########################################################################
# ATENCION, ESTE SCRIPT USA KSH YA QUE CIERTAS CARACTERISTICAS DE
# TYPESET SON ESPECIFICAS DE ESTA SHELL
#########################################################################

#########################################################################
# ESTE SENCILLO SCRIPT USA VARIAS CARACTERISTICAS DE ASIGNACION DE
# VARIABLES PARA RENOMBRAR TODOS LOS FICHEROS DE IMAGENES JPG QUE
# EXISTEN EN UN DIRECTORIO A UN NOMBRE COMUN Y UNA NUMERACION ORDENADA
# CON N DIGITOS. POR EJEMPLO CUMPLEAÑOS-0023.jpg. ESTO SE PUEDE USAR
# PARA UNIFORMIZAR LOS NOMBRES EN UNA COLECCION DE IMAGENES, AUNQUE SE
# PUEDE USAR EN OTRAS CIRCUNSTANCIAS. SE PASA COMO PARAMETROS EL
# DIRECTORIO DE TRABAJO, EL NOMBRE BASE A USAR, EL NUMERO DE ORDEN
# INICIAL Y LOS DIGITOS QUE LLEVARA LA NUMERACION.
#
# EJEMPLO DE USO:
#
# # ll cumplefiesta/
# total 8
# drwxr-xr-x  2 root root 4096 Mar 29 09:27 .
# drwxr-xr-x  4 root root 4096 Mar 29 09:26 ..
# -rw-r--r--  1 root root    0 Mar 29 09:27 DSCN-0001.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 DSCN-0002.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 DSCN-0003.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 DSCN-0004.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 DSCN-0005.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 DSCN-0006.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 DSCN-0007.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 DSCN-0008.jpg
# #
# # ./renombra.sh cumplefiesta/ "cumple2006-" 10 4
# #
# # ll cumplefiesta/
# total 8
# drwxr-xr-x  2 root root 4096 Mar 29 09:29 .
# drwxr-xr-x  4 root root 4096 Mar 29 09:26 ..
# -rw-r--r--  1 root root    0 Mar 29 09:27 cumple2006-0010.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 cumple2006-0011.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 cumple2006-0012.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 cumple2006-0013.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 cumple2006-0014.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 cumple2006-0015.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 cumple2006-0016.jpg
# -rw-r--r--  1 root root    0 Mar 29 09:27 cumple2006-0017.jpg
#
```

```
#####################################################################

# SE ASIGNAN NOMBRES MAS RECONOCIBLES A LAS VARIABLES PASADAS COMO
# ARGUMENTOS

DIRECTORIO=$1
NOMBASE=$2
NUMBASE=$3
DIGITOS=$4

# LA VARIABLE NUMBASE SE DEFINE COMO DE N DIGITOS, PASADOS EN EL
# CUARTO ARGUMENTO Y SE RELLENA CON CEROS A LA IZQUIERDA

typeset -RZ${DIGITOS} NUMBASE

# SE TOMAN SOLO LOS FICHEROS DE EXTENSION .JPG O .jpg Y SE RENOMBRAN
# COMPONIENDO EL NOMBRE EN BASE A LAS VARIABLES PASADAS EL CONTADOR ES
# INCREMENTADO

for i in $DIRECTORIO/*.jpg $DIRECTORIO/*.JPG
do
  mv "$i" "${DIRECTORIO}/${NOMBASE}${NUMBASE}.jpg" 2> /dev/null
  (( NUMBASE = NUMBASE + 1 ))
Done
```

6.16. Gestión de Paquetes de Instalación en Formato .rpm: `maneja_rpm.sh`

```
#!/bin/bash

#####################################################################
# EL MANEJO DE PAQUETES EN FORMATO RPM PARA INSTALAR Y DESINSTALAR A
# VECES PUEDE SER COMPLEJO PARA EL USUARIO YA QUE AUNQUE EL COMANDO ES
# UNO SOLO (RPM) MANEJA UNA VARIEDAD DE OPCIONES BASTANTE AMPLIAS
# DEPENDIENDO DE LO QUE QUERAMOS HACER... INSTALAR, DESINSTALAR,
# LISTAR FICHEROS. AUNQUE LA MAYORIA DE LOS ENTORNOS GRAFICOS YA
# APORTAN HERRAMIENTAS PARA HACER FACIL EL USO DE DICHO COMANDO
# (GNORPM POR EJEMPLO) SIEMPRE ES UTIL TENER UN SCRIPT QUE EN CONSOLA
# NOS PERMITA HACER LAS OPCIONES MAS COMUNES DE RPM. ESE ES EL
# COMETIDO DE ESTE SCRIPT.
#
# SE HA SELECCIONADO EL FORMATO RPM, POR SER EL MAS EXTENDIDO ENTRE
# LAS DISTRIBUCIONES LINUX, AUNQUE ESTE SCRIPT PUEDE ADAPTARSE
# FACILMENTE A CUALQUIER OTRO FORMATO INCLUSO PARA PAQUETES DE HP-UX O
# SOLARIS
#
# ENTRE OTRAS CARACTERISTICAS DE LA PROGRAMACION SHELL, ESTE SCRIPT
# USA LAS FUNCIONES Y LA SENTENCIA SELECT EL USO DE FUNCIONES SE
# APROVECHA PARA IMPLEMENTAR CADA UNA DE LAS CUATRO OPCIONES DEL MENU
# PRINCIPAL. CADA UNA DE ELLAS ES UNA OPCION DE MENU. LA SENTENCIA
# SELECT SE USA EN VARIOS PUNTOS DEL SCRIPT PARA MOSTRAR MENUS CON LAS
# OPCIONES QUE ESTAN DISPONIBLES.
#
# USO DEL SCRIPT:
# LOS MENUS QUE NOS VAMOS ENCONTRANDO SON LOS SIGUIENTES:
#
#  Menu principal
#  |
#  |--> Instalar un paquete
```

```
#   |       |
#   |       |--> Cdrom de linux
#   |       |              |
#   |       |              |--> Lista de paquetes
#   |       |              |
#   |       |              ---> No instalar mas paquetes
#   |       |
#   |       ---> Directorio particular
#   |                      |
#   |                      |--> Lista de paquetes
#   |                      |
#   |                      ---> No instalar mas paquetes
#   |
#   |--> Desinstalar un paquete
#   |       |
#   |       |--> Lista de paquetes
#   |       |
#   |       ---> No desinstalar mas paquetes
#   |
#   |--> Informacion de paquetes no instalados
#   |       |
#   |       |--> Cdrom de Linux
#   |       |       |
#   |       |       |--> Lista de paquetes
#   |       |       |          |
#   |       |       |          |--> Informacion del paquete
#   |       |       |          |
#   |       |       |          ---> Listado de ficheros del paquete
#   |       |       |
#   |       |       ---> No consultar mas paquetes
#   |       |
#   |       ---> Directorio particular
#   |               |
#   |               |--> Lista de paquetes
#   |               |          |
#   |               |          |--> Informacion del paquete
#   |               |          |
#   |               |          ---> Listado de ficheros del paquete
#   |               |
#   |               ---> No consultar mas paquetes
#   |
#   |--> Informacion de paquetes instalados
#   |               |
#   |               |--> Lista de paquetes
#   |               |          |
#   |               |          |--> Informacion del paquete
#   |               |          |
#   |               |          ---> Listado de ficheros del paquete
#   |               |
#   |               ---> No consultar mas paquetes
#   |
#   ---> Salir
#
#   EJEMPLO DEL MENU PRINCIPAL:
#
#   1) Instalar un paquete
#   2) Desinstalar un paquete
#   3) Informacion de paquetes no instalados
#   4) Informacion de paquetes instalados
#   5) Salir
#   Elija una Opcion (ENTER para menu):
#
#####################################################################
```

```
# FUNCION PARA DESINSTALAR PAQUETES

function desinstala
{

# EL LISTADO DE PAQUETES PUEDE SER MUY LARGO DE FORMA QUE SE SOLICITA
# UNA REFERENCIA PARA BUSCAR EL NOMBRE EN DICHA LISTA
 echo -n "Introduzca una cadena que forme parte del paquete \
   a desinstalar: "
 read NOMBRE

# LA VARIABLE PS3 DEFINE EL PROMPT QUE USARA EL MENU DE SELECT.
# A PARTIR DE AHORA SE CAMBIARA CADA VEZ QUE SE ENTRA O SALE DE UNA
# FUNCION
 PS3="Elija un paquete para desinstalar (ENTER para menu): "

# SE ASIGNA A LA VARIABLE RPM TODOS LOS PAQUETES INSTALADOS, MAS UNA
# OPCION PARA ACABAR

 select RPM in $(rpm -qa | grep -i ".*$NOMBRE.*" ) "No desinstalar \
    mas paquetes"
 do

# SE ANALIZAN LOS VALORES DE LA VARIABLE RPM Y SE REALIZA LA ACCION
# CORRESPONDIENTE

  case $RPM in
    "No desinstalar mas paquetes")
    clear
    PS3="Elija una opcion (ENTER para menu: "
    return 0
    ;;
    *)
    echo "Desinstalando el paquete $RPM"
    rpm -e $RPM >> /tmp/rpm.log 2>> /tmp/rpm.log
    if [ ! $? ]
    then
     echo "El paquete no ha podido ser desinstalado."
     echo "Chequee el fichero /tmp/rpm.log para ver los motivos"
    fi
    ;;
  esac
 done
}

# FUNCION PARA INSTALAR PAQUETES

function instala
{

# POR DEFECTO, EL DIRECTORIO DE BUSQUEDA SE ASUME QUE ES EL CDROM DE
# UNA DISTRIBUCION RED HAT
# ESTO PUEDE SER MODIFICADO PARA CUALQUIER OTRO TIPO DE DISTRIBUCION

  DIR=/mnt/cdrom/suse/i586
  PS3="Elija un paquete para instalar (ENTER para menu): "

# SE DA A ELEGIR ENTRE INSTALAR DE UN CD DE DISTRIBUCION LINUX O DE UN
# DIRECTORIO QUE CONTENGA PAQUETES RPM
  echo -n "Los paquetes se instalaran desde el cdrom de Linux? (s/n):"
  read OPCION
  if [[ $OPCION = n* || $OPCION = N* ]]
  then
   echo -n "Indique el directorio que contiene los paquetes: "
```

```
   read DIR
  fi
  cd $DIR

# EL LISTADO DE PAQUETES PUEDE SER MUY LARGO DE FORMA QUE SE SOLICITA
# UNA REFERENCIA PARA BUSCAR EL NOMBRE EN DICHA LISTA
 echo -n "Introduzca una cadena que forme parte del paquete a
instalar: "
 read NOMBRE

# COMO EN LA FUNCION ANTERIOR, AHORA SE ASIGNA LA VARIABLE RPM A TODOS
# LOS PAQUETES QUE HAY EN EL DIRECTORIO, MAS UNA OPCION PARA TERMINAR

  select RPM in $(ls -1 *.rpm | grep -i ".*$NOMBRE.*" ) "No \
    instalar mas paquetes"
  do
    case $RPM in
       "No instalar mas paquetes")
       clear
       PS3="Elija una opcion (ENTER para menu): "
       return 0
       ;;
       *)
       echo "Instalando el paquete $RPM"
       rpm -iv $RPM >> /tmp/rpm.log 2>> /tmp/rpm.log
       if [ ! $? ]
       then
         echo "El paquete no ha podido ser instalado."
         echo "Chequee el fichero /tmp/rpm.log para ver los motivos"
       fi
       ;;
    esac
  done
}

# FUNCION PARA FACILITAR INFORMACION DE PAQUETES INSTALADOS EN EL
# SISTEMA
function infoinst
{

# EL LISTADO DE PAQUETES PUEDE SER MUY LARGO DE FORMA QUE SE SOLICITA
# UNA REFERENCIA PARA BUSCAR EL NOMBRE EN DICHA LISTA
 echo -n "Introduzca una cadena que forme parte del paquete a \
     consultar: "
 read NOMBRE

# SE ASIGNA LA VARIABLE RPM A LA LISTA DE PAQUETES INSTALADOS, PARA
# EVALUARLA LUEGO CON UNA SENTENCIA SELECT

 PS3="Elija un paquete para consultar (ENTER para menu): "
 select RPM in $(rpm -qa | grep -i ".*$NOMBRE.*" ) "No consultar \
    mas paquetes"
 do
  case $RPM in
    "No consultar mas paquetes")
    clear
    PS3="Elija una opcion (ENTER para menu): "
    return 0
    ;;
    *)

# SE PUEDE ELEGIR ENTRE CONSULTAR LA INFORMACION DEL PAQUETE O LISTAR
# LOS FICHEROS QUE LO COMPONEN
```

```
     echo -n "Quiere consultar la informacion del paquete, o \
        listar los ficheros que contiene? (i/l): "
     read OPCION
     if [[ $OPCION = i* || $OPCION = I* ]]
     then
       rpm -qi $RPM
     else
       rpm -ql $RPM
     fi
    ;;
  esac
 done
}

# FUNCION PARA INFORMACION DE PAQUETES NO INSTALADOS QUE SE ENCUETREN
# EN UN DIRECTORIO. LA ESTRUCTURA ES SIMILAR A LA DE LA FUNCION
# ANTERIOR

function infonoinst
{
  DIR=/mnt/cdrom/suse/i586
  PS3="Elija un paquete para solicitar info (ENTER para menu): "

# EN ESTE CASO, ADEMAS, SE PREGUNTA POR EL DIRECTORIO DONDE SE
# ENCUENTRAN LOS PAQUETES

  echo -n "Los paquetes que se consultaran estan en el cdrom de \
     Linux?
(s/n): "
  read OPCION
  if [[ $OPCION = n* || $OPCION = N* ]]
  then
   echo -n "Indique el directorio que contiene los paquetes: "
   read DIR
  fi
  cd $DIR

# EL LISTADO DE PAQUETES PUEDE SER MUY LARGO DE FORMA QUE SE SOLICITA
# UNA REFERENCIA PARA BUSCAR EL NOMBRE EN DICHA LISTA
  echo -n "Introduzca una cadena que forme parte del paquete \
     a consultar: "
  read NOMBRE
  select RPM in $(ls -1 *.rpm | grep -i ".*$NOMBRE.*") "No consultar \
     mas paquetes"
  do
    case $RPM in
       "No consultar mas paquetes")
       clear
       PS3="Elija una opcion (ENTER para menu): "
       return 0
       ;;
       *)
       echo -n "Quiere consultar la informacion del paquete, o \
           listar los ficheros que contiene? (i/l): "
       read OPCION
       if [[ $OPCION = i* || $OPCION = I* ]]
       then
         rpm -qip $RPM
       else
         rpm -qlp $RPM
       fi
       ;;
    esac
 done
```

```
}

# MENU PRINCIPAL DEL SCRIPT.

PS3="Elija una Opcion (ENTER para menu): "
export PS3
clear

# SE USA UNA OPCION SELECT PARA PRESENTAR EL MENU. LA PREGUNTA QUE
# HACE LA OPCION SELECT VIENE DEFINIDA POR LA VARIABLE PS3 QUE SE HA
# ASIGNADO ANTERIORMENTE

select OPCION in "Instalar un paquete" "Desinstalar un paquete"
"Informacion de paquetes no instalados" "Informacion de paquetes
instalados" "Salir"

do
  case $OPCION in
  "Instalar un paquete")
   instala
   ;;
  "Desinstalar un paquete")
   desinstala
   ;;
  "Informacion de paquetes no instalados")
   infonoinst
   ;;
  "Informacion de paquetes instalados")
   infoinst
   ;;
  "Salir")
   clear
   exit 0
   ;;
   *)
   echo "Esa no es una opcion adecuada"
   ;;
  esac
done
```

6.17. Petición de Usuario y Contraseña Para Scripts:

login.sh

```
#!/bin/bash
#######################################################################
# ESTE SCRIPT SIMULA LA OPERACIÃ DE INTRODUCIR UN LOGIN Y UNA
# PASSWORD. PODRIA USARSE EN CUALQUIER SCRIPT QUE NECESITE UNA
# VERIFICACION DE USUARIO, AUNQUE TAMBIEN PUEDE FUNCIONAR COMO "BOMBA
# LOGICA" PUESTA SOBRE UNA TERMINAL SERIE EN ESPERA DE QUE UN INCAUTO
# ADMINISTRADOR INTENTE ENTRAR CON DICHOS PRIVILEGIOS. ESTE TIPO DE
# ATAQUE DE SEGURIDAD ERA MUY COMUN PERO YA ESTA MUY SUPERADO.
#
# EJEMPLO DE USO:
#
# # ./login (SE LIMPIA LA PANTALLA Y APARECE EL LOGIN)
#  login: root
#  password: (SE ESCBRIBE EN INVISIBLE)
#  Tu login es root y tu password es farola
#
#######################################################################
```

```
# ANTES DE SEGUIR EJECUTANDOSE IGNORA LAS SENALES TERM, INT Y QUIT
# PARA EVITAR UNA PARADA INTENCIONADA DEL SCRIPT POR PARTE DEL
# USUARIO.

trap "" 15 2 3
clear

# PONE EL CURSOR INVISIBLE

tput civis

# PIDE EL LOGIN Y LO LEE EN LA VARIABLE "LOGIN"

echo -n "login: "
read LOGIN

# PIDE LA PASSWORD Y LO LEE EN LA VARIABLE "PASSWORD"

echo -n "password: "

# DESACTIVA EL ECO PARA QUE NO SE LEA MIENTRAS SE ESCRIBE LA PASSWORD

stty -echo
read PASSWORD

# SE ACTIVA EL ECO DE NUEVO

stty echo

# ESTAS LINEAS SIMPLEMENTE VERIFICAN QUE LA OPERACION HA SIDO CORRECTA
# Y PUEDEN SER ELIMINADAS DE UN SCRIPT QUE USE ESTAS OPERACIONES.

echo
echo Tu login es $LOGIN y tu password es $PASSWORD

# VUELVE A PONER VISIBLE EL CURSOR DE LA TERMINAL
tput cnorm

# VUELVE A PERMITIR LAS SEÂ¥ALES TERM INT Y QUIT
trap  15 2 3
```

6.18. Eliminación de Segmentos de Memoria Compartida en el Sistema: `rm_shmem.sh`

```
#!/bin/bash
#########################################################################
# ESTE SCRIPT ELIMINA LOS SEGMENTOS DE MEMORIA COMPARTIDA QUE QUEDAN
# SIN NINGUN PROCESO CONECTADO A ELLOS. ESTE CASO SE DA CUANDO UN
# PROCESO TERMINA DE FORMA ANORMAL Y NO ELIMINA DICHOS SEGMENTOS, QUE
# PUEDEN TENER UN TAMAÑO CONSIDERABLE. ESTA TAREA SE PUEDE PROGRAMAR
# CON CRON PARA REALIZARLA PERIODICAMENTE.
#
# EJEMPLO DE USO:
#
# #ipcs -m
#  ------ Shared Memory Segments --------
#  key        shmid    owner   perms    bytes      nattch     status
#  0x00000000 32768    root    777      312        0 (SEG VACIO) dest
#  0x00000000 819201   root    600      262144     1             dest
```

```
# #
# # ./rm_shmem.sh
# Borrado el segmento: 32768
# #
# #ipcs -m
#  ------ Shared Memory Segments --------
#  key        shmid    owner   perms    bytes      nattch     status
#  0x00000000 819201   root    600      262144     1          dest
#
#########################################################################

# EJECUTA EL COMANDO ipcs PARA VER LOS SEGMENTOS DE MEMORIA
# COMPARTIDA. CON DICHA SALIDA ELIMINA LAS CABECERAS Y EXTRAE SOLO LAS
# COLUMNAS CON LOS ID. DE MEMORIA Y EL NUMERO DE PROCESOS CONECTADOS A
# DICHO SEGMENTO. LA SALIDA ES DEPOSITADA EN UN FICHERO TEMPORAL.

ipcs -m | grep -v nattch | grep -v Segments | grep -v '^$' | \
      awk '{ print $2"\t"$6}' > /tmp/$$rm_sh_mem

# LEE EL FICHERO TEMPORAL Y EJECUTA ipcrm PARA ELIMINAR SOLO AQUELLOS
# SEGMENTOS QUE NO TIENEN ACTUALMENTE NINGUN PROCESO CONECTANDO A
# ELLOS.

while read ID NUM
do
  if (( NUM == 0 ))
  then
     ipcrm -m $ID
     echo " Borrado el segmento: $ID"
  fi
done < /tmp/$$rm_sh_mem

# ELIMINA EL FICHERO TEMPORAL

rm  /tmp/$$rm_sh_mem
```

6.19. Selección de la Shell Para una Sesión Interactiva de Trabajo: `selecciona_shell.sh`

```
#!/bin/bash
#########################################################################
# ESTE SCRIPT SE PUEDE PONER AL COMIENZO DE UN FICHERO DE PERFIL DE
# USUARIO PARA QUE EN SU SESION DE TRABAJO PUEDA USAR LA SHELL QUE LE
# INTERESE EN ESE MOMENTO. ES UN BUEN EJEMPLO DE BUCLE SELECT
# COMBINADO CON UNA SENTENCIA CONDICIONAL CASE. CUANDO EL USUARIO
# ACABA CON DICHA SHELL VUELVE AL MENU HASTA QUE DECIDE ACABAR LA
# SESION.
#
# EJEMPLO DE USO:
# # ./selecciona_shell.sh
#  1) sh
#  2) ksh
#  3) csh
#  4) bash
#  5) salir
#  Introduce una de las shells (ENTER para menu): 3
#  Cambiando a csh
# # ps -f
#  UID  PID   PPID  C STIME TTY   TIME CMD
```

```
#  root 5860  5856  0 16:17 pts/6 0:00:01 -bash
#  root 7361  5860  0 18:49 pts/6 00:00:00 /bin/bash
#       ./selecciona_shell.sh
#  root 7371  7361  9 18:49 pts/6 00:00:00 -bin/csh (CSH EN EJECUCION)
#  root 7399  7371  0 18:50 pts/6 00:00:00 ps -f
# # exit
#  exit
#  Introduce una de las shells (ENTER para menu):
#  1) sh
#  2) ksh
#  3) csh
#  4) bash
#  5) salir
#  Introduce una de las shells (ENTER para menu): 2
#  Cambiando a ksh
# # ps -f
#  UID  PID   PPID  C STIME TTY   TIME CMD
#  root 5860  5856  0 16:17 pts/6 00:00:01 -bash
#  root 7361  5860  0 18:49 pts/6 00:00:00 /bin/bash
#       ./selecciona_shell.sh
#  root 7404  7361  0 18:50 pts/6 00:00:00 /bin/ksh (KSH EN EJECUCION)
#  root 7410  7404  0 18:50 pts/6 00:00:00 ps -f
# # exit
#  Introduce una de las shells (ENTER para menu):
#  1) sh
#  2) ksh
#  3) csh
#  4) bash
#  5) salir
#  Introduce una de las shells (ENTER para menu): 5
#  Adios
#
#########################################################################

# LA VARIABLE "PS3" ES EL PROMPT DE SELECCION DE LA SENTENCIA SELECT"
PS3="Introduce una de las shells (ENTER para menu): "
select SHELL1 in sh ksh csh bash salir
do
case $SHELL1 in
        'ksh')
           echo "Cambiando a ksh"
           sleep 1
           /bin/ksh
           ;;
        'sh')
           echo "Cambiando a sh"
           sleep 1
           /bin/sh
           ;;
        'csh')
           echo "Cambiando a csh"
           sleep 1
           /bin/csh
           ;;
        'bash')
           echo "Cambiando a bash"
           sleep 1
           /bin/bash
           ;;
        'salir')
           echo "Adios"
           sleep 1
           exit
           ;;
```

```
        *)
           echo "Opcion incorrecta"
           sleep 1
           ;;
esac
done
```

6.20. Agenda Personal: `agenda.sh`

```
#!/bin/bash
#########################################################################
# ESTE SCRIPT IMPLEMENTA UNA SENCILLA PERO PRACTICA AGENDA PARA LA
# CUENTA DE UN USUARIO EN UN SISTEMA UNIX. PERMITE DAR NOMBRES DE
# ALTA, BAJA, MODIFICAR O VER NOMBRE Y TELEFONO.
#
# EJEMPLO DE USO:
# ./agenda.sh (SE LIMPIA LA PANTALLA Y APARECE EL MENU)
#  1) Ver           3) Baja          5) Salir
#  2) Alta          4) Modificacion
#  Selecciona una Opcion (ENTER para menu): 2
#  Nombre y Apellidos: Jose Lopez Vera
#  Edad: 45
#  Tlf: 234543567
#  Selecciona una Opcion (ENTER para menu):
#  1) Ver           3) Baja          5) Salir
#  2) Alta          4) Modificacion
#  Selecciona una Opcion (ENTER para menu): 1
#  Nombre: Jose Lopez Vera
#  Edad: 45
#  Tlf: 234543567
#
#  Selecciona una Opcion (ENTER para menu): 4
#  Nombre a modificar: Jose Lopez Vera
#  Nombre (Jose Lopez Vera): Jose Lopez Canca
#  Edad (45): 23
#  Tlf (234543567): 654234654
#  Selecciona una Opcion (ENTER para menu):
#  1) Ver           3) Baja          5) Salir
#  2) Alta          4) Modificacion
#  Selecciona una Opcion (ENTER para menu): 1
#  Nombre: Jose Lopez Canca
#  Edad: 23
#  Tlf: 654234654
#
#########################################################################

#INICIALIZACION DE VARIABLES

DATOS="$HOME/agenda.txt"
OPCIONES_MENU="Ver Alta Baja Modificacion Salir"
PS3="Selecciona una Opcion (ENTER para menu): "

clear
select OPCION in $OPCIONES_MENU
do
  case $OPCION in

   # VER INFORMACION DE UN REGISTRO

    Ver)
```

```
    while read LINEA
    do
      NOMBRE=$(echo $LINEA | cut -d: -f1)
      EDAD=$(echo $LINEA | cut -d: -f2)
      TLF=$(echo $LINEA | cut -d: -f3)
      echo "Nombre: $NOMBRE"
      echo "Edad: $EDAD"
      echo "Tlf: $TLF"
      echo
    done < $DATOS
      ;;

 # DAR DE ALTA UN REGISTRO

  Alta)
    echo -n "Nombre y Apellidos: "
    read NOMBRE
    echo -n "Edad: "
    read EDAD
    echo -n "Tlf: "
    read TLF
    echo "${NOMBRE}:${EDAD}:${TLF}" >> $DATOS
    ;;

# DAR DE BAJA UN REGISTRO

  Baja)
    echo -n "Nombre a borrar: "
    read NOMBRE
    grep -v "^${NOMBRE}:" $DATOS > /tmp/$$
    mv -f /tmp/$$ $DATOS
    ;;

# MODIFICAR UN REGISTRO

  Modificacion)
    echo -n "Nombre a modificar: "
    read NOMBRE
    LINEA=$(grep "^${NOMBRE}:" $DATOS)
    NOMBRE=$(echo $LINEA | cut -d: -f1)
    EDAD=$(echo $LINEA | cut -d: -f2)
    TLF=$(echo $LINEA | cut -d: -f3)
    echo -n "Nombre ($NOMBRE): "
    read NUEVO_NOMBRE
    echo -n "Edad ($EDAD): "
    read NUEVO_EDAD
    echo -n "Tlf ($TLF): "
    read NUEVO_TLF
    if [ -z "$NUEVO_NOMBRE" ]; then
      NUEVO_NOMBRE=$NOMBRE
    fi
    if [ -z "$NUEVO_EDAD" ]; then
      NUEVO_EDAD=$EDAD
    fi
    if [ -z "$NUEVO_TLF" ]; then
      NUEVO_TLF=$TLF
    fi
    # SE GENERA UN FICHERO TEMPORAL SIN EL REGISTRO A MODIFICAR
    grep -v "^${NOMBRE}:" $DATOS > /tmp/$$
    # SE INCLUYE EL REGISTRO MODIFICADO
    echo "${NUEVO_NOMBRE}:${NUEVO_EDAD}:${NUEVO_TLF}" >> /tmp/$$
    mv -f /tmp/$$ $DATOS
    ;;
```

```
  # SALIR DE LA APLICACION

    Salir)
      rm -f /tmp/$$
      exit 0
      ;;
  esac
done
```

6.21. Creación de una Cuenta de Usuario:

nuevousuario.sh

```
#!/bin/bash
#################################################################
# ESTE SCRIPT PRETENDE SIMPLIFICAR LA TAREA DE AÂ¥ADIR UN NUEVO
# USUARIO. PARA ELLO HACE POR EL ADMINISTRADOR LAS COMPROBACIONES DE
# SI EL NOMBRE TIENE LA LONGITUD ADECUADA, TIENE LOS CARACTERES
# ADECUADOS, SI LOS GRUPOS A LOS QUE QUIERE AGREGAR EL USUARIO
# EXISTEN, Y FINALMENTE DESPUES DE AGREGAR AL USUARIO TAMBIEN PREGUNTA
# POR LA PASSWORD DE ESTE.
#
# EJEMPLO DE USO:
#
# # ./nuevousuario.sh
#  Login: pepe
#  Grupo Primario: users
#  Lista de Grupos Secundarios (Con Espacios): tomcat icecream
#  Descripcion: usuario de prueba
#  Shell: /bin/bash
#  Usuario pepe agregado con exito
#  Agregando password para el usuario pepe
#  Changing password for pepe.
#  New Password:
#  Reenter New Password:
#  Password changed.
#  Desea que el usuario cambie su password en el siguiente login?
[S/n]: s
#  Password expiry information changed.
# #
# # grep pepe /etc/passwd
#  pepe:x:1001:100:usuario de prueba:/home/pepe:/bin/bash
#
#################################################################

# PREGUNTA POR EL LOGIN Y COMPRUEBA QUE CUMPLE LOS REQUISITOS

echo -n "Login: "
read LOGIN

if [ -z "$LOGIN" ]; then
  echo "Nombre vacio"
  exit 1
fi

if [ ${#LOGIN} -gt 8 ]; then
  echo "Nombre demasiado largo, se trunca a 8 caracteres"
  LOGIN=$(echo $LOGIN | cut -c1-8)
  echo "Login truncado: $LOGIN"
fi
```

```
if echo "$LOGIN" | grep "[^0-9a-zA-Z]" > /dev/null
then
  echo "Nombre con caracteres invalidos"
  exit 1
fi

if grep "^$LOGIN:" /etc/passwd > /dev/null
then
  echo "El nombre existe"
  exit 1
fi

# PREGUNTA POR LOS GRUPOS PRIMARIO Y SECUNDARIOS Y COMPRUEBA QUE
# EXISTEN

echo -n "Grupo Primario: "
read GID

echo -n "Lista de Grupos Secundarios (Con Espacios): "
read GRUPOS


for GRUPO in $GID $GRUPOS
do
  LINEA=$(grep "^$GRUPO:" /etc/group)
  if [ ! -n "$LINEA" ]
  then
    echo "El grupo no $GRUPO existe"
    exit 1
  fi
done

# CONVIERTE LA CADENA DE GRUPOS PARA QUE ESTEN SEPARADOS POR COMAS
GRUPOS=$(echo $GRUPOS | tr " " ",")

# PREGUNTA POR LA DESCRIPCION

echo -n "Descripcion: "
read DESCRIPCION

# PREGUNTA POR LA SHELL PARA ESTE USUARIO Y COMPRUEBA QUE ES VALIDA

echo -n "Shell: "
read SHELL
until [ -x "$SHELL" ]; do
  echo "$SHELL no es un shell valido para el usuario"
  echo -n "Introduzca un nuevo shell: "
  read SHELL
done

# COMPRUEBA CUAL ES EL UID ADECUADO PARA EL USUARIO

MIN_UID=100
MAX_UID=60000
USER_UID=$MIN_UID

for UID in $(cut -d: -f3 /etc/passwd)
do
  if [ $UID -gt $USER_UID ] && [ $UID -lt $MAX_UID ]; then
    USER_UID=$UID
  fi
done 2> /dev/null
USER_UID=$((USER_UID+1))
```

```
# AGREGA EL USUARIO Y DICE SI HA TENIDO EXITO O NO LA OPERACION

if useradd -g $GID -G "$GRUPOS" -m -d /home/$LOGIN -c "$DESCRIPCION" -
s $SHELL $LOGIN
then
   echo "Usuario $LOGIN agregado con exito"
else
   echo "Error al agregar el usuario $LOGIN"
   exit 1
fi

# AGREGA UNA PASSWORD AL USUARIO Y PREGUNTA SI DEBE CAMBIARLA EL MISMO

echo "Agregando password para el usuario $LOGIN"
passwd $LOGIN

echo -n "Desea que el usuario cambie su password en el siguiente
login? [S/n]: "
read RESP

if [[ ! -n $RESP || $RESP == S* || $RESP == s* ]]
then
  passwd -e $LOGIN
fi
```

6.22. Listado de los Usuarios de un Sistema:

usuarios.sh

```
#!/bin/bash
#########################################################################
#
# ESTE SCRIPT EXTRAE UNA LISTA CON TODOS LOS USUARIOS DEL SISTEMA Y SU
# DESCRIPCION. PARA ELLO HACE USO DE MODIFICAR LA VARIABLE "IFS"
# GRACIAS A LO CUAL PUEDE LEER CADA CAMPO DEL FICHERO /ETC/PASSWD DE
# FORMA SEPARADA.
#
# EJEMPLO DE USO:
#
# # ./usuarios.sh
#  Nombre     Descripcion
#  root       root
#  bin        bin
#  daemon     Daemon
#  ...
#  icecream   Icecream Daemon
#  tomcat     Tomcat - Apache Servlet/JSP Engine
#  beagleindex User for Beagle indexing
#  nobody     nobody
#  pedrolopezs Pedro Lopez
#
#########################################################################

IFS=":"
exec 3< /etc/passwd
printf "%-10s %-30s\n" "Nombre" "Descripcion"

read <&3 NAME PASS UID GID COMMENT DIR SHELL 2> /dev/null
while [[ -n $NAME ]]
do
```

```
  printf "%-10s %-30s\n" $NAME $COMMENT
  read <&3 NAME PASS UID GID COMMENT DIR SHELL 2> /dev/null
done
exec 3<&-
```

6.23. Ejemplo de Sentencia `getopts`: `opciones.sh`

```
#!/bin/bash

#####################################################################
#
# TENEMOS AQUI UN EJEMPLO DE USO DE OPCIONES CON GETOPTS. ESTE EJEMPLO
# CONCRETO ADMITE LINEAS DE COMANDO DEL TIPO:
# comando -a -b -c argumento_de_c -d -e argumento_de_e -f -g
# LA ESTRUCTURA CASE SIMPLEMENTE MUESTRA LAS OPCIONES PASADAS Y LOS
# ARGUMENTOS, PERO PUEDE ADAPTARSE A CUALQUIER USO PRACTICO DE UN
# COMANDO QUE LA NECESITE.
#
# EJEMPLO DE USO:
#
# # ./opciones.sh  -c arg1 -d -e arg2 -b -a -g -f arg3 arg4
#  Opcion c Argumento arg1 INDICE 3
#  Opcion d INDICE 4
#  Opcion e Argumento arg2 INDICE 6
#  Opcion b INDICE 7
#  Opcion a INDICE 8
#  Opcion g INDICE 9
#  Opcion f INDICE 10
#  INDICE ACTUAL: 10
#  ARGUMENTOS A DESPLAZAR: 9
#  Resto de la linea: arg3 arg4
#
#####################################################################

while getopts ":abc:de:fg" OPCION
do
  case $OPCION in
    a)
     echo Opcion a INDICE $OPTIND
     ;;
    b)
     echo Opcion b INDICE $OPTIND
     ;;
    c)
     echo Opcion c Argumento $OPTARG INDICE $OPTIND
     ;;
    d)
     echo Opcion d INDICE $OPTIND
     ;;
    e)
     echo Opcion e Argumento $OPTARG INDICE $OPTIND
     ;;
    f)
     echo Opcion f INDICE $OPTIND
     ;;
    g)
     echo Opcion g INDICE $OPTIND
     ;;
  esac
done
```

```
# UNA VEZ PASADAS TODAS LAS OPCIONES, EL RESTO DE LA CADENA SE ASUME
# QUE SON ARGUMENTOS.

echo INDICE ACTUAL: $OPTIND
echo ARGUMENTOS A DESPLAZAR: $(($OPTIND - 1))
shift $(($OPTIND - 1))
echo Resto de la linea: $*
```

6.24. Distribución de Ficheros en Múltiples Servidores: `copia_array_remoto.sh`

```
#!/bin/ksh
#######################################################################
# ATENCION, ESTE SCRIPT USA KSH YA QUE CIERTAS CARACTERISTICAS DE
# TYPESET SON ESPECIFICAS DE ESTA SHELL
#######################################################################

#######################################################################
#
# CON ESTE SCRIPT PODEMOS COPIAR UN FICHERO EN MULTIPLES MAQUINAS UNIX
# DESDE LA MAQUINA LOCAL, PARA ELLO SE ASUME QUE TODAS SE ENCUENTRAN
# EN UNA MISMA SALA Y SE PUEDEN RESOLVER LAS DIRECCIONES DE DICHAS
# MAQUINAS UNIX POR NOMBRES CONSISTENTES EN EL NOMBRE DE LA SALA Y UN
# NUMERO DE ORDEN CON DOS DIGITOS. LA COPIA SE REALIZA MEDIANTE RCP
# POR LO QUE PREVIAMENTE DEBEN CONFIAR LAS MAQUINAS CLIENTES EN LA QUE
# ENVIA LA COPIA MEDIANTE EL FICHERO /.rhosts
#
# EJEMPLO DE USO:
#
# # ./copia_array_remoto.sh
#  Dime el nombre de la sala: GOYA
#  Dame el numero de maquinas: 4
#  Dime el diretorio donde quieres hacer la copia: /tmp
#  Que fichero quieres copiar?: datos.txt
#  Copia correcta en GOYA01
#  Copia correcta en GOYA02
#  Copia correcta en GOYA03
#  Copia correcta en GOYA04
#
#######################################################################

echo -n "Dime el nombre de la sala: "
read SALA
echo -n "Dame el numero de maquinas: "
read MAQUINAS
echo -n "Dime el diretorio donde quieres hacer la copia: "
read DIR
echo -n "Que fichero quieres copiar?: "
read FICHERO
typeset -Z2 NUM
NUM=1

# SE CREA UN ARRAY CON LOS NOMBRES DE LAS MAQUINAS

while (( NUM <= MAQUINAS ))
do
        HOST[$NUM]=$SALA$NUM
        (( NUM = NUM + 1 ))
done
```

```
# SE REALIZA LA COPIA DEL FICHERO A CADA MAQUINA

for i in ${HOST[*]}
do
       rcp $FICHERO $i:$DIR/$FICHERO && echo "Copia correcta en $i"
done
```

6.25. Kit Para Implantación de Aplicaciones en Servidores en Cluster: `clustertoolkit.sh`

```
#!/bin/bash
#########################################################################
#
# UN CLUSTER DE ALTA DISPONIBILIDAD ES UNA ASOCIACION DE VARIOS
# SERVIDORES QUE COMPARTEN UNA INSTANCIA DE UNA APLICACION (POR
# EJEMPLO ORACLE), LAS DIRECCIONES IP DE SERVICIO Y EL ALMACENAMIENTO
# ASOCIADO A ESTA. ESTOS RECURSOS SE AGRUPAN EN UN "PAQUETE" QUE PUEDE
# SER EJECUTADO EN UN NODO Y EN CASO DE FALLAR, SER PASADO A OTRO EN
# CUESTION DE SEGUNDOS. LOS PRODUCTOS COMERCIALES DE ALTA
# DISPONIBILIDAD (SERVICEGUARD, SUN CLUSTER, VERITAS CLUSTER, ETC)
# USAN TOOLKITS QUE BASICAMENTE SON SCRIPTS PREFABRICADOS DE FORMA
# GENERICA Y QUE SE PUEDEN ADAPTAR PARA CREAR UN PAQUETE CON CUALQUIER
# APLICACION ESTE SCRIPT EMULA BASICAMENTE LAS FUNCIONALIDADES DE
# ESAS TOOLKITS. RELLENANDO LAS VARIABLES QUE HAY AL PRINCIPIO ES
# CAPAZ DE REALIZAR LAS ACCIONES DE ARRANQUE, PARADA Y MONITORIZACION
# DE LA APLICACION QUE QUERAMOS CONVERTIR EN PAQUETE.
#
# EJEMPLO DE USO:
#
# LAS VARIABLES DE CABECERA PSE RELLENARIAN DE LA SIGUIENTE FORMA
# PARA UN SERVIDOR WEB APACHE
#
# INSTANCIA=apache1
# APLICACION=httpd
# APL_PATH=/var/httpd
# APL_ARG=
# INTERVALO=10
# LOGFILE=/etc/cmcluster/apache1/apache1.cntl.log
#
# LUEGO EN EL SCRIPT DE CONTROL apache1.cntl DEBERA SER INVOCADO DONDE
# CORRESPONDA DE LA SIGUIENTE FORMA:
#
# INICIAR LA APLICACION:     clustertoolkit.sh start
#
# PARAR LA APLICACION:       clustertoolkit.sh stop
#
# MONITORIZAR LA APLICACION: clustertoolkit.sh monitor
#
#########################################################################

# LISTA DE VARIABLES GENERICAS PARA LANZAR LA APLICACION EN ESTE CASO
# CONCRETO SE LANZA UN COMANDO sleep 2000
INSTANCIA=prueba
APLICACION=sleep
APL_PATH=/bin
APL_ARG=2000
INTERVALO=30
LOGFILE=/tmp/PRUEBALOG
```

```
# EL SCRIPT PUEDE SER LANZADO CON START, STOP O MONITOR
case $1 in

# FUNCION DE ARRANQUE
   "start")
     if [ -f "$APL_PATH/$APLICACION" ]
     then
           $APL_PATH/$APLICACION $APL_ARG >> $LOGFILE 2>&1 &
           echo $! > /tmp/$INSTANCIA.$APLICACION.pid
           echo $(date +"%D - %H:%M") "Instancia $INSTANCIA \
              de la aplicacion $APLICACION iniciada en el nodo \
              $(hostname)" >> $LOGFILE
     else
           echo $(date +"%D - %H:%M") "La aplicacion $APLICACION \
              no esta instalada en este nodo y no pudo \
              iniciarse" >> $LOGFILE
           exit 1
     fi
     ;;

# FUNCION DE PARADA
   "stop")
     if [ -f "/tmp/$INSTANCIA.$APLICACION.pid" ] && ps -ef | awk \
        '{print $2}' | grep -q $(cat /tmp/$INSTANCIA.$APLICACION.pid)
     then
         kill $(cat /tmp/$INSTANCIA.$APLICACION.pid)
         rm /tmp/$INSTANCIA.$APLICACION.pid
         echo $(date +"%D - %H:%M") "Instancia $INSTANCIA de la \
             aplicacion $APLICACION finalizada en el nodo \
             $(hostname)" >> $LOGFILE
     else
         echo $(date +"%D - %H:%M") "La Instancia de la aplicacion \
             no se esta ejecutando en este nodo y no pudo \
             detenerse" >> $LOGFILE
     fi
     ;;

# FUNCION DE MONITORIZACION
   "monitor")
     while true
     do
       if [ -f "/tmp/$INSTANCIA.$APLICACION.mantenimiento" ]
       then
           sleep $INTERVALO
             continue
       else
          if ps -ef | tr -s " " | cut -f3 -d" " | grep -q $([ -f \
             /tmp/$INSTANCIA.$APLICACION.pid ] && cat \
             /tmp/$INSTANCIA.$APLICACION.pid) > /dev/null 2>&1
            then
                  sleep $INTERVALO
            else
                  echo $(date +"%D - %H:%M") "La Instancia de la \
                      aplicacion no se esta ejecutando en este \
                      nodo" >> $LOGFILE
                  echo $(date +"%D - %H:%M") "Monitor para la \
                      instancia $INSTANCIA de la aplicación \
                      $APLICACION en $(hostname) \
                      finalizado" >> $LOGFILE
                  rm /tmp/$INSTANCIA.$APLICACION.pid > /dev/null 2>&1
                   exit 1
            fi
       fi
     done
```

```
     ;;

# EN CASO DE NO SER INVOCADO ADECUADAMENTE LANZA ESTE MENSAJE
   *)
     echo "Uso de $0: $0 (start|stop|monitor)" >> $LOGFILE
     ;;
Esac
```

6.26. Conversión de Ficheros en Formato .lj a Postscript y a Formato Pdf: `lj-ps-pdf.sh`

```
#!/bin/bash
#######################################################################
#
# ESTE SCRIPT SE CREO PARA CONVERTIR FICHEROS CON EXTENSION .lj (PARA
# SER IMPRESOS POR UNA LASERJET HP) AL FORMATO .ps (POSTSCRIPT) Y DE
# AHI A FORMATO .pdf (ACROBAT). OBSERVANDO ESTOS FORMATOS SOLAMENTE SE
# DIFERENCIAN EN QUE EL FICHERO LJ TIENE TRES LINEAS ADICIONALES EN LA
# CABECERA CON INFORMACION PARA LA IMPRESORA. POR LO TANTO LA
# CONVERSION DE FORMATO SIMPLEMENTE CONSISTE EN ELIMINAR DICHAS
# LINEAS. UNA VEZ CONSEGUIDOS ESTOS FICHEROS SE USA UNA HERRAMIENTA
# ESTANDAR (ps2pdf) PARA PASAR DICHOS FICHEROS A FORMATO PDF.
#
# EJEMPLO DE USO:
#
# # ./lj-ps-pdf.sh
#  Directorio de trabajo: pruebalj
#  convirtiendo file1.lj en file1.ps... hecho
#  convirtiendo file2.lj en file2.ps... hecho
#  ...
#  convirtiendo file1.ps en file1.pdf... hecho
#  convirtiendo file2.ps en file2.pdf... hecho
#  ...
#
#######################################################################

#DIRECTORIO DONDE SE TRABAJARA
echo -n "Directorio de trabajo: "
read DIR
if [ -d $DIR ]
then
  cd $DIR
else
  echo "El directorio no existe"
  exit 1
fi

# CONVERSION DE LJ A PS
for i in *.lj
do
# EL COMANDO BASENAME CORTA UNA CADENA HASTA LA SECUENCIA INDICADA
 nombre=$(basename $i .lj).ps
 echo -n "convirtiendo $i en $nombre... "
# SE ELIMINAN LAS TRES PRIMERAS LINEAS CON UN COMANDO SED
 sed '1,3d' $i > $nombre
 echo "hecho"
done

# CONVERSION DE PS A PDF
for i in *.ps
```

```
do
 nombre=$(basename $i .ps).pdf
 echo "convirtiendo $i en $nombre... "
 ps2pdf $i $nombre
 echo "hecho"
done
```

6.27. Generación de un Informe de Totales:

formatea.awk

```
#!/usr/bin/awk -f
#####################################################################
#
# ESTE SCRIPT ESTA ESCRITO INTEGRAMENTE EN LENGUAJE AWK POR LO QUE LA
# PRIMERA LINEA INDICA QUE DEBE EJECUTARSE CON DICHO COMANDO. ADMITE
# UN FICHERO DE ENTRADA CON CINCO COLUMNAS SEGUN EL SIGUIENTE FORMATO:
#
#       NOMBRE APELLIDO1 APELLIDO2 CIUDAD SALARIO BENEFICIOS
#
# Y GENERA UNA SALIDA FORMATEADA CON CABECERAS, CADA CAMPO
# ENCOLUMNADO, Y DOS LINEAS DE TOTALES CON LOS CAMPOS NUMERICOS.
#
# EJEMPLO DE USO:
#
# # cat listado.txt
#  Pedro Fernandez Perez Madrid 23000 8000
#  Jaime Martin Leal Malaga 24500 6000
#  Juana Luque Garcia Malaga 23200 4500
#  Antonio Bernal Mendez Madrid 24500 3200
#  Maria Luque Moran Madrid 30000 8000
#  Juan Atienza Martinez Malaga 24000 4000
#  Antonio Lopez Marques Malaga 23500 3000
# #
# # ./formatea.awk listado.txt
#  Nombre     Apellido1  Apellido2   Salario Beneficios
#
#  Pedro      Fernandez  Perez         23000       8000
#  Jaime      Martin     Leal          24500       6000
#  Juana      Luque      Garcia        23200       4500
#  Antonio    Bernal     Mendez        24500       3200
#  Maria      Luque      Moran         30000       8000
#  Juan       Atienza    Martinez      24000       4000
#  Antonio    Lopez      Marques       23500       3000
#  ------------------------------------------------
#  Total Salarios: 172700
#  Total Beneficios
#  Madrid: 19200 Malaga: 17500
#
#####################################################################

BEGIN {print "Nombre     Apellido1  Apellido2   Salario Beneficios"}
BEGIN {print "" ; TSAL=0 ; BENMD=0 ; BENMA=0 }
$4 ~ /Madrid/   {printf "%-11s%-11s%-11s%8d%11d\n",$1,$2,$3,$5,$6
        TSAL+=$5
       BENMD+=$6}
$4 ~ /Malaga/   {printf "%-11s%-11s%-11s%8d%11d\n",$1,$2,$3,$5,$6
       TSAL+=$5
       BENMA+=$6}
END {print "------------------------------------------------"}
END {print "Total Salarios: " TSAL }
END {print "Total Beneficios"}
```

```
END {print "Madrid: " BENMD " Malaga: " BENMA }
```

CAPÍTULO 7. CONCLUSIONES

En el presente proyecto hemos hecho un recorrido en los ámbitos histórico y práctico del uso de las diferentes shells en los entornos Unix/Linux. Ha quedado patente la necesidad que tiene un administrador de conocer a fondo el lenguaje de programación de la o las shells de que disponga su entorno operativo concreto.

Como hemos visto en el capítulo tres, estas shells tienen notables diferencias entre ellas. Por fortuna, dichas diferencias en la mayoría de los casos se pueden salvar usando la sintaxis más simple, ya que presentan compatibilidad hacia atrás. Donde puede presentarse mayor problemática a la hora de una migración entre entornos, por ejemplo entre HP-UX y Solaris, suele ser a nivel de los comandos de administración que ambos tienen pero esa problemática no es la estudiada en el presente proyecto, algo que sí podría ser objeto de estudio en futuros proyectos.

En cuanto al uso de `sed` y `awk` en los scripts, hemos dejado patente la potencia que ambos comandos tienen a la hora de procesar documentos. Son capaces de tomar un fichero de entrada y generar una salida formateada según las necesidades del usuario, o de extraer aquella información que sea necesaria, incluso efectuando cálculos con los datos del documento.

Una futura línea de investigación para proyectos podría ser, tomando esta idea como base, y aprovechando que `sed` y `awk` son herramientas estándar del sistema operativo, la posibilidad de generar un sistema de bajo coste de formateo y extracción automática de información sobre ficheros de datos, usando exclusivamente scripts basados en dichos comandos.

También hemos visto ejemplos concretos aplicados a la administración. Por supuesto que podríamos haber aportado código mucho más extenso, pero se decidió en principio usar programas cortos y sencillos que no sólo fueran prácticos sino además suficientemente documentados en su código de las técnicas y sintaxis explicadas tanto en el capítulo tres como en el capítulo cuatro.

La principal aportación de este proyecto ha sido la de establecer una comparativa entre la sintaxis de las distintas shells más utilizadas hoy en día. Algo que puede ser determinante a la hora de decidirse entre una u otra, siempre y cuando las tengamos disponibles en nuestro entorno operativo.

El estudio comparativo desarrollado permite deducir que salvo causas mayores, debemos decantarnos por aquella shell que aporte mejores y más avanzadas estructuras sintácticas y mayores facilidades para el usuario. En ambos casos, y salvo pequeñas excepciones, se ha visto que la shell que tiene mayores ventajas es Bash, principalmente por tres características: es libre, es compatible hacia atrás con casi todas las otras shells, y presenta muchas facilidades interactivas para el usuario.

En cualquier caso, las shells estudiadas son las más tradicionales. Continuamente se desarrollan nuevos tipos de shells que aportan características adicionales, por lo tanto este estudio comparativo queda abierto a futuras ampliaciones basadas en las nuevas versiones que puedan salir de las shells vistas en él, o de nuevas shells que puedan aparecer.

En conclusión, se proponen tres líneas futuras de trabajo que pueden servir como continuación del presente proyecto:

- Sistemas de migración automática de scripts entre diferentes entornos operativos UNIX.
- Sistemas de bajo coste de extracción automática y análisis de datos de ficheros ascii basados exclusivamente en scripts con `sed` y `awk`.
- Ampliación del presente proyecto basado en nuevas versiones de las shells actuales y sus nuevas características, o en nuevas shells que puedan aparecer en un futuro.

BIBLIOGRAFÍA

[1] Evi Nemeth, Trent R. Hein, Scott Seebass, and Garth Snynder, *"Unix Systems Administration Handbook,"* Prentice-Hall, 2000.

[2] Aeleen Frisch, *"Essential System Administration Help for Unix System Administrators"* O'Reilly & Associates Inc., 1995.

[3] Stephen G. Kochan, and Patrick H. Wood, *"Unix Shell Programming,"* (2nd. edition), Sams, 1989.

[4] Cameron Newham, and Bill Rosenblatt, *"Learning the Bash Shell"* (2nd. edition), O'Reilly & Associates Inc., 1998.

[5] Bruce Blinn, *"Portable Shell Programming: An Extensive Collection of Bourne Shell Examples,"* Prentice-Hall, 1995.

[6] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, *"The Awk Programming Language,"* Addison-Wesley, 1988.

[7] Dale Dougherty, and Arnold Robbins, *"Sed & Awk,"* (2nd. edition), O'Reilly & Associates, 1997.

[8] Denis M. Ritchie, "The Evolution of the Unix Time-sharing System", *AT&T Bell Laboratories Technical Journal,* vol. 63 no. 6 Part 2, pp. 1577-93, October 1984.

[9] S. C. Johnson and B. W. Kernighan, *"The Programming Language B"*, Computer Science Technical Report #8, Bell Laboratories, Murray Hill NJ 1973.

[10] M. Richards, "BCPL: A Tool for Compiler Writing and Systems Programming", *Proceedings of the AFIPS SJCC* vol. 34, pp. 557-66, 1969

[11] B. W. Kernighan and D. M. Ritchie, *"The C Programming Language"*, Prentice-Hall, Englewood Cliffs NJ, 1978. Second Edition, 1979.

[12] D.M. Ritchie and K. Thompson, "The Unix Time-sharing System", *Communications of the ACM* vol. 17 no. 7, pp 365-37, July 1974.

[13] S. C. Johnson and D. M. Ritchie, "Portability of C Programs and the Unix System", Bell System Technical Journal vol. 57 no. 6, pp. 2021-48, July-August 1978

[14] S. R. Bourne, "The UNIX Shell", Bell Systems Technical Journal vol. 57 no. 6, pp. 19711990, July-August 1978.

[15] *"UNIX shell differences and how to change your shell"*, Usenet News comp.unix.shell unix-faq/shell/shell-differences version 1.17. (Publicada mensualmente en el grupo de noticias).

[16] William Joy, *"An Introduction to the C shell"*, (revised for 4.3BSD by Mark Seiden), disponible en http://docs.freebsd.org/44doc/usd/04.csh/paper.html

[17] G. Anderson and P. Anderson, *"The Unix C shell guide"*, Prentice-Hall, 1986.

[18] *Página oficial de Tcsh* disponible en http://www.tcsh.org

[19] Morris Bolksy and David Korn *"The New Korn Shell Command and Programming Language"*, Prentice Hall, 1995.

[20] *Página oficial del Portable Application Standards Committee (PASC)* disponible en http://www.pasc.org/

[21] David Korn, "*Documentación de Korn Shell"* disponible en http://www.kornshell.com/doc/

[22] Bill Rosenbl, *"Learning the Korn Shell"*, O'Reilly - 1st Edition January 1993 - 1-56592-054-6, Order Number: 0546.

[23] *Manual Online de HP-UX 11I* disponible en http://docs.hp.com/en/B2355-60105/index.html

[24] *Página oficial de GNU (GNU's Not Unix)* disponible en http://www.gnu.org

[25] *Página oficial de Bash Shell* disponible en http://cnswww.cns.cwru.edu/~chet/bash/bashtop.html

[26] Newham & Bill Rosenblat, *"Learning the bash Shell"*, Cameron - 2nd Edition January 1998 1-56592-347-2, Order Number: 3472

[27] Steve Bourne , "An introduction to the Unix Shell", (Revised for 4.3 BSD by Mark Seiden) disponible en http://rhols66.adsl.netsonic.fi/era/unix/shell.html (Documento generado con los fuentes originales de Bell Labs disponible en http://cm.bell-labs.com/7thEdMan/vol2/shell.bun)

[28] David Korn, *"Página de manual de Korn Shell"* disponible en http://www.cs.princeton.edu/~jlk/KornShell/doc/man88.html

[29] Free Software Foundation, Inc, "*Bash Reference Manual"* disponible en http://www.gnu.org/software/bash/manual/bashref.html

[30] David Korn, *" FAQ de Korn Shell",* disponible en http://www.kornshell.com/doc/faq.html

[31] Michael Rendell, *"Public Domain Korn Shell Man Page",* disponible en http://www.cs.mun.ca/~michael/pdksh/pdksh-man.html

[32] *"C Shell: Expresion Syntax",* disponible en http://www.decf.berkeley.edu/help/unix/csh/expressions.html

[33] "*Shells: Users Guide"*, HP Part nº B2355-90046 disponible en http://docs.hp.com

APÉNDICE A. CONTENIDOS DEL CD ADJUNTO

El CD que acompaña a la memoria contiene una serie de directorios que se detallan a continuación:

`D:\scripts` : Scripts presentados en el proyecto

`D:\memoria` : Memoria del proyecto en formato pdf y presentación powerpoint