La KornShell: Lenguaje de Programación y Comando

Una guía para programadores de Shell Scripts de Unix

Jesús Alberto Vidal Cortés

http://inicia.es/de/chube

chube@inicia.es

www.kornshell.com

Madrid, Agosto de 2001

Índice

m

PARTE I: INTRODUCCIÓN

1. SOBRE EL LENGUAJE KORNSHELL

El lenguaje KornShell fue diseñado y desarrollado por David G. Korn en los Laboratorios AT&T Bell. Es un **lenguaje de comandos interactivo** que proporciona acceso al sistema UNÍX y a otros muchos sistemas, en las muy distintas computadoras y workstations para las cuáles está implementado. El lenguaje KornShell es también un completo y potente **lenguaje de programación** de alto nivel para escribir aplicaciones, a menudo más fácilmente y rápidamente que con otros lenguajes de alto nivel. Esto lo hace especialmente apropiado para el prototipado.

¿QUÉ ES UNA SHELL?

Mucha de la excitación cuando emergió la Shell de UNIX, de los Laboratorios AT&T Bell hace cerca de 20 años, era debida al hecho de que este nuevo intérprete de comandos era también un lenguaje de programación. Primeramente, de la noche a la mañana la potencia de la programación se puso en manos de no-programadores, esto es, en manos de los usuarios. De hecho, este nuevo lenguaje fue llamado *Shell* para sugerir una capa externa que apuntaba a los servicios del sistema operativo, llamado un kernel, por debajo de la shell.

Un shell le permite al usuario interactuar con los recursos de la computadora tales como programas, ficheros y dispositivos. Un shell *interactivo* actúa como un intérprete de comandos. En su papel como intérprete de comandos, el shell es un interfaz entre el usuario y el sistema. El usuario le teclea comandos al shell y el shell los trata, normalmente mediante la ejecución de programas. Esta es la función principal de la shell para muchos usuarios. Algunos shells interactivos contienen una facilidad de editor de línea de comandos built-in para facilitarle al usuario la introducción y repetición de comandos introducidos.

La mayoría de las shells pueden ser utilizadas también como lenguajes de programación. Los usuarios pueden combinar secuencias de comandos para crear programas nuevos. Estos programas son conocidos como shell scripts. Los shell scripts automatizan el uso de la shell como un intérprete de comandos. Por ejemplo, si utilizas frecuentemente la misma secuencia de, digamos, 5 comandos de la shell, podrías escribir y ejecutar un fichero con estos cinco comandos para evitar tener que volver a teclearlos cada vez que los necesites. Cuando referencias a este fichero, la shell interpretará las líneas que tenga el fichero del mismo modo que si las tecleases en el terminal.

Algunas shells, como la kornshell, son, en efecto son un lenguaje de programación de alto nivel completo. El lenguaje Kornshell incluye variables, funciones, comandos built-in, y comandos de flujo de control (por ejemplo, comandos iterativos y condicionales). Por lo que, los scripts de Kornshell pueden ser programas de aplicación de la misma forma que lo son aquellos escritos en otros lenguajes de alto y medio nivel como Basic, Fortran o el lenguaje C.

La nueva versión de **ksh** presenta la funcionalidad de otros lenguajes de scripts tales como **awk**, **icon**, **perl**, **rexx**, y **tcl**. Por ésta y otras muchas razones **ksh** es un lenguaje de scripts mucho mejor que ninguna de las otras shells populares.

BENEFICIOS DEL USO DE KSH

Existe un único lenguaje **Kornshell**, pero puedes utilizarlo como un lenguaje de comandos y/o como un lenguaje de programación. Utilizamos el término **comandos** para describir lo que tecleas en el terminal para que se ejecute de forma inmediata. Utilizamos el término **script** para describir programas que escribes en el lenguaje Kornshell y pones dentro de un fichero para una ejecución posterior.

Ksh es muy portable y se ejecuta en muchos sistemas diferentes. Esto hace posible que programas escritos para un sistema sean ejecutados sin tener que ser cambiados en otros muchos sistemas. También les hace posible a los programadores que estén familiarizados con **ksh** usar nuevos sistemas sin tener que pasar por largos periodos de aprendizaje, y usar varios sistemas a la vez sin la confusión causada cuando cada sistema presenta un interfaz distinto.

Comparados con otras shell, **ksh** tiene beneficios para ser usado como lenguaje de comandos y beneficios para ser usado como lenguaje de programación. Estos beneficios se resumen más adelante.

Mejoras como Lenguaje de Comandos

Edición de la línea de comando. Volver a teclear un comando es una tarea tediosa, que consume tiempo y que nos conduce a cometer errores, especialmente si se necesitan hacer los cambios varias veces. Una ventaja mayor de **ksh** es que presenta unas interfaces similares a a **emacs** o **vi** para la edición de la línea de comando actual. Por lo que con **ksh** no es necesario eliminar hacia atrás hasta el punto donde necesitamos hacer el cambio. Los usuarios de **ksh** tienden a coger el hábito de editar sus comandos actuales antes de pulsar RETURN. Se puede utilizar la misma interfaz para modificar comandos introducidos anteriormente y que **ksh** guarda en el fichero histórico.

Mecanismo de histórico de comandos. **Ksh** guarda un fichero histórico que guarda los comandos que introduces. El fichero histórico puede ser accedido a través de directivas del editor **emacs** o **vi**, o a través del comando built-in **hist**. El fichero histórico se mantiene a través de distintas sesiones de conexión o login, y puede ser compartido por varias instancias de **ksh** simultáneas. Por lo que un comando previo puede ser modificado y reintroducido con un esfuerzo mínimo.

Completar el nombre de comando. Puedes hacer que **ksh** complete el nombre de un comando o el nombre de un fichero después de introducir parte de él. Puedes requerir una lista de alternativas o requerir que **ksh** complete tantos caracteres como pueda.

Alias de nombre de comando. Las combinaciones de nombres de comandos y opciones pueden ser personalizadas definiendo un nombre más corto para las mismas, llamadas alias, para comandos que utilizas frecuentemente.

Control de trabajos. **Ksh** proporciona una facilidad para la gestión de múltiples trabajos simultáneamente. En la mayoría de los sistemas, los trabajos pueden ser detenidos y llevados al o traidos desde el modo desatendido o background.

Nuevas capacidades para el comando cd. Puedes volver al directorio previo sin tener que volver a teclear su path completo. Puedes cambiar el directorio actual a un directorio nombrado de forma parecida sin tener que teclear el path completo. **ksh** te permite extender la funcionalidad del comando **cd** reemplazándolo con una función definida por el usuario.

Expansión de la tilde. El directorio home de cualquier usuario, y el último directorio en el que estuviste, pueden ser referenciados de forma simbólica. No es necesario teclear, o incluso conocer, el nombre del directorio.

Mejoras como Lenguaje de Programación

Más mecanismos de entrada/salida generales. Se pueden abrir y leer más de un fichero simultáneamente. El número de columnas impreso para cada item de información puede ser especificado en un programa. La nueva versión de **ksh** tiene las mismas capacidades de formateo que ANSI C.

Primitiva de menú de selección. **ksh** proporciona una forma estructurada para escribir programas que soliciten una entrada del usuario a través de un menú. **ksh** ajusta la visualización del menú al tamaño de visualización de la pantalla.

Aritmética built-in. **ksh** puede realizar aritmética utilizando la sintaxis de expresión del lenguaje C. La nueva versión de **ksh** realiza cálculos en punto flotante y soporta la librería matemética estándar, frente a la versión anterior que estaba restringida a la aritmética entera.

Operadores de subcadenas. **ksh** puede generar subcadenas a partir del valor de variables shell. La nueva versión de **ksh** soporta nuevos operadores y permite que las operaciones se apliquen a agregados.

Atributos y variables de array. Las cadenas se pueden pasar a mayúsculas o minúsculas. Se pueden utilizar arrays de una dimensión de cadenas o números. La nueva versión de **ksh** también soporta los arrays asociativos (arrays cuyos subíndices son cadenas) además de los arrays indexados (arrays cuyos subíndices son enteros).

Variables de referencia y espacio de nombre ampliado. El espacio de nombre para variables es una jerarquía de identificadores con **.** (punto) como delimitador, permitiendo que se definan agregados de datos. Las variables de referencia proporcionan un modo de referirse a agregados de datos sin la necesidad de copiarlos.

Facilidad de funciones más general. Las variables locales dentro de funciones pueden ser definidas y, por lo tanto, se pueden escribir procedimientos recursivos. Se puede especificar código para que se ejecute cuando finalice la función. Las funciones pueden ser cargadas dinámicamente.

Variables activas. Cada variable puede tener funciones definidas que son llamadas cuando se referencia a la variable, se define o se elimina la definición. Estas son llamadas funciones disciplina.

Capacidad de co-procesos. **ksh** proporciona la capacidad de ejecutar uno o más programas de forma desatendida, y enviar y recibir peticiones desde ellos. Esto hace que los shell scripts sean más fáciles de utilizar como front end a un sistema de gestión de bases de datos, un servidor de Internet u otro sistema.

Fácil de depurar. **ksh** visualiza un mejor diagnóstico de error cuando encuentra un error. Por lo que la causa del error se puede encontrar más fácilmente. La ejecución de cada función puede ser rastreada de forma independiente.

Mejor desempeño. Los scripts de **ksh** pueden ser a menudo escritos para que se ejecuten en un orden de magnitud más rápido que scripts de Bourne o C shell similares.

Mejor seguridad. **ksh** permite a una administrador de sistema realizar un log y/o deshabilitar todos los privilegios de un script. En sistemas UNIX actuales, los usuarios necesitan permiso de lectura sobre un script para poder ejecutarlo. Con **ksh**, un administrador de sistema puede permitir a **ksh** que lea y ejecute un script sin tener que darle permiso a un usuario para que pueda leerlo.

Internacional. **ksh** tiene una transparencia de 8-bits completa de forma que puede ser utilizado con conjuntos de caracteres extendidos. **ksh** puede ser compilado para que soporte conjuntos de caracteres multibyte y multiancho tal y como se encuentran en varios lenguajes Asiáticos. La nueva versión de **ksh** también soporta encaje de patrones específicos al locale, comparación y traducción de cadenas.

Extensible. En sistemas con enlace dinámico, los comandos built-in **ksh** pueden ser añadidos en tiempo de ejecución. Estos comandos built-in pueden personalizar partes del espacio de nombre de variable añadiendo funciones disciplina adicionales.

VERSIONES A LAS QUE SE APLICA ESTE MANUAL

ksh se ha desarrollado y madurado con realimentación de los usuarios extensiva. Por lo que existen varias versiones de **ksh**. Este libro detalla la versión de **ksh** de fecha 28 de Diciembre de 1993. Nuevas características en esta versión se listan en el apéndice. Este manual también describe

la versión más antigua de 16 de Noviembre de 1988, extendida y utilizada todavía en muchos sistemas.

Las nuevas características o cambios de la versión de 28 de Diciembre de 1993 se anotan en este manual mediante la palabra "***Versión***", seguida por usa sentencia tal y como "Esta característica está disponible solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988". Advierta que la mayoría de los ejemplos no utilizan las características de la versión de 28/Dic/1993, excepto donde se indica.

Puedes averiguar la fecha de la versión de **ksh** que estás utilizando mediante:

- Utilizando la directiva CONTROL+V de los editores built-in **vi** o **emacs**.
- Mirando en la documentación de tu sistema
- Utilizando el comando **what** con el pathname apropiado. (es posible que tu sistema no tenga el comando **what**)

  Ejemplo

  ```
  what /bin/ksh | grep Versión
        Version 12/28/93b
  ```

- Visualizando la variable **.sh.versión**. **Versión**: La variable **.sh.versión** está solamente disponible en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

  Ejemplo:

  ```
  print ${.sh.versión}
  Versión 12/28/93b
  ```

Un sufijo (*b* en el ejemplo superior) que siga a una fecha indica modificaciones menores y/o locales al código. Sin embargo, las características son prácticamente las mismas para todas las releases de **ksh** con la misma fecha, sin tener en cuenta el sufijo. **Nota**: La última release de la versión de 1988 fue la 16/11/88i.

Como con otros muchos programas, **ksh** está implementado en muchos sistemas diferentes, y en muchas computadoras multiusuario y personales diferentes. Existen diferencias inevitablemente entre implementaciones, normalmente pequeñas, y presumiblemente documentadas por el suministrador del programa. Las características de **ksh** que son más propensas a variar en las diferentes implementaciones se anotan en este manual con las palabras "**Dependiente de la implementación**", seguido de una explicación.

¿CÓMO OBTENER KSH?

La versión de 1988 de **ksh** se incluye en UNIX System V Release 4, y también con otros sistemas operativos. Algunos ejemplos específicos son sistemas UNIX AT&T Soluciones de Información Globales (GIS), Hewlett-Packard ejecutando HP-UX; IBM RS/6000 ejecutando AIX; Computador Apple ejecutando A/UX; Sun Microsystems ejecutando Solaris 2.x; DEC ejecutando OSF/1; e Intel 386 ejecutando SCO UNIX.

Otros vendedores venden el binario de la versión de 1988 de **ksh** como añadidos a sus sistemas o para otros sistemas. Por ejemplo, una versión de MS-DOS del lenguaje KornShell de 1988 está disponible en Mortice Kern Systems, Inc. de Waterloo, Notario, Canadá.

La versión de **ksh** de 1993 será incluida con el entorno de escritorio común, (CDE), que fue definido por COSE, el entorno de sistemas operativo común soportado por la mayoría de los vendedores más importantes de hardware de sistemas UNIX.

El código fuente de las versiones de **ksh** de 1988 y 1993 está disponible desde el AT&T Toolchest. Para mayor información, telefonee o escriba al grupo de soluciones software de AT&T.

Precaución: Algunos programas que podrían parecer similares a **ksh**, tales como **pdksh** y **bash**, desafortunadamente no se conforman actualmente con ninguna de las versiones de **ksh** de 1988 o 1993. Ellos carecen de muchas de las características de **ksh** documentadas en este libro y no pueden ejecutar muchos de los ejemplos.

2. NOTACIÓN UTILIZADA EN ESTE MANUAL

GENERAL

Para aclarar conceptos, nos referimos:

- El programa que implementa el lenguaje KornShell, como **ksh**.
- Los editores built-in **emacs** y **vi**, como **emacs** y **vi**.
- Comandos, simplemente por sus nombres. Los nombres están en negrita, por ejemplo, **alias** y **date**.
- La persona que proporciona la entrada a **ksh**, como *"tú"*. Dependiendo del contexto, esto quiere decir la persona que utiliza **ksh** como un lenguaje de comando interactivo, o como un lenguaje de programación.

TECLAS SOBRE EL TERMINAL

Las teclas especiales que pulsas, tales como CONTROL, están en mayúsculas. Las letras individuales o teclas símbolo que pulsas, tales como **h** o **]**, están también en negrita. Los caracteres en mayúsculas y en minúsculas son equivalentes cuando utilizas la tecla CONTROL; por ejemplo, CONTROL+h o CONTROL+H.

Algunas de estas teclas especiales podrían ser etiquetadas de forma diferente en terminales distintos. Algunas etiquetas alternativas posibles se muestran más abajo. Si tu teclado no tiene ninguna de estas etiquetas alternativas, puedes probar la tecla CONTROL más la letra individual o tecla de símbolo que se muestran más abajo. Si es necesario, vea "Conjunto de caracteres" xxxx para una clarificación más extensa, más la documentación de tu teclado.

BACKSPACE Pulse la tecla etiquetada BACKSPACE, o CONTROL+h

CONTROL Siempre utilice esta tecla en conjunción con una letra individual o tecla de símbolo. Mientras mantiene pulsada la tecla etiquetada con CONTROL o CTRL., pulse la letra individual o tecla de símbolo apropiada.

DELETE Pulse la tecla etiquetada DEL o DELETE, o RUB o RUBOUT, o CONTROL+?, o incluso CONTROL+BACKSPACE

ESCAPE Pulse la tecla etiquetada ESCAPE o ESC, o incluso CONTROL+[

RETURN Pulse la tecla etiquetada RETURN o NEWLINE o ENTER o incluso CONTROL+m

SPACE Pulse la barra espaciadora

TAB Pulse TAB o incluso CONTROL+i.

NOMBRES SIMBÓLICOS PARA CONSTANTES

Si alguno de estos nombres está siguido por una **s**, significa que se permiten uno o varios. Por ejemplo, *Espacios* o *Spaces* significa que se permiten uno o más espacios.

True Valor de retorno de 0 (cero). Esto significa que un comando se ha completado de forma exitosa.

False Valor de retorno distinto de cero. Esto significa que un comando no ha finalizado de forma exitosa. Un número encerrado entre paréntesis después de False indica el valor.

Null Cadena vacía; esto es, una cadena sin caracteres.

Space Carácter ASCII 32 decimal.

Tab Carácter ASCII 9 decimal.

Newline Carácter ASCII 10 decimal.

Return Carácter ASCII 13 decimal.

Bell Carácter ASCII 7 decimal.

NOTACIÓN DE SINTAXIS DE COMANDOS

[ ] (llaves) indican un argumento opcional. Por ejemplo,

[**a**] significa **a** o ningún argumento

a[**b**] significa **a** o **ab**.

[**a**[**b**]] significa **a**, **ab** o ningún argumento

[**a**][**b**] significa **a**, **b**, **ab** o ningún argumento

... (puntos suspensivos) indica que puedes repetir el argumento precedente, separando las repeticiones por algunos de los siguientes:

- Espacios o Tabuladores
- Newlines o saltos de línea, según se indica en el formato mediante los puntos suspensivos que aparecen solos en una línea. Si los puntos suspensivos siguen a unos corchetes, puedes repetir todo dentro del corchete.

PARTE II: TUTORIAL

3. CONCEPTOS DEL SISTEMA OPERATIVO

Ksh es un lenguaje de programación y de comandos que te proporciona un método para comunicarte con tu sistema operativo. Para hacer el mejor uso de **ksh**, necesitas entender dos conceptos básicos del sistema operativo – *ficheros* y *procesos*. Además, para usar **ksh** de forma eficiente, es necesario entender los conceptos de *cadenas* y *patrones*.

Este capítulo te describe lo que debes saber sobre ficheros, procesos, cadenas y patrones para usar **ksh** de la forma más provechosa. Te proporcionará una información adicional que te ayudará a entender los siguientes dos capítulos de la KornShell.

Los usuarios neófitos podrían encontrar este capítulo difícil, ya que introduce muchos conceptos que no le son familiares. Sin embargo, no hay necesidad de dominar estos conceptos para pasar al capítulo siguiente. Ayúdate del "**Glosario**" en el Apéndice para aquellos términos que no entiendas. Si tienes dificultades con este capítulo, vuelve a él posteriormente. Los usuarios de la KornShell experimentados podrían saltarse este capítulo directamente.

FICHEROS

La mayor parte de las veces, un fichero es un conjunto de datos con un nombre que reside en un dispositivo de almacenamiento permanente, normalmente un disco. Los datos contenidos en un fichero pueden variar desde instrucciones de un programa hasta un documento de un procesador de textos. Un tipo especial de fichero llamado *directorio* contiene los nombres de otros ficheros. Sería útil pensar en los dispositivos de almacenamiento como en una cabina de ficheros, un directorio como una carpeta de ficheros, y los ficheros como fichas o ficheros dentro de las carpetas. Anotando que una carpeta podría a su vez contener otras subcarpetas (un directorio puede contener otros directorios y así sucesivamente).

Un sistema de ficheros organizados en disco tal y como el descrito arriba, es conocido como *sistema de ficheros*. Además, un sistema de ficheros normalmente dictamina cuales son los nombres de los ficheros y como están sus permisos fijados, al igual que otras varias propiedades y su organización.

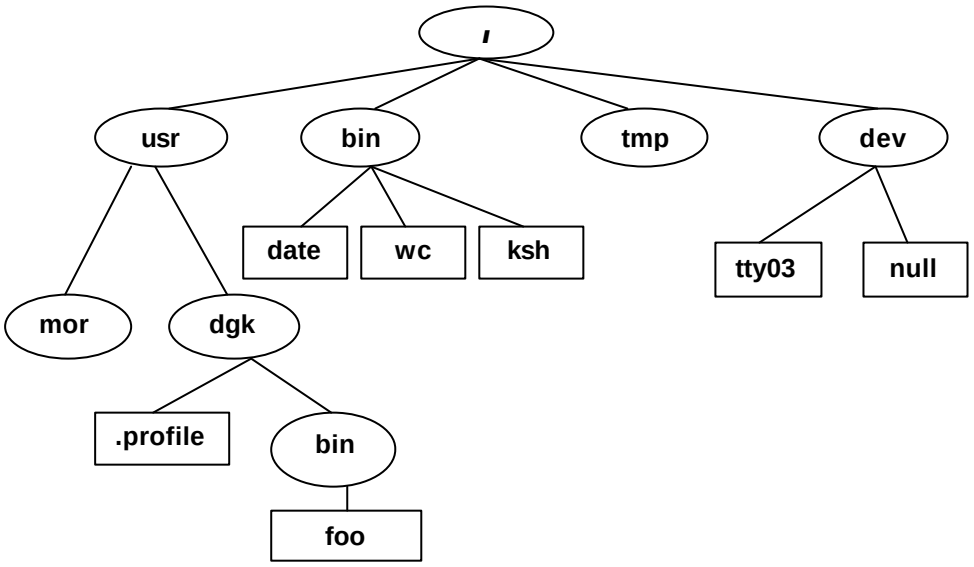
Ksh utiliza el modelo del sistema de ficheros de UNIX, el cuál se describe a lo largo de este capítulo. Si estás usando **ksh** en un sistema que no sea UNIX, tu sistema operativo probablemente utilice un sistema de ficheros diferente. En este caso, existen dos posibilidades. Si tu sistema operativo proporciona un método para "imitar" el sistema de ficheros de UNIX, conocido como emulación, **ksh** no se verá afectado y se comportará del modo que se describe en el presente capítulo. Si no, algunos de las características de **ksh** no funcionarán. Deberás dirigirte a la documentación de **ksh** de tu sistema para buscar diferencias entre tu versión y la versión de UNIX.

Convenciones de nomenclatura

La estructura general de un sistema de ficheros de UNIX es la de un árbol, tal y como se muestra en el diagrama de más abajo.

El árbol está compuesto de directorios (se muestran como óvalos) y ficheros (mostrados como rectángulos). El nombre del directorio raíz es ***/***. Todos los demás nombres de directorios y nombres de ficheros pueden ser una secuencia de caracteres distinta a ***/***. Los caracteres en mayúsculas y en minúsculas son distintos. Algunos sistema limitan un nombres de fichero a 14 caracteres, y algunos sistemas limitan el conjunto de caracteres que pueden ser utilizados en un nombre de fichero. Los nombres de fichero que comienzan con un . (punto) son llamados algunas veces *ficheros ocultos* porque no son listados a menos que se solicite explícitamente.

Un árbol de ficheros de ejemplo. Los directorios son óvalos y los ficheros rectángulos.



*** El "Pathname" o camino del fichero ".profile" es "/usr/dgk/.profile"**

Un pathname es más específico que un simple nombre de directorio o nombre de fichero. Un pathname prefija el nombre de directorio o nombre de fichero con un path o camino, una ruta desde un directorio hasta otro. Un pathname puede también ser referenciado como un *enlace* al fichero. Un path es nombrado por una serie de nombres de directorios separados unos de otros por un carácter ***/***. Por ejemplo, en el ejemplo anterior, un pathname desde el directorio **dgk** hasta el fichero **foo** sería **bin/foo**. Un pathname desde el directorio raíz (***/***) hasta el fichero **tty03** sería **/dev/tty03**.

Un pathname que comienza con ***/*** comienza en el directorio raíz y se conoce como *pathname absoluto*. Si el pathname no comienza con el carácter ***/***, se conoce como *pathname relativo*. Los pathnames relativos comienzan en el directorio actual, o *directorio de trabajo*, en lugar de en el directorio raíz. El directorio de trabajo es mantenido automáticamente por **ksh** y es fijado inicialmente al directorio en el que te encontrabas cuando invocastes **ksh**, o a tu directorio home. Además, el *directorio padre* de un directorio **X** es el directorio que contiene **X**. En este caso, **X** podría también ser referenciado como un *subdirectorio* de su padre.

El directorio de trabajo y el directorio padre pueden ser también referenciados para usar los nombres de directorios especiales **.** (punto) y **..** (punto-punto), respectivamente. Todos los directorios en tu sistema automáticamente contienen un directorio **.** el cuál se refiere al él mismo, y un directorio **..** el cuál se refiere a su directorio padre. El padre del directorio raíz (**/**) es él mismo. Estos dos directorios sirven para múltiples propósitos como se verá posteriormente.

Permisos de Ficheros

Cada fichero tiene un identificador de propietario y un identificador de grupo, los cuales regulan el acceso al fichero.

El identificador de propietario de un fichero nuevo depende del proceso que cree dicho fichero. El identificador de grupo de un fichero nuevo podría depender del proceso que lo cree o del identificador de grupo del directorio en el cual se cree el fichero, dependiendo de tu sistema.

Los ficheros y directorios tienen tres tipos de permisos, conocidos como de lectura, de escritura y de ejecución. Los permisos de lectura y escritura su propio nombre dice lo que indican. El permiso de ejecución sobre un fichero permite la ejecución del programa contenido en el fichero.

Para un directorio, el permiso de lectura permite leer la lista de nombres de ficheros contenidos en dicho directorio. El permiso de escritura de un directorio nos da permiso para crear nuevos ficheros dentro de él o eliminar fichero existentes en el mismo. El permiso de ejecución sobre un directorio nos da permiso para buscar ficheros dentro de dicho directorio, de forma que los ficheros de ese directorio pueden ser accedidos.

Un fichero o directorio puede tener tres conjunto de permisos, cada uno de ellos consiste en una combinación de permiso de lectura, escritura y ejecución. Los tres conjuntos de permisos son:

- Permisos para el propietario del fichero
- Permisos para los miembros del grupo al que pertenece el propietario de dicho fichero
- Permisos que se aplican al resto de usuarios

Además, en algunos sistemas un fichero ejecutable puede tener permiso setuid y/o permiso setgid.

Algunos sistemas tienen un control adicional de acceso más fino, llamado “lista de acceso” que hace un control de acceso por usuario.

Ficheros y Directorios Especiales

Por convención, distintos directorios tienen un propósito especial. Por ejemplo, el directorio **/bin** normalmente contiene programas del sistema usados frecuentemente. Los usuarios, a menudo crean un subdirectorio dentro de su directorio home llamado **bin** para almacenar sus propios programas. Los ficheros temporales son normalmente almacenados en un directorio nombrado por convención **/tmp**.

El directorio **/dev** está normalmente reservado para los *ficheros de dispositivos*. Los ficheros de dispositivos son ficheros especiales que se usan solamente para referirse a dispositivos de entrada y salida tales como terminales, cintas, unidades de disco, altavoces, etc. El fichero de dispositivo llamado **/dev/null** es un fichero especial el cuál está siempre vacío.

Un pipe es un fichero especial que está diseñado para pasar datos desde un proceso a otro.

Algunos sistemas tienen un tipo de fichero especial llamado *enlace simbólico*. Los contenidos de un fichero enlace simbólico es un pathname. Cuando se referencia un nombre de fichero y el sistema determina que es un enlace simbólico, el sistema substituye el nombre de fichero o pathname por el pathname definido por el enlace simbólico. Cuando el pathname definido por el fichero comienza con un **/**, el pathname es substituido; de otro modo, el nombre de fichero es

substituido. Nos referimos al pathname original como el nombre lógico, y al nombre que se obtiene mediante la substitución de todos los nombres de ficheros con los pathnames de los enlaces simbólicos como el nombre físico. Por ejemplo, si **/bin** es un enlace simbólico a **/usr/bin**, entonces el pathname físico para **/bin/date** es **/usr/bin/date**.

PROCESOS

ksh utiliza el modelo de procesos del sistema UNIX.

Puedes pensar en un proceso como en un programa que ha sido lanzado o ejecutado pero todavía no ha finalizado. Es un objeto creado y controlado por el sistema UNIX y otros sistemas similares. Consiste en un programa para ser ejecutado, y de recursos asociados a él por el sistema operativo (tales como memoria y ficheros) que puede leer y/o escribir.

Cada proceso tiene un único thread (hilo) de control. En teoría, varios procesos pueden estar ejecutándose a la vez, incluso el mismo programa. En una computadora que tenga varios procesadores, esto podría ser realmente posible. Sin embargo, en una computadora con un único procesador, los procesos lo que hacen es compartir el procesador por turnos. Esto se conoce como "time-sharing" (compartir el tiempo). Time-sharing es también posible en un sistema de varios procesadores. Incluso aunque la computadora tenga tantos procesadores como procesos, los procesos podrían no estar ejecutándose todos al mismo tiempo debido a que algunos de ellos podrían estar esperando datos de un terminal o de otro proceso.

Creando y Destruyendo Procesos

El sistema operativo sobre el que se ejecuta **ksh** debe proporcionar una forma para crear y destruir procesos. **ksh** te proporciona un interfaz simplificado para que el sistema cree, destruya y gestione procesos.

Cuando se crea un proceso, se le asignan los recursos tales como el área de memoria y los ficheros abiertos. Cuando el proceso termina de ejecutarse, el sistema destruye el proceso y recupera sus recursos, para que puedan ser asignados de nuevo a otros procesos.

Cada proceso tiene un número único asociado a él, conocido como identificador de proceso. Este número distingue al proceso del resto de procesos que han sido creados y se están ejecutando.

El sistema asigna una prioridad a cada proceso. Cuando existen más proceso a ser ejecutados que procesadores existentes, los procesos que tienen una prioridad mayor tiene preferencia sobre aquellos procesos con prioridad menor. Un programa como **ksh** puede influir en la prioridad del proceso.

Cada proceso pertenece a un grupo identificado por el identificador de grupo de proceso. Solamente un único grupo de procesos puede ser asociado a tu terminal a la vez; este grupo es conocido como el grupo de proceso de primer plano (foreground). Cualquier proceso que tengas y no esté en el grupo de proceso de primer plano es conocido como proceso de segundo plano (background).

Relaciones entre Procesos

A un proceso que crea otro proceso se le conoce como proceso padre. Al proceso creado se le conoce como proceso hijo.

En el sistema UNIX, un proceso crea un proceso hijo haciendo una copia de si mismo. A excepción de por el identificador de proceso, estos procesos son inicialmente idénticos. Para ejecutar un programa nuevo, el proceso hijo se sobrescribe a si mismo con el código y los datos iniciales del nuevo programa y entonces se comienza a ejecutar el nuevo programa. Una propiedad de este método es que el proceso hijo tiene una copia de todos los recursos del proceso que lo creó. **ksh** puede usar, pero no lo requiere, el método de creación de procesos del sistema UNIX.

El término “entorno del proceso” se refiere a toda la información asociada a un proceso y que afecta a como se comporta. Cuando se crea un proceso hijo, el proceso hijo hereda una copia de la mayor parte del entorno del proceso padre. Según se va ejecutando un proceso, éste podría modificar su copia del entorno. El entorno de proceso del padre no se ve afectado por los cambios que se realicen en el proceso hijo, a excepción de los efectos laterales debido a modificaciones que se realicen en ficheros físicos. De forma similar, el proceso hijo no se ve afectado por cambios posteriores realizados por el proceso padre.

Un proceso padre puede interrumpir su ejecución hasta que uno o más de sus procesos hijos hayan finalizado. Por ejemplo, **ksh** normalmente ejecuta los programas creando un proceso hijo y esperando a que finalice antes de solicitar el siguiente comando. Cuando un proceso hijo finaliza, devuelve un número (el valor de retorno) a su proceso padre que indica si se produjo o encontró algún error en la ejecución; por convención, los procesos que se ejecutan correctamente devuelven un valor de cero (0).

Permisos de Procesos

Cada proceso tiene un identificador de usuario real y uno o más identificadores de grupo reales. Estos son fijados cuando te conectas, y son heredados por todos los procesos que creas.

Cada proceso tiene también un identificador de usuario efectivo y un identificador de grupo efectivo que determina qué permisos tiene dicho proceso para leer, escribir, y/o ejecutar ficheros. Estos identificadores son fijados al identificador de usuario real y al identificador de grupo real cuando te conectas al sistema, y son heredados del proceso padre.

El identificador de usuario efectivo y el identificador de grupo efectivo se utilizan para determinar a qué ficheros puede acceder el fichero. El identificador de usuario de cualquier fichero creado por un proceso viene determinado por el identificador de usuario efectivo del proceso. El identificador de grupo de cualquier fichero creado por el proceso viene determinado o por el identificador de grupo efectivo del proceso o por el identificador de grupo del directorio en el cual se crea el fichero, dependiendo del sistema. En algunos sistemas, el permiso setgid de un directorio hace que los ficheros creados en dicho directorio pertenezcan al mismo grupo al cual pertenece el directorio. En otros sistemas el grupo del fichero viene determinado por el identificador de grupo efectivo del proceso.

Si un fichero que contiene un programa tiene un permiso setuid y/o setgid, entonces cuando el sistema ejecuta dicho programa, el sistema fija el identificador de usuario y/o grupo efectivo del proceso al del propietario y/o grupo del fichero. **Precaución**: Existen publicaciones de seguridad importantes asociadas a los programas setuid y setgid. No especifiques ninguno de estos permisos a no ser que entiendas las consecuencias y ramificaciones que de ellas se derivan. Algunos sistemas eliminan estos permisos cuando se va a escribir a un fichero como precaución de seguridad.

Señales

Una señal es un mensaje (un número representado por un nombre simbólico) que puede ser enviado a un proceso o un grupo de proceso por el sistema o por otro proceso.

También, presionando ciertas teclas en tu terminal hace que se le envíe una señal a todos los procesos del grupo de procesos de primer plano (foreground).

Sistemas diferentes soportan señales diferentes. Para cada una de estas señales, un proceso debe ignorar explícitamente la señal o debe especificar la acción a realizar cuando reciba la señal; de otro modo, se realiza un acción por defecto. A menos que se especifique, la acción por defecto para las señales listadas más abajo es terminar el proceso. **ksh** requiere que el sistema soporte al menos estas señales:

- **HUP**: Hangup. Esta señal es enviada por el sistema al grupo de proceso de primer plano cuando te desconectas del sistema. Algunos sistemas también envían **HUP** a los grupos de proceso de segundo plano asociados a tu terminal cuando te sales del sistema.
- **INT**: Interrupción. Esta señal es enviada al grupo de proceso de primer plano mediante la pulsación de la tecla de Interrupción. Dependiendo del sistema, la tecla de interrupción por defecto es normalmente DELETE o CONTROL+C.
- **KILL**: Kill. Esta señal es enviada por un proceso a otro o más procesos pertenecientes al mismo usuario para hacer que terminen. La señal no puede ser ignorada por el proceso que la reciba; también, el proceso que la reciba no puede especificar ninguna acción a realizar cuando reciba dicha señal. Por lo tanto, el proceso se terminará. **Precaución**: No confundas la señal **KILL** con el carácter *Kill* descrito posteriormente.
- **TERM**: Terminación. Esta señal es enviada por un proceso a uno o más procesos pertenecientes al mismo usuario, para solicitar su terminación. Un proceso puede especificar un acción a realizar cuando reciba esta señal, o puede ignorarla.

La característica de control de trabajos descrita en el capítulo “Lenguaje de Comandos” requiere estas señales:

- **CONT**: Continúa. Esta señal hace que un proceso detenido continúe con su ejecución.
- **STOP**: Detención. Esta señal hace que el proceso que la reciba se detenga. Un proceso detenido continúa cuando recibe la señal **CONT**. Esta señal no puede ser ignorada por los procesos que la reciben.

- **TSTP**: Detención del teclado. Esta señal es enviada al grupo de procesos de primer plano cuando presionas el carácter Suspensión (*suspend*), normalmente CONTROL+z. Por defecto, un proceso se detiene si recibe esta señal.
- **TTIN**: Entrada de Terminal (tty). Esta señal se envía a cada proceso del grupo de procesos de segundo plano que intenta leer del terminal. Por defecto, un proceso se detiene si recibe esta señal.

Las teclas que hacen que el grupo de procesos de segundo plano reciba una señal puede ser alterado por cualquier proceso que tenga acceso al terminal. Sin embargo, estos ajustes están asociados al terminal (no al entorno del proceso), de forma que todos los procesos que compartan el terminal estén afectados.

Un proceso puede enviar una señal a otro proceso solamente si tiene un permiso apropiado. Para la mayoría de las señales, esto significa que el proceso que lo reciba debe tener el mismo identificador de usuario efectivo que el proceso que envía la señal. Algunos sistemas son menos restrictivos con las señales enviadas.

Comunicación entre Procesos

Además de mediante el envío de señales, los procesos se comunican unos con otros mediante la lectura y escritura de ficheros ordinarios de disco, o mediante la utilización de un fichero especial llamado un pipe. Un pipe está diseñado para ser compartido por más de un proceso. Cuando un proceso intenta leer de un pipe, el proceso realiza una de estas acciones:

- Devuelve los datos si un proceso ha escrito al pipe
- Devuelve una indicación de final de fichero si ningún otro fichero tiene el pipe abierto para escritura
- Suspende la ejecución hasta que un proceso escriba datos al pipe

Otra forma de comunicación ínter procesos es cuando el proceso hijo utiliza el valor de retorno cuando se sale, para comunicárselo a su proceso padre.

Entorno del Proceso

El entorno de un proceso consta de toda la información dentro de un proceso que afecta al proceso. Esto incluye:

- Procesos e identificadores de grupo de proceso
- Ficheros abiertos
- Directorio de trabajo
- Máscara de creación de ficheros (a esto se le conoce normalmente como mask o umask.)
- Identificadores de grupo y usuario real y efectivo
- Límite de los recursos tales como el tamaño más grande de fichero que puede crear un proceso, y la máxima cantidad de memoria que puede utilizar.
- Ajuste de las acciones a realizar cuando reciba una señal
- Un conjunto de nombres de variables

Cada proceso tiene un entorno separado que es inicializado a partir de su proceso padre. Un proceso puede cambiar su entorno utilizando algunos de los comandos de **ksh**. Sin embargo, los cambios realizados en el entorno de los programas que **ksh** ejecuta no cambia el propio entorno de **ksh**.

Los ficheros abiertos son heredados por los procesos hijos a menos que sea especificado close-on-exec.

Descriptores de Fichero

Un proceso asocia un número con cada fichero que ha abierto. A este fichero se le llama el descriptor de fichero. Cuando te conectas al sistema, tu primer proceso tiene los siguientes tres ficheros abiertos y conectados a tu terminal:

- Entrada Estándar: El descriptor de fichero 0 es abierto para lectura
- Salida Estándar: El descriptor de fichero 1 es abierto para escritura
- Error Estándar: El descriptor de fichero 2 es abierto para lectura y escritura.

Un descriptor de fichero puede ser duplicado, dándole otro número que se refiera el mismo fichero. Cada descriptor de fichero que se corresponde con un fichero regular tiene un desplazamiento de byte (byte offset) actual que es compartido por todos los duplicados del fichero. Por lo que, la lectura de y la escritura a un fichero afecta al byte actual de todos los duplicados del fichero.

Cada descriptor de fichero puede tener un bit fijado (conocido como el bit close-on-exec) que hará que el descriptor de fichero se cierre cuando ejecutes un programa nuevo. Si este bit está desactivado (off) el proceso hijo hereda un duplicado del descriptor de fichero.

CADENAS Y PATRONES

Esta sección presenta conceptos y definiciones que son usados a lo largo del manual.

Caracteres y Cadenas

Un carácter es un conjunto de uno o más bytes que representan un símbolo gráfico (también conocido como glyph). El conjunto de caracteres válidos, y el número de bytes que forman un carácter, son definidos como el lugar o "*locale*" actual. En la mayoría de los lugares o "*locales*" de América y Europa, un carácter es un único byte. El conjunto de caracteres para el "*locale*" POSIX por defecto que utiliza la codificación ASCII se lista en la tabla de la página xxxx; cada carácter consiste en un único byte. Todos los sitios o "*locales*" deben contener los caracteres marcados con un √ en la tabla. Sin embargo, los caracteres pueden utilizar una codificación de carácter distinta de la ASCII.

Cada carácter puede ser un miembro de una o más de las clases cuyos nombres aparecen en la página 133. Por ejemplo, la clase **alpha** normalmente contiene todos los caracteres que son usados para formar una palabra en el "*locale*" actual. La tabla de la página xxxx lista las clases de caracteres a las cuales cada carácter del "*locale*" POSIX pertenece.

Una cadena es una secuencia de cero o más caracteres. La longitud de una cadena es el número de caracteres que tenga la cadena. Por convención, los caracteres de una cadena se

numeran empezando por el 0. Una cadena que consista en cero caracteres se le denomina cadena Nula. Al proceso de añadir una cadena al final de otra cadena se le conoce como concatenación. Una secuencia de uno o más caracteres contiguos de una cadena se le llama subcadena.

No existe límite para la longitud de una cadena en el lenguaje **ksh**. A diferencia de otros lenguajes de programación, las cadenas en **ksh** no necesitan ser entrecomilladas, a menos que contengan caracteres con un significado especial para el lenguaje. Las reglas de entrecomillado de **ksh** de la página xxxx describen como entrecomillar un carácter o cadena de caracteres.

Ordenación por Comparación de Cadenas

Las cadenas pueden ser ordenadas o comparadas basándose en un "locale" dado. Para realizar la ordenación, la cadena es descompuesta en una secuencia de elementos únicos.

Por ejemplo, en Español, la secuencia de caracteres "***ch***" es un elemento único. A cada elemento único se le asignan uno o más pesos que son usados para la ordenación de las cadenas. Al primer peso se le llama el peso primario y todos los caracteres que tienen el mismo peso primario forman una clase de equivalencia. Las reglas para la clasificación, las cuales dependen del "*locale*" actual, podrían ser muy complejas y están fuera del ámbito de este manual. En el "*locale*" POSIX por defecto, cada carácter es un elemento único y el peso de cada carácter es el valor numérico del carácter como se muestra en la tabla de la página xxxx. En el "*locale*" POSIX, las cadenas son ordenadas de izquierda a derecha.

Patrones

Una de las principales características de **ksh** es su capacidad de encaje de patrones (pattern-matching). Un patrón es una notación usada para representar un conjunto de cadenas de caracteres. Ya que un patrón viene representado por una secuencia de caracteres, el mismo patrón es una cadena. En cualquier situación en la que se pueda utilizar una cadena o un patrón, y se usa una cadena, es necesario entrecomillar los caracteres que de otra forma serían caracteres especiales propios del encaje de patrones.

Existen tres notaciones comunes para los patrones. Podrías estar ya familiarizado con las dos notaciones de encaje de patrones llamadas expresiones regulares básicas y expresiones regulares extendidas, utilizadas en algunas utilidades del sistema UNIX tales como **ed**, **sed**, **grep** y **egrep**. La construcción de patrones de encaje de **ksh** difiere en notación de estas, aunque ofrece una funcionalidad similar.

Un uso de los patrones es especificar conjuntos de nombres de ficheros. Por ejemplo, el patrón *, el cual es un comodín que representa una cadena de cualquier longitud compuesta de cualquier carácter, puede ser utilizada para que se expanda a una lista de todos los nombres de ficheros del directorio actual que no comiencen con un . (punto), los cuales están escondidos por defecto.

Los patrones se usan para extraer cadenas con unas características particulares de una lista de cadenas. Por ejemplo, el patrón *.c puede ser usado para representar todos los ficheros de código fuente de Lenguaje C en el directorio actual y que no empiezan en . (punto). Los pathnames que comienzan con . (punto) son ficheros escondidos.

Los patrones son utilizados también para operaciones con subcadenas, por medio de lo cual una subcadena de una cadena puede ser eliminada de dicha cadena, puede ser extraída de la cadena, o puede ser reemplazada por otra cadena.

4. LENGUAJE DE COMANDOS

Este capítulo es un tutorial del lenguaje de comandos de la KornShell. Se intenta que te sirva de guía a través del lenguaje, proporcionándote instrucciones paso a paso para algunos de los usos típicos de **ksh**. No se intenta explicar en detalle todas las características de **ksh**, o todos los posibles usos de las características que son presentadas en este capítulo.

Este capítulo se ha preparado tanto para los usuarios neófitos con relativamente poca experiencia con computadores y/o shell como para usuarios experimentados de computadoras que estén familiarizados con otras shells.

Existe un único lenguaje KornShell. La división del lenguaje de comandos y el lenguaje de programación en capítulos separados se debe únicamente a motivos pedagógicos. Utilizamos el término **comandos** para indicar lo que tecleas en tu terminal para una ejecución inmediata. Usamos el término **scripts** para indicar que pones estos comandos dentro de un fichero ejecutable para una posterior ejecución. Los scripts son programas que son escritos en el lenguaje de programación KornShell. El aprendizaje de **ksh** como un lenguaje de comandos es una de las mejores maneras de prepararte el camino para el aprendizaje de los scripts.

Ejecuta **ksh** tecleando su nombre, **ksh** (en algunos sistemas podría ser **sh**), y después pulsa RETURN. Para hacerlo incluso más fácil, en la mayoría de los sistemas, tu administrador de sistema puede hacer que **ksh** sea tú shell por defecto cuando enciendas tu computadora o te conectes al sistema. En algunos sistemas puedes hacer que **ksh** se tu shell por defecto tecleando **chsh** RETURN. Te saldrá a continuación un prompt preguntándote por el nombre de la nueva shell que deseas; teclea el pathname de **ksh** de tu sistema, normalmente **/bin/ksh**.

ksh visualiza un prompt cuando ya está preparado para leer un comando. Este prompt se conoce como prompt primario. El cadena de prompt primario por defecto es **$** (un signo de dólar seguido por un espacio).

Para ver los comandos numerados del mismo modo que en este manual, teclea **PS1='!$ '**. La numeración de comandos en tu prompt se explicará posteriormente.

EJECUCIÓN DE COMANDOS SIMPLES

Introduce un comando **ksh** tecleando una secuencia de palabras. La primera palabra de la línea es el nombre del comando. Las palabra(s) siguiente(s) son elementos de información opcionales u obligatorio utilizados por el comando, llamados los argumentos del comando. Pulsa ESPACIO y/o TAB una o más veces para separar la palabra del nombre de comando de la palabra(s) de los argumento(s) del comando, y para separar cada una de las palabras de los argumentos del comando. Caracteres en mayúsculas y minúsculas, tales como **A** y **a**, son distintos. Por convención, la mayoría de los nombres de comandos y argumentos son en minúsculas.

ksh no comienza a procesar el comando hasta que pulsas RETURN. Si el comando se refiere a un programa, entonces **ksh** crea un proceso hijo para ejecutar el programa y espera a que

el proceso hijo se complete. De otra forma, **ksh** ejecuta el mismo el comando; los comandos ejecutados por el propio **ksh** son llamados built-in.

Para continuar un comando en otra línea, teclea **** como último carácter de la línea antes de pulsar RETURN. **ksh** visualiza entonces un prompt secundario, normalmente **>**. El **** y el RETURN son descartados por **ksh**.

ksh visualiza el prompt primario si pulsas RETURN sin introducir ningún comando. **ksh** visualiza el siguiente prompt primario cuando completa el procesamiento del comando precedente. En la mayoría de los sistemas puedes comenzar a teclear tu siguiente comando tan pronto como hayas pulsado ENTER después del comando anterior, sin tener que esperar al prompt. Esto es muy útil para aquellos comandos que tardan varios segundos en completarse. La salida del comando se podría entremezclar en pantalla con lo que tecleas para el siguiente comando, pero esto no afecta al comando que estás entrando nuevo.

Para abortar un programa antes de que se complete, pulsa la tecla Interrupción. Esto hace que el proceso reciba una señal **INT**, que hace que la mayoría de los programas finalicen. Esto también hace que los caracteres tecleados como adelanto del siguiente comando se pierdan. En algunos sistemas, la tecla BREAK podría causar que el proceso reciba una señal **INT**. Muy pocos comandos ignoran la señal **INT**; en ese caso, utilice la tecla Quit, normalmente CONTROL+\, para enviar la señal **QUIT** si tu sistema la soporta.

En sistema que permiten detener la salida (flujo de control), presiona la tecla Stop, normalmente **CONTROL+S**, para detener la visualización de la salida en tu terminal. Continúe la visualización pulsando la tecla de Restart (reiniciar), normalmente **CONTROL+Q**. En algunos sistemas, puedes continuar la salida pulsando cualquier tecla.

El ejemplo de más abajo utiliza los siguientes comandos:

date Para visualizar el día y la hora

print Visualiza sus argumentos

cat Para visualizar los contenidos de los ficheros que le especifiques como argumentos.

Ejemplos:

```
1$ date
Fri Jul 13 14:41:19 MET DST 2001
2$ print Hola a todos
Hola a todos
3$ cat fichero1 fichero2
Esto es lo que hay en el fichero1.
Estas leyendo ahora la segunda línea.
Esta es la última línea
Esto es lo que hay en el fichero2.
Estas leyendo ahora la segunda línea.
Esta es la última línea.
```

Puedes especificarle opciones a muchos comandos. Por convención, los argumentos de los comandos que representan las opciones siguen al nombre del comando; consisten en un única letra precedida por un – (signo menos). La opción **-?** visualizará la lista de opciones disponibles para casi todos los comandos built-in, y muchos otros comandos que permiten opciones. **Versión**: La opción **-?**

funciona solo con comandos built-in en versión de **ksh** posteriores a la versión del 16 de Noviembre de 1988.

Algunas opciones tienen un valor asociado a ellas. El valor viene dado por el argumento que sigue a la opción. La mayoría de los comandos te permiten agrupar opciones sin valores asociados en una única palabra precedida por un **–**. Por ejemplo, las opciones **–x –v** podrían ser escritas como **–xv**. Los argumentos tales como los pathnames de ficheros se especifican al final. Un **--** por si mismo se utiliza a menudo para indicarle al comando que no vienen más opciones. Esto te permite introducir otros argumentos de comando que comiencen con **-**, sin que **ksh** piense que son opciones.

Sin embargo, no todos los comandos utilizan la convención -. Algunos usan un **+** además del **-**, a menudo para invertir el significado normal de las opciones.

Si introduces un comando de forma incorrecta, proporcionándole opciones desconocidas, o demasiados argumentos o que le falten argumentos, se visualizará un mensaje de error. Los mensajes de error nos muestran frecuentemente el uso correcto del programa.

El comando **ls** visualiza los nombres de los ficheros de tu directorio de trabajo. La opción **–l** del comando **ls** realiza un listado en formato amplio de cada uno de los ficheros. Este formato nos muestra la siguiente información:

- La cadena de permisos de acceso
- Número de nombres que se refieren al mismo fichero
- Propietario
- Grupo
- Tamaño en bytes
- Momento en el que sufrió la última modificación
- Nombre del fichero

Ejemplo:

```
4$ ls –l
drwxr-xr-x   2 dgk      user         128 Jun  7 15:37 bin
-rw-rw-r--   1 dgk      user          86 Jul 18 09:15 file1
-rw-r--r--   1 dgk      user          86 Jul 18 09:27 file2
-rwxr-xr-x   1 dgk      user        4786 Jun 27 09:59 prog
-rw-r--r--   1 dgk      user         674 Jun 29 13:33 prog.c
-rw-r--r--   1 dgk      user         526 Jul  5 13:51 todo
-rw-rwxrwx   1 dgk      user       29386 Jul  6 12:17 wishlist
```

FIJANDO Y VISUALIZANDO LAS OPCIONES

ksh es en si mismo un programa. Por lo que **ksh** tiene varias opciones que puedes especificar o cuando invocas a **ksh** o con el comando **set**. Este manual describe muchas de estas opciones.

Algunas opciones afectan a cómo **ksh** procesa los comandos. **ksh** activa algunas opciones automáticamente, y tú puedes especificar algunas otras opciones. Por ejemplo, **ksh** activa la opción **interactiva** automáticamente cuando invocas a **ksh** de forma interactiva. **ksh** muestra el prompt solamente cuando esta opción está activa. **ksh** le asigna valores por defecto a las opciones, pero

puedes cambiar la mayoría. Utilice **set** para activar, desactivar y visualizar los valores de las opciones.

Normalmente especificas valores por defecto para las opciones en tu fichero profile, el cuál es ejecutado por **ksh** cuando te conectas al sistema; y/o en tu fichero de entorno, el cual es ejecutado por **ksh** cuando comienza su ejecución. Estos ficheros son descritos con más detalle en el capítulo “Personalizando tu entorno”. Por ejemplo, la opción **ignoreeof** es fijada normalmente en tu fichero profile. Evita que te salgas del sistema si pulsas sin darte cuenta el carácter Final-de-fichero que suele ser normalmente CONTROL+d. Con la opción **ignoreeof** fijada, deberás teclear el comando **exit** para salirte del sistema. El ejemplo de más abajo utiliza el comando **set** para activar la opción **ignoreeof** y para visualizar todos los ajustes de opciones. Puntualizar que activas una opción con **–o**, y desactivas una opción con **+o**.

Ejemplos:

```
5$ set –o ignoreeof
6$ set –o
Current option settings
allexport        off
bgnice           on
emacs            off
errexit          off
gmacs            off
ignoreeof        off
interactive      on
keyword          off
markdirs         off
monitor          on
noexec           off
noclobber        off
noglob           off
nolog            off
notify           off
nounset          off
privileged       off
restricted       off
trackall         off
verbose          off
vi               on
viraw            off
xtrace           off
```

CORRIGIENDO LO TECLEADO

El comando **stty** define los ajustes de control del terminal. En particular, puedes utilizarlo para fijar el carácter de backspace (conocido como carácter de Borrado), y el carácter de borrado de línea (conocido como carácter Kill). El valor por defecto para el carácter de Borrado en algunos sistemas es #, y para el carácter Kill es a menudo @. Puedes cambiar los caracteres de Borrado y Kill de forma que puedas utilizar los caracteres # y @ con normalidad. Utiliza **stty** como se muestra en el siguiente ejemplo para fijar el carácter de Borrado a la tecla BACKSPACE y el carácter Kill a la combinación CONTROL+x. **stty** afecta a los ajustes de terminal de todos los procesos que acceden a tu terminal. **stty** interpreta el ^ delante de cualquier carácter como si fuese la tecla de CONTROL. **stty** es puesto normalmente en tu fichero profile.

Ejemplo:

```
7$ stty erase ^h kill ^x
```

En muchos sistemas puedes entrar un carácter de control de terminal como un carácter regular precediéndolo con un \.

Si cometes un error durante la introducción de un comando antes de pulsar el RETURN, existen varias formas para realizar las correcciones:

- Pulsar el carácter de Borrado para volver al error, y corregir la línea desde ese punto
- Pulsar el carácter Kill para eliminar la línea completa y volver a rescribirla

UTILIZANDO EL "ALIAS" COMO ABREVIATURA

Un alias es un nombre que puedes usar como la abreviatura a un comando. Como ejemplo de cómo usar los aliases, supón que siempre deseas ver un listado largo con el comando **ls**. Definiendo un alias para el comando **ls –l**, no tienes que teclear la opción **–l** cada vez que listas el contenido de un directorio.

Defines un alias con el comando **alias**, seguido de una palabra con la siguiente forma *nombre=valor* donde *nombre* es la abreviatura para *valor*. No pulses ESPACIO o TABULADOR antes o después del signo =; si lo haces, **ksh** entenderá que tecleaste dos o tres palabras en lugar de la única palabra que tiene que ser. Si el *valor* tiene algunos espacios o tabuladores dentro de él, entonces debes entrecomillar el *valor* usando cualquier de los mecanismos descritos brevemente en este capítulo y con profundidad en un capítulo posterior. **Nota**: Puedes utilizar el nombre del alias dentro de su valor. Por ejemplo, en el ejemplo de más abajo, el alias para **ls** será fijado a **ls –l**. El **ls** dentro de las comillas no será reemplazado de nuevo por **ls –l**.

Para visualizar el valor de uno o más aliases, especifique el nombre de dichos aliases como argumentos al comando **alias.** Para visualizar la lista completa de aliases, ejecute **alias** sin argumentos. Algunos aliases han sido prefijados.

ksh chequea cada nombre de comando para ver si has definido un alias para él. Si lo has definido, **ksh** substituye el comando con el valor de alias.

Ejemplos:

```
8$ alias ls='ls –l'
9$ alias ls
'ls –l'
10$ ls
drwxr-xr-x   2 dgk      user         128 Jun  7 15:37 bin
-rw-rw-r--   1 dgk      user          86 Jul 18 09:15 file1
-rw-r--r--   1 dgk      user          86 Jul 18 09:27 file2
-rwxr-xr-x   1 dgk      user        4786 Jun 27 09:59 prog
-rw-r--r--   1 dgk      user         674 Jun 29 13:33 prog.c
-rw-r--r--   1 dgk      user         526 Jul  5 13:51 todo
-rw-rwxrwx   1 dgk      user       29386 Jul  6 12:17 wishlist
```

REINTRODUCIENDO COMANDOS PREVIOS

ksh mantiene un log de los comandos que introduces desde el terminal en un fichero histórico. Teclea **history** para visualizar los comandos escritos más recientemente. **history** por si mismo visualiza como máximo 16 comandos. Sin embargo, puedes especificar, por ejemplo:

`history –11` para limitar la visualización a los últimos 11 comandos

`history 3 7` para visualizar los comandos desde el 3 hasta el 7

`history 11` para visualizar los comandos comenzando a partir del 11

Los comandos indicados deben estar aún accesibles en el fichero history.

Para reintroducir el comando **ksh** previo, ejecuta el comando **r** sin argumentos. **ksh** visualiza el comando, y después lo ejecuta. Para volver a ejecutar el comando 4, teclea **r 4**. Para volver a ejecutar el último comando que comience con **d**, teclea **r d**.

Ejemplos:

```
11$ r d
date
Mon Jul 23 12:09:53 MET DST 2001
12$ history
1     date
2     print hello world
3     cat file1
4     ls -l
5     set –o ignoreeof
6     set -o
7     stty erase ^h kill ^x
8     alias ls=’ls –l’
9     alias ls
10    ls
11    date
12    history
```

CAMBIANDO EL PERMISO DE LOS FICHEROS

Los permisos de los ficheros son especificados en uno de los dos posibles formatos.

Puedes especificar los permisos con un número octal de 3 o 4 dígitos.

1º Permisos setuid y/o setgid. Use este dígito solamente si deseas darle a un programa permisos especiales. Los permisos setuid y setgid a menudo no existen en sistemas que no son UNIX.

2º Propietario

3º Grupo

4º Otros

Cada dígito octal, excepto el primero, es la suma de todos los valores correspondientes a los permisos del propietario, del grupo y/o de otros. De esta forma, por ejemplo, si el propietario tiene permisos de lectura y escritura, entonces el segundo dígito debería ser la suma de 4 y 2, la cuál es 6. Los valores son:

- Lectura: 4
- Escritura: 2
- Ejecución o búsqueda: 1

Para que sirva de ejemplo completo, el permiso 4751 significa:

- Setuid
- Lectura, escritura y ejecución para el propietario
- Lectura y ejecución para el grupo
- Ejecución para el resto

O puedes especificar los permisos con una expresión de permisos simbólica. El formato de la expresión son uno o más “***[a quién] op [permiso]***” separados por una coma, donde:

- ***a quién*** es una combinación de las letras:

u Permisos para el propietario (u significa usuario)

g Permisos de grupo

o Permisos para otros

a Valor por defecto si no especificas "***a quién***". Especifica permisos para el propietario, el grupo y el resto.

- ***op*** es uno de los siguientes:

+ Dar permiso

- Quitar permiso

= Asignar permiso

- ***permiso*** puede ser cualquiera de estos o todos ellos:

r Lectura

w Escritura

x Ejecución o búsqueda

s Fija el identificador de propietario y/o de grupo cuando se usa con **u** y/o **g**.

La máscara de permiso **4751** del ejemplo anterior puede ser representada con la expresión de permisos simbólica **u=rwxs,g=rx,o=x**.

Los permisos de acceso son frecuentemente visualizados como una cadena de permiso de 10 caracteres (tal y como muestra **ls –l** más arriba). El primer carácter es **d** para un directorio, **c** para un dispositivo tipo carácter tales como los terminales, o **–** para un fichero. Los otros nueve caracteres, en grupos de tres, representan los permisos para el usuario, grupo y el resto de usuarios respectivamente. Una **r** representa permiso de lectura (read) una **w** representa permiso de escritura (write). Una **x** representa permiso de ejecución para un fichero y permiso de búsqueda para un directorio (execute). Un **–** en cualquiera de estas posiciones representa la carencia del permiso correspondiente. Una **s** en el lugar de la **x** indica el permiso setuid o setgid. Como ejemplo, `-rwsr-x---x` representa el permiso de fichero **4751** descrito anteriormente.

Cada proceso tiene una máscara de creación de fichero para deshabilitar permisos cuando se crea un fichero. La máscara de creación de ficheros puede ser representada como un número octal de 3 dígitos, o como un permiso simbólico. Sus dígitos representan los permisos a ser deshabilitados cuando se crea un fichero. Por ejemplo, un valor de máscara de creación de fichero de **022** deshabilita el permiso de escritura para el grupo y el resto de usuarios. De forma alternativa, la expresión de permisos simbólicos **go-w** representa la misma máscara de creación de fichero.

Utilice el comando **umask** para fijar y/o visualizar la máscara de creación de fichero. Utilice el comando **chmod** si es necesario cambiar los permisos de acceso.

Ejemplos:

```
13$ umask
002
14$ umask –S
=rx,ug+w
15$ umask 022
16$ umask –S
=rx,u+w
```

```
17$ chmod g+w prog.c
18$ ls –l prog.c
-rw-rw-r--   1 dgk      user         674 Jun 29 13:33 prog.c
```

REDIRECCIONANDO LA ENTRADA Y LA SALIDA

Por convención, la mayoría de los comandos escriben su salida normal a la salida estándar (descriptor de fichero 1), y los mensajes de error al error estándar (descriptor de fichero 2). Por defecto, **ksh** direcciona ambas, tanto la salida estándar como el error estándar para que sean visualizados en tu terminal.

Puedes redireccionar la salida estándar de un comando a un fichero en lugar de a tu terminal tecleando el operador de redirección, **>**, seguido por el nombre del fichero.

Puedes colocar el **>** en cualquier parte de la línea; por convención, el **>** se coloca normalmente al final de la línea de comando detrás de sus argumentos. Los espacioes y /o tabuladores antes o después del **>** son opcionales, a menos que el argumento que preceda al **>** sea un único dígito. **Nota**: Este párrafo se aplica a todos los operadores de redireccionamiento, no solamente a **>**.

Si el fichero ya existe, y la opción **noclobber** está activa, entonces **ksh** visualiza un mensaje de error. Sin embargo, puedes utilizar **>|** en lugar de **>** para hacer el redireccionamiento a un fichero que ya exista incluso si la opción **noclobber** está activa. Si **noclobber** no está activa, o si especificas **>|** entonces **ksh** elimina el contenido del fichero anterior.

Ejemplos:

```
19$ date > savedate
20$ cat savedate
Mon Jul 23 13:16:28 MET DST 2001
```

Si el fichero no existe, entonces **ksh** lo crea. **ksh** fija los permisos de lectura y escritura a todo el mundo, menos los permisos especificados como valor de la máscara de creación de ficheros.

Utilice **>** sin especificar ningún comando previo para crear un fichero vacío. Utilice **rm** para eliminar un fichero que hayas creado.

Ejemplos:

```
21$ > tempfile
22$ ls –l tempfile
-rw-r--r--   1 dgk      user           0 Jul 23 13:25 tempfile
23$ rm tempfile
```

Si quieres descartar la salida estándar de un comando, puedes hacer eso redireccionandola a un fichero especial, **/dev/null**. Cualquier carácter que se escriba a este fichero será perdido.

Puedes añadir la salida estándar de un comando a un fichero tecleando el operador de redirección **>>** seguido por el nombre del fichero. Si el fichero no existe, entonces **ksh** lo creará por ti.

Ejemplos:

```
24$ date >> savedate
25$ cat savedate
Mon Jul 23 13:16:28 MET DST 2001
Mon Jul 23 13:31:27 MET DST 2001
```

Puedes redireccionar la entrada estándar de un comando de forma que lea de un fichero en lugar de leer de tu terminal. Teclee el operador de redirección **<** seguido por el nombre del fichero.

Ejemplo:

```
26$ mail morris < savedate
```

El comando **cat** lee de la entrada estándar si no le especificas ningún fichero. Por lo tanto, ambos **cat savedate** y **cat < savedate** proporcionan el mismo resultado. Sin embargo, en la primera forma, **savedate** es un argumento de **cat**, y en la segunda forma **cat** lee de la entrada estándar donde **ksh** ha abierto el fichero **savedate** para que se a usado por **cat**. Si **savedate** no existe, obtendrás distintos mensajes de error dependiendo de cual de las dos formas hayas utilizado. Esto se debe a que el mensaje de error es generado por **cat** en la primera forma y por **ksh** en la segunda forma.

Puedes redireccionar cualquier descriptor de ficheros desde **0** a **9** especificando el número de descriptor de fichero inmediatamente antes del operador de redireccionamiento <, >, y >>. Por ejemplo, para redireccionar el error estándar, especifica **2>**. Puedes descartar ambas salidas, tanto la salida estándar como el error estándar redireccionando ambos a **/dev/null**.

Ejemplo:

```
27$ date > /dev/null 2> /dev/null
```

Puedes también especificar que un descriptor de fichero sea redireccionado al mismo sitio que otro descriptor de fichero. Para hacer que el descriptor de fichero 2 sea redireccionado al mismo fichero que el descriptor de fichero 1, especifique **2>&1**. Puedes pensar en **>&** "*redirecciona al mismo fichero que*".

El orden en el que se especifica la redirección es importante porque los operadores de redireccionamiento se procesan de izquierda a derecha. En el siguiente ejemplo, la salida estándar es redireccionada a **/dev/null** y el error estándar es también redireccionada al mismo fichero que la salida estándar de forma que ambas se pierden. Si el orden de redireccionamiento fuera inverso, el error estándar se redireccionaría a donde quiera que estuviese redireccionada la salida estándar en ese momento (el terminal), y posteriormente, la salida estándar sería redireccionada a **/dev/null**.

Ejemplo:

```
28$ date > /dev/null 2>&1
```

PIPELINES (ENCAUZAMIENTO) Y FILTROS

Puedes conectar la salida estándar de un comando con la entrada estándar de otro comando utilizando el comando pipeline (¿gaitero?) **|** entre los comandos. Este es aproximadamente equivalente a ejecutar el primer comando con su salida redireccionada a un fichero temporal, después ejecutar el segundo comando con su entrada redireccionada desde dicho fichero temporal, y después eliminando el fichero temporal. Sin embargo, **ksh** utiliza el fichero especial pipe en sistemas que lo tienen, en lugar de crear un fichero intermedio.

Utilizando un fichero especial pipe, **ksh** funcionará incluso si el primer comando produce más salida que el tamaño de ficheros más grande permitido. También, **ksh** no esperará a que finalice el primer comando antes de comenzar el segundo comando. Los pipes son normalmente más rápidos que la utilización de ficheros temporal, debido a que los datos son mantenidos en memoria.

Puedes conectar una secuencia de comandos juntos con un **|** para crear un pipeline. El número de comandos que puedes conectar podría estar limitado por el número de procesos que el sistema te permite crear. Puedes continuar los pipelines en más de una línea tecleando RETURN después del carácter **|**.

Un comando que lee de su entrada estándar y escribe a su salida estándar es llamado un filtro. Puedes usar el operador **|** para conectar uno o más filtros para preprocesar y/o postprocesar la salida de cualquier comando.

Al igual que con muchos comandos que procesan ficheros, **grep** lee de la entrada estándar si no le especificas el nombre de un fichero como un argumento. Por lo tanto, **grep** puede ser usado como un filtro. **grep** muestra las líneas de uno o más ficheros que contienen un secuencia de caracteres especificada. Utilice **grep** cuando quieras limitar la salida a solamente aquellas líneas que necesitas ver. El ejemplo de más abajo visualiza solamente aquellas líneas de la salida del comando **set –o** que contienen **ignoreeof**.

Ejemplo:

```
29$ set –o | grep ignoreeof
ignoreeof        off
```

ksh redirecciona la salida estándar al pipe antes de la redirección de la Entrada/Salida de los comandos individuales. Por lo tanto puedes utilizar **2>&1** para direccionar el error estándar a el pipeline.

Utilice el comando **tee** para capturar, en un fichero, los datos que pasan a través de un pipe en un pipeline. **tee** no altera los datos que pasan a través de él. En el ejemplo de más abajo, el fichero **users** obtendrá la lista de los usuarios conectados al sistema según es generada por el comando **who**. En el ejemplo, tanto la salida estándar como el error estándar son escritos al pipe.

El comando **wc** visualiza el número de líneas, palabras y caracteres en el fichero(s) especificado(s). La opción **–l** limita la salida al número de línea del fichero. Si no se especifican ficheros, **wc** utiliza la entrada estándar.

Ejemplo:

```
30$ who 2>&1 | tee users | wc –l
159
```

En algunos terminales no puedes hacer un scroll hacia atrás a páginas anteriores de la salida por pantalla. En este caso, podrías desear que la salida se detuviese cada vez que la pantalla se llenase, hasta que tu indicases que estás dispuesto a ver la siguiente pantalla de salida. Un programa que hace esto es conocido como paginador, y **more** es un programa paginador usado comúnmente. Puedes hacer que la salida de cualquier comando sea paginada ejecutando el comando **more** como último comando en un pipeline. Los sistemas que no tienen el comando **more**, a menudo tienen un comando llamado **pg** para realizar esta función. Una diferencia entre los comandos **pg** y **more** es que **more** utiliza la tecla ESPACIO para mostrar la siguiente pantalla de salida, mientras que **pg** utiliza RETURN.

EXPANSIÓN DE LA TILDE

Puedes reducir el esfuerzo de tecleado utilizando **~** al comienzo de ciertas palabras. **ksh** chequea la expansión de cada palabra que tecleas que comience con un **~**. Utilice el mecanismo de expansión de tilde como una abreviatura para:

- Tu directorio home. Utilice el mismo **~**.

- El directorio home de cualquier usuario del sistema. A continuación de **~** especifique el nombre de login del usuario deseado (tu mismo o cualquier otro)
- El pathname absoluto de tu directorio de trabajo. Utilice **~+**.
- El directorio de trabajo anterior. Utilice **~-**.

Los caracteres de expansión de la tilde se prolonga hasta el primer */*, si hay alguno. Si no existe */*, **ksh** procesa la palabra entera. **ksh** deja la palabra tal y como la tecleaste, si la **~** no es seguida por ninguno de los caracteres especificados más arriba.

Ejemplo:

```
31$ print ~morris/reminders
/user/morris/reminders
```

VARIABLES

Las variables te permiten almacenar y manipular información con **ksh**. Las variables son utilizadas principalmente dentro del lenguaje de programación de **ksh**, y son por lo tanto descritas en más detalles en el capítulo siguiente. Sin embargo, una breve introducción a las variables es apropiada antes de continuar.

Una variable tiene un nombre, llamado *varname*, y un valor. Un valor es una cadena de cualquier longitud. Se le puede asignar un valor a una variable de dos formas – o por el usuario, o por **ksh**. Un usuario fija el valor de la variable explícitamente utilizando un *comando de asignación*. Esto es hecho comúnmente en tus ficheros de entorno y profile. En otros casos, **ksh** asigna el valor de una variable, normalmente como el resultado de un comando. Algunas variables son inicializadas o heredadas por **ksh** cuando te conectas al sistema. Por ejemplo el valor de **HOME** se fija al pathname absoluto de tu directorio de entrada al sistema.

Para obtener el valor de una variable, precedes su nombre con un **$** y rodeando el nombre de la variable con **{}**. Puedes pensar en el signo **$** como si significase, "*el valor de*". Por ejemplo, **${PWD}** significa "*el valor de la variable **PWD***". El **{** y **}** son requeridos solamente si el nombre de la variable contiene un **.** (punto), o el carácter que sigue al **{** es una letra, dígito o _ (subrayado).

Ejemplo:

```
32$ d=$HOME/.profile s=~morris
33$ print ${d}_file $s/bin
/usr/dgk/.profile_file /usr/morris/bin
```

EXPANSIÓN DEL "PATHNAME"

Muchos comandos toman una lista de ficheros como argumentos. Puedes utilizar **ksh** para generar una lista de argumentos que son pathnames tecleando una palabra que consiste en un patrón en lugar de teclear cada nombre individualmente. Los patrones son utilizados en otros contextos por **ksh**. Si utilizas cualquiera de los siguiente caracteres sin entrecomillar en una palabra de comando, entonces **ksh** procesa esa palabra como un patrón y la expande como sigue:

* Concuerda con cualquier cadena de caracteres, incluyendo el Nulo

? Concuerda con cualquier carácter (un único carácter)

[...] Concuerda con cualquier carácter(eres) especificados entre los corchetes.

(...) Cuando es precedido por un *, ?, @, +, o !, los patrones dentro de los paréntesis son encajados utilizando las reglas de encaje de patrones.

Cuando especificas un patrón como una palabra de comando, **ksh** expande el patrón a la lista completa de pathnames que encaja con el patrón. Si no existen pathnames que encajen, entonces **ksh** deja el patrón sin cambios.

Cuando los patrones se usan para encajar con pathnames, un **.** (punto) como el primer carácter de cada nombre de fichero debe encajar explícitamente. Cada nombre de fichero debe encajar. Si la variable **FIGNORE** está activada, entonces los nombres de ficheros que encajen con el patrón definido por el valor de la variable **FIGNORE** son excluidos del encaje en lugar de los nombres de ficheros que contienen un **.** (punto) inicial. **Versión**: **FIGNORE** está solamente disponible en versiones de **ksh** posteriores a la versión de 16 de Noviembre de 1988.

Si la opción **markdirs** está activa, entonces un **/** final es añadido a cada nombre de directorio que encaja con el patrón.

Los pathnames que encajan con el patrón se convierten en argumentos para el comando. Debido a que el encaje es realizado por **ksh**, la expansión de pathname se aplica a todos los comandos.

Si especificas un patrón como el pathname asociado con un operador de redireccionamiento de entrada/salida, **ksh** expande el patrón solamente si encaja con un único pathname.

Ejemplos:

```
34$ print *
AWK Geo Herramientas Shell_Scripts Usuarios agrupar_campos.sh c_shell
35$ ls –l c??
-rw-rw-r--   1 dwadmin  informix      86 Apr 18  2000 con
-rw-r--r--   1 dwadmin  informix      86 Mar 15 11:45 crm
```

COMANDOS DE TIMING

Puedes averiguar cuanto tiempo ha tardado en procesarse un comando o un pipeline, como en el ejemplo de más abajo, utilizando el comando **time**. **time** es una palabra reservada en **ksh** y es procesada de forma especial. Se aplica a pipelines completos, no solamente a comandos simples.

Después de que el comando o el pipeline se complete, **ksh** visualiza en el error estándar tres líneas que especifican en minutos y segundos:

- El tiempo de reloj que ha transcurrido
- El tiempo de procesador utilizado por el programa
- El tiempo de procesador utilizado por el sistema en la ejecución del comando o pipeline

Las redirecciones de la entrada / salida se aplican al comando al que aplicas el **time**, no al propio **time**. Encierra el comando **time** en uno de los comandos de agrupación, y redirige el error estándar del comando agrupado para redireccionar la salida del comando **time**.

Ejemplo:

```
36$ time ls –l /bin | wc
       1      11      72

real    0m0.02s
user    0m0.00s
```

```
sys     0m0.03s
```

El comando **sleep** en el siguiente ejemplo detiene la ejecución durante el número de segundos especificados como argumento. Este comando es normalmente utilizado dentro de uno de los comandos compuestos descritos en el capítulo de "Lenguaje de Programación". **Versión**: En versiones de **ksh** posteriores a la de 16 de Noviembre de 1988, **sleep** es un comando built-in, y el número de segundos no tiene por qué ser un entero. Se pueden especificar fracciones de segundo.

Ejemplo:

```
37$ time sleep 10

real    0m10.01s
user    0m0.00s
sys     0m0.01s
```

ENTRECOMILLANDO CARACTERES ESPECIALES

Supón que quieres visualizar el carácter **|**. Si tecleas **print |** entonces **ksh** te sacará el prompt secundario **PS2** (normalmente **>**), indicando que está esperando a que teclees el resto del pipeline.

Para eliminar el significado normal que **ksh** le asigna a **|**, o a cualquier otro carácter especial, debes entrecomillarlo. Puedes entrecomillar un único carácter precediéndolo con un signo ****. Puedes entrecomillar una cadena de caracteres encerrándola en:

- Comillas literales (simples), **'…'** para eliminar el significado especial de todos los caracteres a excepción del '.
- Cadenas ANSI C, **$'…'**, para eliminar el significado especial de todos los caracteres a excepción de las secuencias de escape de ANSI C. Por ejemplo, **$'\t\n\'\\'** representa la cadena de cuatro caracteres Tabulador, Salto de línea, ' y \. **Versión**: Las cadenas de ANSI C están solamente disponibles en versiones de **ksh** posteriores al 16 de Noviembre de 1988.
- Comillas de agrupación (dobles), **"…"**, para eliminar el significado especial de todos los caracteres excepto de **$**, ****, y `. Si el **$** no está entrecomillado, la substitución de variable se realiza dentro de las comillas dobles.

ksh normalmente particiona los resultados de la expansión de parámetros y la substitución de comando de las palabras de comando en campos, utilizando como delimitadores de campo el carácter que se encuentra almacenado como valor de la variable **IFS**. Por defecto, el valor de **IFS** es el Espacio, Tabulador y Salto de Línea. **ksh** genera los argumentos reales realizando una expansión de pathname de los campos. Utilice las comillas de agrupación (dobles) para prevenir ambos, tanto el particionamiento de campos como la expansión de pathnames. Los caracteres de entrecomillados son eliminados después de que cada palabras de comando es expandida.

En el siguiente ejemplo utilizamos la opción **–r** para prevenir que el carácter **** sea procesado de forma especial por **print**.

Ejemplo:

```
38$ print –r \\foo \~ '<$HOME>' "<$HOME>"
\foo ~ <$HOME> </usr/dgk>
```

DIRECTORIO DE TRABAJO

Cuando quiera que un comando utilice un pathname que no comience con un **/**, el sistema busca el fichero con respecto a tu directorio de trabajo actual.

Para saber el nombre del directorio de trabajo, utilice **pwd**. Por defecto, **pwd** visualiza el nombre lógico del directorio. Use **pwd –P** para visualizar el nombre físico del directorio, el nombre del directorio con todos los links simbólicos resueltos. Para cambiar el directorio de trabajo, use **cd** seguido del pathname del nuevo directorio. La opción **–P** de **cd** hace que el nombre que especifiques sea convertido a nombre físico. Puedes usar la variable **CDPATH** para especificar una lista de directorios para que **cd** busque en ellos cuando especifiques un pathname que no comience con un **/**. **cd** visualiza el nombre del nuevo directorio de trabajo cuando utiliza la variable **CDPATH** para encontrar dicho directorio, y el nuevo directorio no es un subdirectorio del directorio actual.

Para volver al directorio de trabajo anterior en el que estabas, utiliza **cd-**. **ksh** visualiza el nombre del nuevo directorio de trabajo.

ksh fija la variable **PWD** al directorio de trabajo cuando utilizas **cd**. **ksh** también fija la variable **OLDPWD** a tu directorio de trabajo anterior.

Si quieres cambiar a un directorio cuyo pathname difiere ligeramente del de tu directorio de trabajo actual, puedes hacer usando **cd** con dos argumentos. El primero especifica la parte del pathname que quieres cambiar, y el segundo especifica el directorio al que te quieres cambiar. **ksh** visualiza el nuevo directorio de trabajo.

Ejemplos:

```
39$ pwd
/usr/dgk
40$ pwd –P
/n/toucan/u6/dgk
41$ cd /usr/morris
42$ cd –
/usr/dgk
43$ cd –
/usr/morris
44$ cd morris dgk
/usr/dgk
```

COMO ENCUENTRA KSH UN COMANDO

Si el nombre de comando es un alias, **ksh** reemplaza el nombre de comando por el valor del alias y comienza la búsqueda de nuevo.

Si el nombre de comando contiene un **/**, si es posible, **ksh** ejecuta el programa cuyo pathname es el nombre de comando.

Si el nombre de comando no contiene un **/**, entonces **ksh** chequea el nombre de comando contra uno de los siguientes, en el orden mostrado:

- **Palabra reservada**. Si la palabra reservada comienza un comando compuesto, entonces **ksh** lee tantas líneas como sean necesarias hasta que lee el comando completo o encuentra un error, o hasta que pulsas la tecla de Interrupción. Si la palabra reservada es inválida como primera palabra del comando, entonces **ksh** visualiza un mensaje en el error estándar.

- **Comando built-in especial**. **ksh** ejecuta el comando built-in especial en el entorno actual.
- **Función**. **ksh** ejecuta la función en el entorno actual.
- **Utilidad built-in regular**, **ksh** ejecuta la utilidad built-in regular en el entorno actual.
- **Si no**, **ksh** utiliza el valor de la variable **PATH** seguida del valor de la variable **FPATH** para construir una lista de directorios en los que buscar el comando. Las variables **PATH** y **FPATH** cada una contiene una lista de directorios separados por el carácter **:** (dos puntos). La búsqueda se realiza en el orden en el que los directorios aparecen en la variable **PATH**. Si el nombre de comando se encuentra en un directorio, **ksh** hará algo de lo siguiente:
    - Si dicho directorio está también en la lista de directorio definidos por la variable **FPATH**, entonces **ksh** lee y ejecuta el fichero en el entorno actual y después ejecuta una función del nombre dado.
    - O si no, si **ksh** contiene una versión built-in del comando correspondiente al pathname absoluto del comando, este comando built-in es ejecutado.
    - De otro modo, **ksh** ejecuta el comando dado como un programa en un entorno separado.

Versión: Con la versión del 16 de Noviembre de 1988 de **ksh**, los directorios en la variable **PATH** no podían contener directorios de funciones y no existían comando built-in que se correspondiesen a pathnames absolutos. También, todos los comandos built-in eran encontrados antes de las funciones del mismo nombre, de forma que era necesario definir un alias y una función para anular el nombre de un comando built-in con una función.

ksh visualiza un mensaje en el error estándar si no pudiese encontrar el comando, o si no tuvieses permisos de ejecución para este fichero.

Utilice el alias prefijado **type** para averiguar que comando se ejecutará cuando especifiques un determinado nombre de comando. **type** te dice a qué tipo de elemento se refiere el nombre de comando especificado. Si el nombre se corresponde con un programa, **type** visualiza un pathname para el programa. **whence** es similar a **type** excepto que **whence** visualiza solamente el pathname absoluto del comando, si existe.

Ejemplos:

```
45$ type ls date print r
ls is a tracked alias for /usr/bin/ls
date is a tracked alias for /usr/bin/date
print is a shell builtin
r is an exported alias for fc -e –
46$ whence date
/usr/bin/date
```

SUBSTITUCIÓN DE COMANDOS

Puedes desear que la salida de un comando se convierta en el valor de una variable. O podrías querer que el argumento(s) de un comando fuesen generados por otro comando. Puedes indicarle a **ksh** que haga cualquiera de estas cosas encerrando el comando cuya salida deseas dentro de **$(...)**. Además, **$(<fichero**) se expande a el contenido del fichero.

Bourne shell: La sintasis de la Shell de Bourne `…`, es reconocida por **ksh** pero se considera como obsoleta. Todos los ejemplos de este libro utilizan **$(...)** en lugar de **`...`**. **$(...)** presenta unas reglas de entrecomillado más simples y se anida más fácilmente.

Ejemplos:

```
47$ print dgk > foobar
48$ foo=$(<foobar)
49$ print $foo: "$(date)"
dgk: Tue Jul 24 11:58:14 MET DST 2001
50$ d=$(whence date)
51$ print $d
/usr/bin/date
```

EJECUCIÓN DE COMANDOS EN MODO DESATENDIDO

Por defecto, **ksh** espera a que los comando se acaben de ejecutar antes de mostrar el siguiente prompt. El comando que está esperando **ksh** a que acabe se conoce como comando *foreground* (de primer plano).

Algunos comandos tardan mucho tiempo en ejecutarse. Puedes teclear un **&** al final de un comando o pipeline justamente antes de pulsar RETURN, para hacer que este comando o pipeline se ejecute en *background* (modo desatendido).

Antes de que **ksh** comience la ejecución de un comando en background, visualiza un mensaje con un número de trabajo dentro de corchetes, **[ ]**, seguido por el identificador de proceso. Si no redireccionaste la salida del comando background, entonces la salida del comando background es visualizada en tu terminal.

El comando **find** realiza las acciones especificadas en todos los ficheros de uno o más directorios o subdirectorios especificados que satisfagan las condiciones especificadas. A menudo tardan bastante tiempo en finalizar. En el siguiente ejemplo, se especifica el directorio **/usr** y la acción es visualizar el pathname de todos los ficheros con nombre **foobar** como último componente. No se especifican condiciones, de forma que todos los pathnames serán visualizados.

Ejemplo:

```
52$ find /usr –name foobar –print &
[1]     9947
```

ksh disocia los comandos background de tu terminal. Por lo tanto, no puedes utilizar la tecla de Interrupción para finalizar o abortar un comando background. Puedes utilizar el comando **kill** para enviarle al trabajo una señal **TERM** o cualquier otra señal.

Si la opción **monitor** está activa, **ksh** ejecuta cada comando en background como un trabajo en un grupo de procesos separado que no está asociado con tu terminal. De otra forma, si no has redireccionado la entrada estándar, **ksh** fija la entrada estándar para el comando al fichero vacío **/dev/null**, para prevenir que el comando intente leer del terminal a la misma vez que lo está haciendo la Shell.

Si la opción **monitor** está activa, **ksh** visualiza un mensaje de finalización para cada comando en background que se haya completado antes de visualizar el prompt. Si está fijada la opción **notify**, el mensaje de finalización será visualizado tan pronto como finalice el comando en background. **Versión**: La opción **notify** está disponible solamente en versión de **ksh** posteriores al 16 de Noviembre de 1988.

Si la opción **bgnice** está activa, entonces **ksh** ejecuta los trabajos en modo background con una prioridad menor. La opción **bgnice** está por defecto activa para los shells interactivos.

Cuando te sales del sistema, **ksh** envía una señal **HUP** a cada trabajo en background. Utilice el comando **disown** para evitar que **ksh** le envíe la señal **HUP** a los trabajos background dados cuando te salgas del sistema. **Versión**: El comando **disown** está disponible solamente en versiones de **ksh** posteriores a la versión del 16 de Noviembre de 1988.

Utilice el comando **nohup** para hacer que un trabajo ignore cualquier señal **HUP** que reciba, de forma que pueda continuar su ejecución incluso después de que te salgas del sistema. La salida estándar y el error estándar son redireccionadas para que sean añadidas al fichero llamado **nohup.out** en el directorio de trabajo. **nohup** toma como sus argumentos el nombre de comando y los argumentos del comando que deseas ejecutar.

Ejemplos:

```
53$ find /usr/morris –name foobar –print > saveout 2>&1 &
[2] 2347
54$ disown %2
55$ nohup find /usr/morris –name foobar –print &
[3] 2348
```

Utilice **tail –f nohup.out** para hacer un seguimiento del progreso de un comando ejecutándose en modo background con **nohup**. **tail** visualiza unas pocas líneas del final de un fichero. La opción **–f** hace que **tail** mantenga un chequeo de salida adicional cuando llegue al final del fichero **nohup.out**. Utilice la tecla Interrupción para hacer que **tail –f** termine. Esto no afectará al comando **nohup** debido a que se está ejecutando en modo background. El siguiente ejemplo informa del progreso del ejemplo precedente.

Ejemplo:

```
56$ alias ckhup='tail –f nohup.out'
57$ ckhup
/usr/morris/shell/book/foobar
```

CONTROL DE JOBS O TRABAJOS

El control de trabajos te permite gestionar la ejecución de trabajos en background y en foreground. **ksh** proporciona solamente un subconjunto de las características de control de trabajo en algunos sistemas debido a las limitaciones del sistema operativo; las limitaciones típicas incluyen la carencia de los comandos **bg** y **fg**, y CONTROL+z para detener un trabajo.

La opción **monitor** deber estar fijada para que esté activo el control de trabajos.

Dependencias de la Implementación:

- En sistemas que permiten un control de trabajos completo, la opción **monitor** es activada implícitamente para las invocaciones interactivas de **ksh**.
- En otros sistemas, debes especificar esta opción con **set –o monitor**. Normalmente harás esto en tu fichero de entorno.

A cada pipeline que ejecutas se le llama trabajo. **ksh** le asigna a cada trabajo un número pequeño. Si la opción **monitor** está activa, entonces **ksh** visualiza el número de trabajo encerrado en **[ ]** cuando comienza cada trabajo en background. Antes de mostrarte el prompt, **ksh** te visualiza un

mensaje de estado para cada trabajo background que ha finalizado cuando la opción **monitor** está activa.

Puedes referirte a un trabajo por su identificador de proceso, por su número de trabajo, o por su nombre de trabajo con **bg**, **disown**, **fg**, **jobs**, **kill**, y **wait**. El nombre de trabajo es el comando que introdujiste para ejecutar el trabajo. Para referirte a un trabajo por su número o nombre utilice:

%número Para referirte al trabajo por su número

%cadena Para referirte al trabajo cuyo nombre comienza con *cadena*.

%?cadena Para referirte al trabajo cuyo nombre contiene *cadena*.

%+ o %% Para referirte al trabajo actual

%- Para referirte al trabajo anterior

Utilice **jobs** para visualizar la lista de trabajos en background y sus estados.

Puedes hacer que **ksh** espere a que un trabajo en background específico finalice, o hacer que espere a que todos los procesos en background finalicen, con el comando **wait**.

Puedes hacer que un trabajo foreground (en primer plano) reciba una señal de interrupción, **INT**, presionando Interrupción. La mayoría de los programas finalizan cuando reciben esta señal. En sistemas que soportan la señal **QUIT**, puedes hacer que un trabajo en primer plano reciba la señal **QUIT** pulsando Quit. Algunos sistemas generan un "*core dump*" cuando haces esto. Podrías ser capaz de utilizar el "*core dump*" para hacer una depuración.

Puedes terminar un trabajo o proceso en background mandándole una señal **TERM** con el comando **kill**.

Los sistemas que tienen un control de trabajo completo, te permiten detener un proceso y reestablecer la asociación del proceso con tu terminal. En estos sistemas, puedes utilizar **ksh** para detener trabajos y para pasar trabajos a background o traerlos a primer plano. Los siguientes tres párrafos y los ejemplos se aplican a sistemas que tienen esta capacidad.

Pulse *Suspend* (suspensión), normalmente CONTROL+z, para detener el trabajo que se está ejecutando en primer plano. **ksh** visualiza un mensaje cuando el trabajo se detiene, y te muestra un prompt.

Un trabajo en modo background se detendrá cuando intente leer de tu terminal. Utilice **stty tostop** para especificar que un trabajo background se detenga cuando intente mostrar la salida al terminal. Utilice **kill** para enviar una señal **STOP** para detener un proceso en background. El alias **stop='kill –STOP'** es útil si detienes con frecuencia trabajos en background.

Utilice **fg** para pasar un trabajo desde modo background a primer plano o para continuar un trabajo que has detenido y ejecutarlo en primer plano. Utilice **bg** para continuar un trabajo que has detenido y ejecutarlo en background.

Ejemplos:

```
58$ sleep 60
CONTROL+z
^Z[1] + Stopped (SIGTSTP)        sleep 60
59$ bg
[1]     sleep 60&
60$ jobs
[1] +  Running                 sleep 60
61$ kill %s
[1] + Terminated               sleep 60
```

COMANDOS COMPUESTOS

Los comandos compuestos son descritos en el capítulo “Lenguaje de programación” que viene a continuación y son presentados en más detalle en el capítulo “Comandos Compuestos”. Utilizas de forma ordinaria los comandos compuestos tales como **if, while, select** y **for** cuando escribes scripts. Sin embargo, una vez que te familiarices con estos comandos, puedes introducir también cualquiera de estos comandos compuestos de forma interactiva.

La mayoría de los comandos compuestos son escritos normalmente en más de una línea. Teclea RETURN a continuación de cualquier comando dentro de un comando compuesto para hacer que el comando continúe en otra línea; en este caso, **ksh** visualiza el prompt secundario (el valor de la variable **PS2**). **Nota**: **ksh** no comienza la ejecución hasta que no introduces el comando completo.

El siguiente ejemplo utiliza el comando compuesto **for** (descrito en el siguiente capítulo). La cadena ***[0-9]** es un patrón que encaja con todos los nombres de ficheros del directorio actual que acaban en un dígito. Para cada fichero, este ejemplo crea un nuevo fichero cuyo nombre es el nombre inicial de dicho fichero pero añadiéndole al final **.new**. El comando **tr**, como se especifica, convierte los caracteres en minúsculas del fichero original a mayúsculas en el nuevo fichero que se crea.

Ejemplo:

```
62$ for i in *[0-9]
> do print -- "$i"
>       tr "[a-z]" "[A-Z]" < $i > $i.new
> done
fichero1
fichero2
```

SHELL SCRIPTS

Utilice un editor para poner un grupo de comandos **ksh** en un fichero para crear un comando nuevo. A los ficheros que contiene comandos de **ksh** se les conoce como Shell Script, o simplemente un Script a secas. Un script se comporta como cualquier otro programa que invoques. Para ejecutar un script, teclea su nombre seguido de sus argumentos separados éstos por espacios en blanco o tabuladores de la misma forma que harías con cualquier otro comando. Esto es conocido como invocación de un script por nombre. Puedes ejecutar también un script ejecutando el comando **ksh** y pasándole como parámetros el nombre del script y los argumentos del script que deseas ejecutar.

Scripts especiales son también procesados por **ksh** cuando entras al sistema o invocas a **ksh** de forma interactiva. Estos scripts te permiten personalizar tu entorno.

No es necesario compilar tu script después de escribirlo, pero tienes que hacer que el fichero sea ejecutable con **chmod +x** para ejecutar el comando por su nombre. Si escribes scripts frecuentemente, podrías definir un alias tal y como éste **cx=’chmod +x’**.

Utilice la sintaxis de comentario # para que tu script sea más fácil de leer. **Precaución**: Algunos sistemas procesan la combinación **#!** de manera especial cuando son usados como los primeros dos caracteres de la primera línea de un script.

También, utiliza la indentación en tu script para hacerle más legible. Vea los ejemplos en el capítulo “Programación de aplicaciones” para conocer un estilo de indentación sugerente.

ksh lee, expande y ejecuta los comandos de tu script de forma secuencial hasta que recibe una señal de terminación o hasta que no haya más comandos para ejecutar. Ver el capítulo “procesamiento de comandos” para una descripción de cómo se leen y expanden los comandos. Más adelante podrás ver una lista de condiciones que hacen que un script finalice su ejecución.

Ejemplo:

```
63$ alias cx=’chmod +x’
64$ print ‘print hello world’ > world
65$ cx world
66$ world
hello world
67$ ksh world
hello world
```

5. LENGUAJE DE PROGRAMACIÓN

INTRODUCCIÓN A LOS PARÁMETROS

Las variables de la shell fueron presentadas en el capítulo precedente. Una variable es uno de los tres tipos de parámetros que **ksh** entiende. Ya que las variables son usadas principalmente cuando se escriben scripts, muchos aspectos de su uso no fueron discutidos en el capítulo precedente y serán discutidos en éste. El capítulo "Parámetros" describe en detalle cada parámetro y variable que usa **ksh**.

Un parámetro se utiliza para almacenar y manipular información. Un parámetro puede ser cualquiera de los siguientes:

- Variable: Un nombre de variable consiste en uno o más identificadores separados por un **.** (punto) y se le llama *varname* (nombre de variable). Las variables fueron presentadas brevemente en el capítulo anterior. Asígnale valores a las variables con comandos de asignación de variables. Los comandos de asignación de variables presentan la siguiente forma *nombre=valor* ... . No puedes introducir ningún espacio o tabulador antes o después del signo =. **Versión**: Los nombres de variables fueron restringidos a un único identificador en la versión de 16 de Noviembre de 1988 de **ksh**.
- Parámetros Posicional: El nombre para un parámetro Posicional es un número. Los parámetros posicionales son usados principalmente en los scripts.
- Parámetro Especial. El nombre de un parámetro especial es uno de los siguientes caracteres: *** @ # ? - $!** Los parámetros especiales son descritos en más detalle más adelante. Los parámetros especiales son fijados por **ksh**. Por ejemplo, el parámetro **$** se expande al identificador de proceso del entorno de shell actual, y el parámetro **!** se expande al identificador de proceso del proceso background creado más recientemente.

Accedes al valor de un parámetro precediendo su nombre con el **$** y rodeando el nombre con **{}**. Puedes pensar que el significado del **$** es "el valor de". Los **{}** son opcionales cuando el nombre del parámetro es un dígito único o si el nombre es un identificador único o un parámetro especial, a menos que el parámetro venga seguido por caracteres que pudiesen ser considerados partes del nombre de parámetro. Por ejemplo, **$$** significa "el valor del parámetro especial **$**".

Puedes especificar que un parámetro se expanda en cualquier parte, incluso como parte de una palabra, por ejemplo **${foo}.c**

PARÁMETROS POSICIONALES

Los parámetros denotados por números son llamados parámetros posicionales.

Cuando invocas a un script, **ksh** almacena los argumentos del script como parámetros posicionales. **ksh** fija el parámetro Posicional **0** al nombre del script y fija el resto de parámetros posicionales desde el **1** en adelante al resto de argumentos respectivamente.

Referencia a los parámetros posicionales de alguna de las siguientes maneras:

$4 El cuarto parámetro. Cuando el número de parámetro es superior a 9, es necesario encerrar el número de parámetro dentro de llaves, por ejemplo **${20}**

$# Número de parámetros posicionales, sin tener en cuenta el parámetro 0 (nombre del script)

"$*" Un argumento consistente en todos los parámetros posicionales desde el 1 en adelante

"$@"** Número de argumentos **$# que consiste en todos los parámetros posicionales desde el 1 en adelante. Las comillas dobles previenen de que los valores de parámetros con espacios, tabuladores o saltos de línea sean partidos en argumentos separados, y previene al parámetro posicional Nulo de ser eliminado. Esto tiene el efecto de expandirse a **"$1" "$2" "$3" ...** para tantos parámetros como existan.

"${@:3}" Todos los parámetros posicionales desde el de la posición 3 en adelante

"${@:3:7}" Parámetros desde el 3 hasta el 7, ambos inclusive

Versión: Las últimas dos formas solamente están disponibles en versiones de **ksh** posteriores a la de 16 de Noviembre de 1988.

Ejemplos:

```
68$ print 'print hello $1' > hi
69$ cx hi
70$ hi there world
hello there
71$ print 'print hello ${@:2:3}' > hi
72$ hi there how are you feeling
hello how are you
```

Utilice **set** para asignarle valores nuevos a los parámetros posicionales a partir del parámetro posicional 1. No puedes asignarle valores a los parámetros posicionales individualmente. Puedes solamente reemplazar un conjunto de parámetros posicionales con otro.

Utilice **set - -** sin argumentos para alterar los parámetros posicionales.

Utilice **set –s** para ordenar los parámetro posicionales basándose en los pesos de colación del lugar (locale) actual. En el lugar (locale) por defecto, la posición en el conjunto de caracteres ASCII se utiliza para determinar el orden de ordenación.

Utilice **shift** para mover los parámetros posicionales hacia la izquierda. **ksh** descarta tantos parámetro(s) como veces se ejecute **shift**, empezando con el parámetro posicional 1.

Ejemplos:

```
73$ set –– foo bar bam
74$ print "$@"
foo bar bam
75$ set –s
76$ print "$@"
bam bar foo
77$ shift 2
78$ print "$@"
foo
```

MÁS SOBRE LOS PARÁMETROS

ksh no limita la longitud del nombre de variables o su valor. Elige nombres para las variables de forma que el script sea más entendible y fácil de leer. Recomendamos utilizar minúsculas para variables que son locales a tu script. Para concatenar el valor de dos o más parámetros juntos, referencia a uno después de otro, por ejemplo **foo1**.

Cuando referencias a un parámetro, puedes modificar el valor de su expansión siguiendo al parámetro dentro de las llaves con unos de los siguientes modificadores y una palabra (ver **Parámetros** para una lista completa). Si la palabra que sigue al modificador de parámetro se necesita para completar la expansión de parámetro, **ksh** realiza la substitución de comandos y la expansión de parámetro sobre él antes de utilizarlo. Utilice el modificador de parámetro para especificar:

= Un valor a ser usado si el parámetro no está fijado; por ejemplo, **${1-por_defecto}**. Utilice un **:** delante del **=**, si *por_defecto* va a ser usado si el parámetro es Nulo o no está fijado.

? Un mensaje a ser visualizado en el error estándar si el parámetro no está fijado. Utilice un **:** delante del **?**, para hacer que el mensaje sea visualizado si el parámetro es Nulo o no está fijado. Los Shells no interactivos se salen después de imprimir este mensaje.

Que la porción inicial más pequeña del valor que encaje con el patrón que se especifique a continuación del # sea descartada.

Que la porción inicial más grande del valor que encaje con el patrón que se especifique a continuación del ## sea descartada.

% Que la porción final más pequeña del valor que encaje con el patrón que se especifique a continuación del % sea descartada.

%% Que la porción final más grande del valor que encaje con el patrón que se especifique a continuación del %% sea descartada.

: Que solamente la subcadena que comienza en la posición aritmética de carácter especificada a continuación del **:** será expandida. La primera posición de carácter es la 0. Un segundo **:** y una longitud puede ser también especificado

/ Un patrón a ser buscado y una cadena para reemplazarlo. Solamente la primer ocurrencia del patrón será reemplazada.

// Un patrón a ser buscado y una cadena para reemplazarlo. Todas las ocurrencias del patrón serán reemplazadas.

Versión: Las últimas tres formas están solamente disponibles en las versiones de **ksh** posteriores a la de 16 de Noviembre de 1988.

Ejemplos:

```
${foo%??}     # Expande foo y elimina los últimos tres caracteres
${0##*/}      # Expande el pathname del script a un nombre de fichero.
${d-$(date)} # Expande el valor de d si está fijado;
              # si no, expande a la salida la fecha.
${d:3}        # Expande el valor de d, comenzando en el cuarto carácter
${d:3:2}      # Expande el valor de d al cuarto y quinto carácter
${d/x/foo}    # Expande el valor de d, reemplazando la primera x encontrada por foo
${d//x/foo}   # Expande el valor de d reemplazando todas las x por foo
```

Precaución: A menos que la referencia a un parámetro se encuentre dentro de unas comillas dobles, los espacios iniciales y finales son descartados cuando el parámetro se usa como argumento de comando.

El valor de **${#parámetro}** es el número de caracteres del valor de parámetro.

A cada variable se le puede asignar uno o más atributos o tipo de datos. Algunos atributos afectan al valor de la variable. Utilice **typeset** para asignar o cambiar cualquier atributo de una variable. Algunos atributos son utilizados para especificar cuantos espacios ocuparan las variables cuando sean visualizadas. Esto es útil para formatear informes. Para controlar el ancho y la justificación de su valor, utilice:

typeset –L [width] Para especificar una variable de ancho fijo justificada a la izquierda. Cuando le asignas un valor que es más ancho que el ancho establecido, **ksh** descarta los caracteres de sobra del final de la cadena. Si el valor es más estrecho que el ancho establecido, **ksh** añade espacios al final del valor hasta completar el ancho.

typeset –R [width] Para especificar una variable de ancho fijo justificada a la derecha. Cuando le asignas un valor que es más ancho que el ancho establecido, **ksh** descarta los caracteres de sobra del inicio de la cadena. Si el valor es más estrecho que el ancho establecido, **ksh** añade espacios al principio del valor hasta completar el ancho.

typeset –Z [width] Para especificar un campo de ancho fijo que se comporta como un atributo justificado a la derecha, excepto que si le asignas un valor que comience con un dígito y sea más estrecho que el ancho establecido, se insertarán cero(s) al principio del valor hasta completar el ancho establecido.

Ejemplos:

```
typeset –L4 x       # x es de 4 caracteres, justificados a la izquierda
typeset –R5 y=7     # y es de 5 caracteres, justificados a la derecha
typeset –L1 z       # z será solamente el primer carácter
typeset –Z3 n=7     # n valdrá 007
```

Para especificar si las variables de caracteres alfabéticos estarán en mayúsculas o minúsculas, utilice:

typeset –u Para hacer que todos los caracteres en minúsculas sean convertidos a mayúsculas.

typeset –l Para hacer que todos los caracteres en mayúsculas sean convertidos a minúsculas.

Ejemplos:

```
typeset –u x=abc          # x valdrá ABC.
typeset –L2 –l y=ABC      # y tendrá el valor ab.
print x=$x y=$y
x=ABC y=ab
```

VALORES DE RETORNO

Cada comando tiene un valor de retorno. El valor de retorno de un comando se usa con los comandos condicionales e iterativos que se tratarán posteriormente. El valor de retorno de un comando que no termina debido a una señal es un número que va desde 0 a 255. Un valor representa:

0	Salida normal. Cuando se usa en comandos iterativos y condicionales, este valor representa verdadero (True). Cualquier otro valor significa que el comando devuelve falso (*False*)
1-125	Fallo
126	Se encontró un comando pero no puede ser ejecutado
127	Un comando no pudo ser encontrado
128-255	Fallo
256 en adelante	Un comando ha finalizado debido a la recepción de una señal. Utilice la opción **–l** de **kill** para determinar el nombre de la señal que hice que el comando finalizase, o réstale 256 para obtener el número.

Versión: Con la versión de 16 de Noviembre de 1988 de **ksh**, un valor entre 129 – 160 indicaba que un comando había finalizado debido a que había recibido una señal. Restándole 128 se determina el número de señal que hizo que acabase el comando. Un comando que no podía ser encontrado o no podía ser ejecutado devolvía un valor de 1.

ksh le da al parámetro **?** el valor de retorno del último comando ejecutado. El valor de retorno se documenta con cada comando de este manual.

Asegúrate de que cada script o función que escribas tiene valores de retorno razonables. El valor de retorno debería ser **0** (cierto) si se ejecutó correctamente. El valor de retorno de un script o función es el valor de retorno del último comando que ejecuta – o fijado explícitamente por un comando **exit** o **return**, o fijado explícitamente por cualquier otro comando.

Utilice **exit** dentro de un script para hacer que el script finalice con el valor de retorno que especifiques.

MÁS SOBRE ENTRECOMILLADO

El entrecomillado restablece el significado literal a los caracteres que son procesados de forma especial por **ksh**. Las comillas literales (simples), **‘ … ‘**, causan que todos los caracteres colocados dentro de ellas mantengan su significado literal. Las comillas simples no son pasadas al comando.

Las cadenas ANSI C, **$’…’**, hacen que todos los caracteres dentro de ellas mantengan su significado literal a excepción de las secuencias de escape, las cuales son precedidas por un carácter escape (****) y son interpretadas como secuencias de escape ANSI C que se describen posteriormente. El **$** es descartado. **Versión**: Este mecanismo de entrecomillado está disponibles solamente en las versiones de **ksh** posteriores al 16 de Noviembre de 1988.

Las comillas de agrupación (dobles), **“...”**, permiten que tengan lugar la expansión de parámetros, expansión aritmética y la substitución de comandos, pero procesa el resto de caracteres literalmente. Solamente los caracteres **$, `,** y **** son especiales con **“…”**. Cuando utilizas **“...”**, **ksh**:

- Expande los parámetros y comandos contenidos dentro de las comillas dobles.
- Trata literalmente los caracteres de la expansión resultante
- Elimina las comillas de agrupación (dobles)

Una cadena, entrecomillada con comillas dobles, precedida por un **$** representa una cadena que necesita ser traducida al lenguaje local cuando no esté en el locale (sitio) de C o POSIX. El **$** es descartado. **Versión**: Esta característica está disponible solamente con versiones de **ksh** posteriores a la de 16 de Noviembre de 1988.

Utilice el carácter escape, ****, para entrecomillar el siguiente carácter. Dentro de una cadena entrecomillada con comillas dobles, **** mantiene su significado literal excepto cuando es seguido de un **$**, **`**,**"**, o ****.

Precaución: Utilice comillas dobles alrededor de las palabras de comando que contienen referencias a parámetros para evitar que el valor sea partido en argumentos separados y para prevenir las expansión del pathname en el valor. También, si una palabra de comando se expande a Nulo y no está encerrada en **"…"**, **ksh** no crea un argumento de comando. Por lo tanto, utilice **"…"** para especificar un argumento de comando obligatorio como un parámetro que pudiese expandirse a Nulo. Las comillas dobles no son necesarias para referencias de parámetro dentro del valor de asignaciones de variables porque la partición en campos y expansión de pathname no son realizadas allí.

Ejemplos:

```
x='foo bar'          # La variable x tiene un espacio en su valor
"|$x|"               # Se expande a un argumento simple, |foo bar|.
*foo\ bar\**         # Encaja con cualquier cosa que contenga foo bar*.
y=                   # Le asigna a y la cadena nula
set -- $y "$y"       # Expande a un argumento nulo. La primera $y es descartada
x=$'\E]a\'\n'        # Le asigna la cadena consistente en los caracteres ESC ] a '
                     # Newline a la variable x
x=$"hello"           # Le asigna hello a la x en el locale (sitio) C o POSIX.
                     # De otra forma, busca y reemplaza hello.
```

MATCHING O "ENCAJE" CON PATRONES

Excepto por la sintaxis, los patrones son similares a las expresiones regulares que son usadas por **grep** y **ed**. De hecho, la conversión de formato **%P** de **printf** convertirá una expresión regular extendida a un patrón de shell de forma que puede ser usado dentro de un shell script. **Versión**: La conversión de formato **%P** está disponible solamente en versiones de **ksh** posteriores a la de 16 de Noviembre de 1988.

Los patrones fueron presentados brevemente en relación con la expansión de los pathname en el capítulo 3. Además de en la expansión del pathname, **ksh** utiliza los patrones en los comandos compuestos **case** y **[[...]]**, y para expansiones de subcadenas. Las reglas completas para formar patrones pueden ser encontradas empezando en la página xxx.

Además de *****, **?**, y **[**, los patrones pueden ser formados con cualquiera de los siguientes:

?(patrón[|patrón]...)	Para encajar con cero o uno de los patrones especificados
*(patrón[|patrón]...)	Para encajar con cero o más de los patrones especificados
+(patrón[|patrón]...)	Para encajar con uno o más de los patrones especificados
@(patrón[|patrón]...)	Para encajar exactamente con uno de los patrones especificados
!(patrón[|patrón]...)	Para encajar con cualquier cosa excepto con los patrones especificados

Los patrones pueden estar formados utilizando cualquier combinación de estas expresiones de patrón y caracteres regulares. Los grupos de patrones con paréntesis son llamados sub-patrones.

La porción de la cadena que encaja con un sub-patrón puede ser referenciada posteriormente en el patrón utilizando una referencia hacia atrás. Una referencia hacia atrás (*backreference*) se indica por ***dígito***, donde *dígito* es el número del paréntesis izquierdo del sub-patrón empezando por la izquierda. **Versión**: La referencia hacia atrás está disponible solamente en versiones de **ksh** posteriores a la de 16 de Noviembre de 1988.

Ejemplos:

```
@(foo|bar|bam)          # Encaja con foo, bar o bam
?(foo|bar|bam)          # Encaja con foo, bar, bam o nulo
+([0-9])?(.)*([0-9])    # Encaja con uno o más dígitos, opcionalmente seguidos
                        # de un punto decimal y cualquier número de dígitos
!(*.o)                  # Encaja con cualquier cadena que no finalice en .o
@(¿)*\1*\1              # Encaja con cadenas que empiecen y terminen con el mismo
                        # carácter y que contengan ese mismo carácter en algún
                        # lugar entre medias.
```

ABRIENDO Y CERRANDO FICHEROS

Utilice **exec** sin argumentos para abrir y cerrar ficheros en el entorno actual. Utilice la sintaxis de redirección de entrada/salida definida en la página xxxx para especificar los ficheros que quieres abrir o cerrar. Puedes abrir y/o cerrar descriptores de ficheros desde **0** a **9**. **ksh** fija el bit "*close-on-exec*" sobre los descriptores de ficheros del **3** al **9**.

Utilice la redirección de la entrada/salida sin un nombre de comando para crear un fichero o para eliminar el contenido de un fichero existente. **ksh** abre o crea el fichero y después lo cierra.

El parámetro **$** se expande al identificador de proceso, el cual es un número que es único para todos los scripts que se están ejecutando al mismo tiempo. Utilice **$$**, el valor del parámetro **$**, dentro de un pathname para generar el nombre de un fichero temporal para asegurarte que el nombre de fichero temporal en ese momento es único.

Ejemplos:

```
exec 3< foo  # Abre foo para lectura, como descriptor de fichero 3.
exec 3<&-    # Cierra el descriptor de fichero 3.
>/tmp/foo$$  # Crea un fichero temporal
exec 3<> foo # Abre foo para lectura y escritura como descriptor de fichero 3.
```

ksh procesa ciertos nombres de fichero de forma especial cuando se especifican después de un operador de redireccionamiento. Un nombre de fichero de la forma **/dev/fd/***número* se trata como si significase utilice el descriptor de fichero ***número***.

Los nombre de fichero de la forma **/dev/tcp/***iden_host/id_puerto* y **/dev/udp/***iden_host/id_puerto* son tratados como conexiones socket utilizando los protocolos **tcp** y **udp** respectivamente. *iden_host* es una dirección de la forma n.n.n.n, donde cada *n* representa cualquier número. El *iden_host* **0.0.0.0** puede ser utilizado para referirse a los servicios de tu máquina. El *id_puerto* es el número del servicio, por ejemplo 13 es el servicio de fecha y hora. Esto te facilita la conexión a los servicios de Internet directamente desde la shell. Los medios para obtener el *iden_host* de una máquina, y el *id_puerto* de un servicio en particular son dependientes del sistema y va más allá del propósito de este manual. **Versión**: Esta característica está disponible en versiones de **ksh** posteriores a la versión de 16 de Noviembre de 1988 y algunas versiones de la versión del 16 de Noviembre de 1988.

Ejemplos:

```
exec 3< /dev/fd/5         # Equivalente a exec 3<&5
cat < /dev/tcp/0.0.0.0/13 # hace un cat del servidor de fecha/hora
Tue Jul 31 10:04:35 2001
```

LEYENDO DE TERMINALES Y FICHEROS

Por defecto, la entrada de datos a **ksh** es orientada a la línea. **ksh** no procesa su entrada hasta que lee un carácter Newline (salto de línea). Utilice:

- **read** para leer de cualquier fichero abierto. **read** lee una línea cada vez
- **read –u***n* para especificar el descriptor de fichero n. Si omites el **–u**, el valor por defecto es **0** (entrada estándar)

Normalmente especificas **read –r** cuando lees de ficheros. Si no especificas **–r** y el carácter que precede al carácter Newline (salto de línea) es el carácter ****, entonces **ksh** descarta el carácter **** y une la línea actual con una lectura (**read**) de la siguiente línea.

read lee una línea y asigna los caracteres a la variable(s) que le especificas como argumento. Si especificas:

- Ninguna variable, **ksh** asigna los caracteres que lee a la variable **REPLY**
- Una variable, **ksh** asigna los caracteres que lee a la variable que especificas
- Más de una variable, **ksh** parte la línea que lee en campos utilizando los caracteres contenidos en la variable **IFS** según sigue:
    - Cada carácter **IFS** finaliza un campo. Cada campo es asignado a la siguiente variable
    - Dos caracteres **IFS** adyacentes en la línea, ninguno de los cuales pertenece a la clase de caracteres **espacio**, indican un campo cuyo valor es Nulo; por ejemplo, si **IFS** es **:**, entonces **a::b** será dividido en tres campos, **a**, **Nulo** y **b**.
    - **ksh** asigna los caracteres que sobren a la última variable
    - **ksh** les asigna a las variables sobrantes el valor Nulo, si especificas más variables de las necesarias.

Para especificar un prompt que es visualizado cuando lees de un valor de un terminal, especifica el prompt a continuación del signo **?** que sitúes a continuación del nombre de variable a leer.

Si la entrada a leer viene de un terminal y la opción editor está activa, entonces las directivas de editor pueden ser utilizadas cuando se introducen líneas. La opción **–s** hace que cada línea que sea leía sea salvada en el fichero histórico. (ver la página xxxx)

Puedes hacer que **ksh** lea hasta que se encuentre un determinado carácter, ***c***, en la entrada, en lugar de leer hasta que se introduce el Newline, utilizando la opción **–d *c***.

Cuando se lee de un terminal o de un pipe, puedes utilizar la opción **–t *timeout*** para limitar el tiempo de espera para que se produzca la entrada a ***timeout*** segundos. El valor de retorno será Falso si se agota el tiempo para el **read**.

Versión: Las opciones **–d** y **–t** están solamente disponibles en versiones de **ksh** posteriores a la versión del 16 de Noviembre de 1988.

Ejemplos:

```
read –r                          # Lee una línea en la variable REPLY
read –rs                         # También copia la línea al fichero histórico
read –r linea                    # Lee una línea en la variable linea
read –u3 linea                   # Lee del descriptor de fichero 3
read –r linea?"Introduce foo "   # Visualiza como prompt de la entrada "Introduce
                                 # foo"
read –d : campo                  # Lee hasta el carácter : en la variable campo
read –t 15 linea                 # Espera hasta 15 segundos para leer una línea en
                                 # la variable linea.
IFS=: read –r a b c              # Lee una línea en las variables a, b y c
                                 # utilizando un símbolo : para delimitar los
                                 # campos
```

ESCRIBIENDO A TERMINALES Y FICHEROS

Utilice **print** o **echo** para visualizar líneas en un terminal y para escribir líneas en un fichero. **Precaución**: El comportamiento de **echo** es dependiente del sistema y es proporcionado por compatibilidad con la shell de Bourne. Utilizamos el comando **print** en todos los programas y ejemplos de este manual.

Normalmente **ksh** reemplaza ciertas secuencias que comienzan con ****, según se describen en la página xxxx, y después visualiza cada uno de los argumentos de **print** a la salida estándar seguido e un carácter Newline. **Precaución**: Debes entrecomillar las secuencias de escape para evitar que **ksh** elimine el **** y expanda el argumento a **print**.

Especifique:

–r	Para evitar que las secuencias de escape sean reemplazadas. Utilice **–r** para visualizar un parámetro cuyo valor puede contener una secuencia de escape.
–u n	Para dirigir la salida al descriptor de fichero **n**. Si se omite, **ksh** utiliza el descriptor de fichero **1**.
–n	Para evitar que se añada un carácter Newline a la salida
–s	Para hacer que la salida sea añadida al fichero histórico como un comando.
– –	Para indicar que los argumentos que siguen no son opciones del comando **print**. Utilice – – para visualizar cualquier cosa que pudiese empezar con un –. Los ejemplos de este manual utilizan – – cuando quiera que el siguiente argumento es una referencia a parámetro, para evitar problemas que tendrían lugar cuando el parámetro se expande a una cadena que comience con –.
–f formato	Para imprimir los argumentos según es especificado por la especificación de formato de ANSI C. No se añade el carácter Newline al final. Si hay mas argumentos que especificaciones de formato, el formato es reutilizado. El comando **printf** es equivalente a **print –f**. **Versión**: Este mecanismo de formateo está disponible solamente en versiones de **ksh** posteriores a la de 16 de Noviembre de 1988.

Ejemplos:

```
print –r –– "$foo"         # Visualiza una línea con el valor foo.
print –u2 –r –– "$foo"     # Igual que el ejemplo anterior pero al descriptor de
                           # fichero 2.
print –rs –– "$foo"        # Añade una línea con el valor foo al fichero histórico
print –n "\t\t"            # Visualiza dos tabuladores y permanece en la misma línea
print –f '%s\t%s\n' *      # Visualiza los ficheros del directorio actual, dos
                           # ficheros por línea.
```

HERE-DOCUMENTS (DOCUMENTOS EMPOTRADOS)

Puedes especificar que uno o más comandos de tu script lean la entrada de líneas dentro de tu propio script, en lugar de tu terminal o un fichero. Utilice el operador de redireccionamiento de entrada/salida **<<**, seguido por una palabra delimitadora arbitraria. Las líneas que empiezan después del siguiente Newline (salto de línea) y continúan hasta una línea que contenga solamente la palabra delimitadora, son llamadas here-document. Un here-document es también algunas veces referidos como datos in-line.

Normalmente, **ksh** realiza expansiones de parámetros, expansiones aritméticas y substitución de comandos en el here-document antes de que los contenidos sean pasados al comando especificado. Los caracteres **$**, `, y **** son especiales y deberían ser siempre precedidos por un **** si quieres que ellos mantengan su significado literal. Sin embargo, si entrecomillas cualquier carácter de la palabra delimitadora de cualquier forma, entonces el here-document se le pasa al comando sin ninguna expansión.

Si especificas el operador **<<–** en lugar de **<<**, entonces **ksh** elimina los tabuladores de principio de línea de cada línea del here-document antes de pasarlo al comando. Como se ilustra en el siguiente ejemplo, los tabuladores que preceden a la palabra delimitadora son también ignorados. Los tabuladores pueden hacer tu script más fácil de leer.

Puedes utilizar un here-document como la base para un generador de impresos de cartas o generador de programas. Especifique **cat** con la plantilla para el impreso o programa que quieras generar. Utilice la expansión de parámetros y/o la substitución de comandos para generar la información variable del impreso o programa.

Ejemplo:

```
nombre=Morris
cat <<– EOF  # Elimina los tabuladores iniciales, finaliza en el EOF siguiente
      Dear $nombre,

      Estoy escribiendo esta carta el día $(date).
      Recuerda ir al dentista a las dos y media.
      EOF
Dear Morris,

Estoy escribiendo esta carta el día Tue Jul 31 12:57:27 MET DST 2001.
Recuerda ir al dentista a las dos y media.
```

CO-PROCESOS

Desde dentro de un script, puedes ejecutar un comando o un pipeline en background que puede comunicarse con tu programa. Esto es particularmente útil cuando quieres proporcionar un nuevo interfaz a un programa existente, o escribir un programa que interactúa con un programa orientado a la transacción tal y como un gestor de bases de datos.

Para ejecutar un co-proceso en background, pon el operador **|&** después del comando. La entrada y la salida estándar del comando será cada una conectada a tu script con un pipe.

Utilice **print –p** para escribir al co-proceso, y utilice **read –p** para leer del co-proceso.

Precaución: El co-proceso debe:

- Enviar cada mensaje de salida a la salida estándar
- Tener un carácter Newline (salto de línea) al final de cada mensaje
- Vaciar su salida estándar cuando escriba un mensaje

Un shell script tiene todas estas propiedades; por lo que es posible escribir un co-proceso como un shell script.

Ejemplo:

```
ed – foobar |&
print –p /morris/
read –r –p linea
print –r –– "$linea"
Esta línea en el fichero foobar contiene morris.
```

Utilice la redirección de la entrada/salida para mover las pipes de entrada y/o salida del co-proceso a un descriptor de fichero numerado. Utilice **exec 3>&p** para mover la entrada del co-proceso al descriptor de fichero **3**. Una vez que conectas la entrada de un co-proceso a un descriptor numerado, puedes redireccionar la salida de cualquier comando al co-proceso con la sintaxis de redirección usual. Por ejemplo, **date>&3** direcciona la salida de **date** al co-proceso que ha sido movida al descriptor de fichero **3**. Puedes invocar a otro co-proceso una vez que mueves la entrada del co-proceso a un descriptor numerado. La salida de ambos co-procesos será conectada al mismo pipe. Utilice **read –p** para leer de este pipe. Utilice **exec 3<&–** para cerrar la conexión al primer co-proceso.

Utilice **exec 4<&p** para mover la conexión desde la salida del co-proceso al descriptor de fichero **4**. Utilice **read –u4** para leer del co-proceso después de mover la conexión al descriptor de fichero **4**.

Ejemplo:

```
(read; print –r "$REPLY") |&     # Comienza el co-proceso 1
[1]   258
exec 5>&p                        # Mueve al descriptor 5
(read; print –r "$REPLY") |&     # Comienza el co-proceso 2
[2]   261
exec 6>&p                        # Mueve al descriptor 6
date >&6                         # Fecha al co-proceso 2
print –u5 foobar                 # Imprime en el co-proceso 1
exec 3<&p                        # Mueve la entrada de ambos co-procesos al
                                 # descriptor de fichero 3
read –ru3 linea                  # Lee del co-proceso
print –r "$linea"                # Visualiza la linea
Sat Aug 28 12:13:05 2001
read –ru3 linea                  # Lee otra línea
print –r "$linea"                # La visualiza
```

AGRUPACIÓN DE COMANDOS

Un carácter Newline (salto de línea) finaliza los comandos. Sin embargo, puedes poner varios comandos en la misma línea separándolos con un **;** (punto y coma). Debido a que el **;** es un operador, no necesitas espacios o tabuladores antes o después de él.

Ejemplo:

```
print –n '\t'; print –r –– "$HOME"
        /home/dwadmin
```

Utilice llaves **{}** para agrupar varios comando juntos. Por razones históricas, **{** y **}** son palabras reservadas (ver la página xxxx), no operadores. Un espacio, tabulador, carácter Newline o

operador de control (ver la página xxxx) se necesita antes y después del **{** y del **}**. Si no, **ksh** leerá a las llaves como parte de la siguiente palabra. Debido a que **{** y **}** son palabras reservadas, deben aparecer como la primera palabra de un comando. El **;** (o un Newline) justo antes del **}** en el ejemplo de más abajo es requerido; si omites el **;** el **}** será procesado como un argumento de comando en lugar de cómo una palabra reservada. Sin embargo, para evitar conflictos potenciales con cambios futuros en el lenguaje, cada **{** o **}** que represente a el mismo debería ser entrecomillado. Puedes especificar redirección de la entrada/salida después del **}**. Cualquier redirección que especifiques se aplica a todos los comandos dentro del grupo a excepción de aquellos que presente otro tipo de redireccionamiento explícito. El siguiente ejemplo lee las primeras dos líneas del fichero **foobar** en las variables **linea1** y **linea2** y después visualiza el resto del fichero en la salida estándar.

Ejemplo:

```
{ read –r linea1; read –r linea2; cat;} < foobar
```

Puedes también utilizar **( )** para agrupar varios comandos juntos. Ya que **(** y **)** son operadores, no tienes que separarlos con espacios o tabuladores; sin embargo, el **(** debe aparecer como la primera palabra del comando. **( )** crea un entorno de subshell (ver la página xxxx) para ejecutar los comandos encerrados entre los paréntesis. Por lo tanto, no pueden existir efectos laterales, excepto en ficheros. Debido a que se ejecutan en un entorno de subshell, la ejecución podría ser más lenta que usando **{ }**, a menos que los comandos agrupados se ejecuten en un entorno de subshell de todas formas; por ejemplo como sucedería con un grupo de comandos entre llaves seguido de un **&**. Puedes especificar redirección de la entrada/salida después del **)**. Cualquier redirección que especifiques a continuación del **)** se aplica a todos los comandos dentro del grupo a excepción de aquellos que presente otro tipo de redireccionamiento explícito

Ejemplo:

```
( date; who ) | wc
```

ARITMÉTICA

La aritmética es denotada encerrando una expresión aritmética dentro de **((...))**. Las expresiones aritméticas utilizan los operadores, precedencias y funciones de librería matemática del lenguaje ANSI C. Ver la sección de Expresiones aritméticas en la página xxxx para obtener una lista completa de operadores, precedencias y funciones de la librería matemática.

Las operaciones aritméticas son realizadas usando la aritmética de los punto flotante de doble precisión de C. Cuando quiera que **ksh** encuentra un nombre de variable dentro de una expresión, reemplaza el nombre de variable por el valor de la variable. Por lo que un **$** precediendo un nombre de variable es innecesario y podría ser un poco más lento. **ksh** puede realizar también aritmética entera en una base aritmética desde 2 hasta 64. Utilice el formato ***base#número*** para constantes en cualquier base distinta de base 10.

Un comando de la forma **((expresión))** es llamado comando aritmético. El valor devuelto por un comando aritmético es ***Verdadero*** cuando el valor del la expresión encerrada es distinta de cero, y ***Falso*** cuando la expresión se evalúa a 0.

La construcción **$((expresión))** puede ser usada como una palabra o parte de una palabra; es substituida por el valor de ***expresión***.

La construcción **let "expresión"** es equivalente a **((expresión))**. **Precaución**: cada argumento de **let** es una cadena que se evalúa como una expresión aritmética. Por lo tanto no puedes utilizar espacios o tabuladores sin utilizar las comillas. Muchos de los operadores aritméticos tienen un significado especial para **ksh**, y deben ser entrecomillado.

Puedes especificar el atributo entero y una base aritmética para cualquier variable con **typeset –i**, o con el alias prefijado **integer**. Puedes especificar el atributo punto flotante o el atributo de notación científica para cualquier variable con **typeset –F** o **typeset –E** respectivamente. **float** es un alias prefijado para **typeset –E**. El valor de una variable entero o punto flotante es evaluado cuando especificas un valor con un comando de asignación de variable. No necesitas utilizar **let**.

Utilice la variable **RANDOM** para generar una secuencia distribuida uniformemente de números aleatorios en el rango de 0-32767. Cada vez que se referencia a **RANDOM**, se expande a un número aleatorio diferente.

Versión: La aritmética en punto flotante y bases mayores de 36 están solamente disponibles en versiones de **ksh** posteriores a la de 16 de Noviembre de 1988. Versiones anteriores tenían solamente aritmética entera.

Ejemplo:

```
float r a=1 b=1.5 c=-1
(( r=sqrt(b*b-4*a*c) ))
print r1=$(( (-b+r)/2 )) r2=$(( (-b-r)/2 ))
r1=.5 r2=-2
```

CHEQUEANDO FICHEROS Y CADENAS

ksh tiene expresiones condicionales para chequear la existencia de un fichero, sus permisos de acceso, y/o su tipo; para chequear si dos pathname se refieren al mismo fichero; y/o para chequear si un fichero es más antiguo que otro. También pueden ser usadas expresiones condicionales para comparar dos cadenas o para comparar dos expresiones aritméticas. Ver las "Primitivas de Expresión Codicional" en la página xxxx para obtener una lista de condiciones de chequeo.

Utilice el comando compuesto **[[...]]** para especificar expresiones condicionales. Debido a que las palabras dentro de **[[...]]** no están sometidas a particionamiento de campos y expansión de pathname, muchos de los errores asociados con **test** y **[** son eliminados. También, desde que **ksh** decide los operadores dentro de los **[[...]]** antes de ser expandidas las palabras, no sucede ningún problema cuando un parámetro se expande a un valor que empieza con –. Finalmente, **&&** y **||** son utilizados como operadores conectivos lógicos, y paréntesis sin entrecomillar pueden ser usados para agrupamiento.

Use **[[–z $parámetro]]** o **[[$parámetro == ""]]** para chequear si parámetros es **Nulo**. Necesitas:

- Espacios y/o tabuladores después del **[[** y antes del **]]**.
- Espacios y/o tabuladores antes y después de **–z**o **==**, ya que **–z** y **==** son argumentos separados para el comando **[[...]]**.

Versión: La primitiva de expresión condicional **==** está disponible en versiones de **ksh** posteriores a la de 16 de Noviembre de 1988 y en algunos versiones del 16 de Noviembre de 1988. Debes utilizar **=** en versiones anteriores.

Para especificar chequeos más complejos, utilice los operadores conectivos lógicos (que se listan aquí en orden de precedencia decreciente):

!	Not (negación lógica)
&&	And (y lógico)
‖	Or (o lógico)

Use paréntesis para saltarse la precedencia normal. Para el **[[...]]**, los paréntesis son parte de la gramática y no deben estar entrecomillados. Para **test** y **[**, los paréntesis son argumentos separados y deben estar entrecomillados.

Nota: Cada operador y operando debe ser un argumento separado.

Ejemplos:

```
[[ -x fichero && ¡ -d fichero ]]       # Cierto si el fichero es ejecutable y no es
                                       # un directorio.
[[ $1 == abc ]]                        # Cierto si $1 se expande a abc.
[ "X$1" = Xabc ]                       # Lo mismo que el ejemplo anterior
test "X$1" = Xabc                      # Lo mismo que el ejemplo anterior
[[ $1 –ef . ]]                         # Verdadero si $1 es el directorio de
                                       # trabajo
[[ foo –nt bar ]]                      # Verdadero si el fichero foo es más nuevo
                                       # que el fichero bar
[[ ¡ (-w fichero || -x fichero) ]]     # Verdadero si fichero no se puede ni
                                       # escribir ni ejecutar.
test ¡ \( -w fichero –o –x fichero \)  # Lo mismo que en el ejemplo anterior
```

COMANDOS COMPUESTOS

Un comando compuesto es un comando que comienza con una de las palabras reservadas que se listan en la página xxxx, o un comando agrupado, o una secuencia de uno o más comandos separados por un operador. El comando pipeline descrito anteriormente es un ejemplo de comando compuesto. Ver el capítulo de "Comandos Compuestos" para una descripción más detallada de cada uno de los comandos compuestos descritos aquí.

Utilice pipelines para realizar tareas de programación complejas. Cada elemento de un pipeline puede ser cualquier comando, incluyendo cualquiera de los comandos compuestos. El pipeline en el ejemplo siguiente lee su entrada y visualiza en la salida estándar cada palabra en orden alfabético, precedida por un contador del número de veces que aparece en su entrada. Esto se hace con una secuencia de filtros, los cuales realizan cada uno de los siguientes pasos en orden:

- Crea una lista de todas las palabras de la entrada, una por línea. La opción **–c** (complemento) de **tr**, combinada con la clase de carácter **[A–Za–z]**, hace que todos los caracteres que no son alfabéticos sean remplazados por un carácter Newline. La opción **–s** hace que varios caracteres Newline consecutivos se conviertan en un único carácter Newline.
- Convierte todos los caracteres a minúsculas
- Ordena las líneas

- Visualiza cada línea distinta precedida por un contador de cuantas veces aparece en orden de ocurrencia descendente.

Ejemplo:

```
tr –cs ‘[A-Za-z]’ ‘\n’ < fichero | tr ‘[A-Z]’ ‘ [a-z]’ | sort –r | uniq -c
```

ksh no reconoce palabras tales como **if** como palabras reservadas, a menos que aparezcan como la primera palabra de un comando. Es usual poner la mayoría de las palabras reservadas como la primera palabra de una línea. Utilice la indentación para mejorar la legibilidad.

Use **if** para realizar una ejecución condicional. **ksh** ejecuta la parte del comando **then** solamente si el valor devuelto por el comando condicional es Cierto (True). **ksh** ejecuta la parte **else** del comando solamente si el valor devuelto por el comando condicional es Falto (False). El comando **if** finaliza con la palabra reservada **fi**. El propio comando condicional puede el mismo ser un comando compuesto. En el ejemplo de más abajo, advierta que **read** devuelve Verdadero si a leído de forma exitosa una línea de datos. El comando built-in **:** (dos puntos) no tiene efecto. Se necesita porque la palabra reservada **then** debe estar seguida de un comando.

Ejemplo:

```
if print ‘Por favor introduzca su nombre: \c’
   read –r nombre
then if mail “$nombre” < mailfichero
     then :
     else print “No puedo mandar un correo a $nombre”
     fi
else print ‘end-of-file’
     exit 1
fi
```

ksh proporciona otros dos operadores para combinar comandos de forma mas compacta que con el comando **if**. El operador binario:

||	Ejecuta el comando que sigue a continuación del **||** solamente si el comando que precede al || devolvió Falso.
&&	Ejecuta el comando que sigue a continuación del **&&** solamente si el comando que precede al **&&** devolvió Cierto.

Ejemplos:

```
read –r linea || exit
cd foobar && echo $PWD
```

Utilice **case** para una ramificación de múltiples posibilidades. **case** encuentra cuál de los patrones especificados, si existe, encaja con una palabra dada, y ejecuta la lista de comandos asociada con ese patrón. Utilice **;;** o **;&** para finalizar cada lista de comandos. Si se usa **;&** en lugar de **;;**, entonces **ksh** fracasará a la siguiente lista de comandos si está lista es seleccionada. Puedes especificar una lista de patrones para la misma secuencia de comandos, separando los patrones con **|**. El siguiente ejemplo intenta encajar el valor del parámetro posicional **1** con cadenas que comiencen con **–d** o **–D**, cadenas que empiecen por **–e**, cadenas que comiencen por **–f**, o cualquier otra cosa, en ese orden. Si encaja **–e**, entonces tanto **eflag** como **fflag** serán fijados a 1. No es necesario utilizar comillas dobles alrededor de **$1** porque **ksh** no realiza particionamiento de campos o expansión de pathname en este contexto.

Ejemplo:

```
case $1 in
-[dD]*) dflag=1;;
```

```
-e*) eflag=1;&
-f*) fflag=1;;
*) print –u2 ‘Debes especificar –d, -D, o –e o –f’
   exit 1;;
esac
```

Los parámetros en las palabras de patrones de **case** son expandidos para formar el patrón. Esto hace que sea fácil construir patrones dentro de un script. **Versión**: Las palabras que se expandían a patrones que tuviesen paréntesis no funcionaban con la versión de **ksh** de 16 de Noviembre de 1988.

El siguiente ejemplo lee una línea y chequea si la línea comienza con cualquier carácter encontrado en el valor del parámetro posicional **1**.

Ejemplo:

```
read –r linea
case $linea in
[$1]*) ;; # ok
*)     print –u2 “La línea debe empezar con: $1”
       exit;;
esac
```

Utilice **while** para ejecutar un grupo de comandos repetidamente mientras el comando condicional tenga un valor de retorno de Verdadero (True).

Ejemplo:

```
while read –r linea        # lee una linea
do print –r – “$linea”     # imprime la línea
done
```

until es como **while**, a excepción de que el cuerpo del bucle se ejecuta hasta que el comando condicional tiene un valor de retorno de Cierto. El ejemplo de más abajo utiliza el valor de retorno de un comando pipeline. El valor de retorno de un comando pipeline es el valor de retorno del último comando del pipeline.

Ejemplo:

```
until who | grep morris
do sleep 60
done
```

Utilice **for** para repetir una secuencia de comandos una vez para cada ítem de una lista especificada. Antes de empezar el bucle, **ksh** expande la lista de palabras que especificaste a continuación de la palabra reservada **in**. Estas palabras son expandidas del mismo modo que **ksh** expande las palabras de comando. El cuerpo del comando **for** se ejecuta una vez para cada argumento en la lista expandida. **ksh** le asigna a la variable que actúa como índice del bucle, **i** en el ejemplo siguiente, el valor de cada argumento por turno.

Ejemplo:

```
for i in *   # Para cada fichero del directorio de trabajo
do if [[ -d $i ]]
   then print –r – “$i”  # Imprime los nombres de los subdirectorios
   fi
done
```

Utilice el comando aritmético **for** descrito en la página xxxx para ejecutar una secuencia de comandos hasta que alguna condición aritmética se satisfaga. El comando aritmético **for** es similar a la sentencia **for** del lenguaje C. **Versión**: El **for** aritmético está disponible solamente en versiones de **ksh** posteriores a la versión de 16 de Noviembre de 1988.

Ejemplo:

```
integer par=0 impar=0 count
for ((count=0; count < 100; count++))
do    if    ((RANDOM%2==0))
      then par=par+1
      else impar=impar+1
      fi
done
print par=$par impar=$impar
par=43 impar=57
```

ARRAYS

Un array es una variable que puede almacenar múltiples valores, donde cada valor está asociado con un subíndice. Existen dos tipos de arrays – los arrays indexados y los arrays asociativos. El subíndice de un array indexado es una expresión aritmética; el subíndice de un array asociativo es una cadena de texto arbitraria. No tienes que declarar que una variable es un array indexado. Utilice **typeset –A** para declarar que una variable es un array asociativo.

Puedes especificar un subíndice con una variable utilizando la notación de sintaxis de array, ***nombre_de_variable[subíndice]***. El subíndice de un array indexado puede ser cualquier expresión aritmética que se evalúe entre **0** y algún valor definido por la implementación que sea menor que 4095. El subíndice de un array asociativo puede ser cualquier cadena.

Cada atributo que especifiques se aplica a todos los elementos de un array.

Puedes asignar valores a los elementos del array de forma individual con comando de asignación de variables. Puedes asignar valores a los elementos de un array indexado de forma secuencial, comenzando en el índice 0, usando la asignación compuesta **nombre=(valor ...)** De forma alternativa puedes utilizar **set –A *nombre*** para asignar una lista de valores a un array de forma secuencial. Utilice la asignación compuesta **nombre=([subíndice]=palabra ...)** para asignar múltiples elementos a un array asociativo.

Utilice llaves alrededor del nombre de array y subíndice para referenciar al elemento de un array asociativo o indexado, por ejemplo **${foo[bar]}**. Si omites las llaves, **ksh** expandirá **$foo**, concatenándola con el patrón **[bar]**, y lo remplazará con el pathname que encaje con este patrón. Cuando referencias una variable de array sin especificar un subíndice, entonces **ksh** utiliza el elemento **0**. Utilice el subíndice @ o ***** para referirse a todos los elementos de un array.

Utilice el operador de nombre **!** para expandir la lista de subíndice en un array. Por ejemplo, **${!foo[@]}** se expande a la lista de subíndices de array para la variable **foo**. El operador de tamaño # puede ser utilizado en lugar de **!** para obtener el número de elementos del array.

Versión: Los arrays asociativos, asignaciones compuestas, y el operador de nombre **!** están disponibles solamente en versiones de **ksh** posteriores a la de 16 de Noviembre de 1988. Versiones anteriores solamente tenían arrays indexados, y algunas implementaciones limitaban el subíndice a un máximo de 1023.

Ejemplos:

```
# Lo siguiente imprime una carta aleatoria de una baraja
integer i=0
typeset -u card   # Mayusculas
for suit in treboles rombos corazones picas
do  for n in As 2 3 4 5 6 7 8 9 10 jota reina rey
    do card[i]="$n de $suit"
```

```
       i=i+1  # El comando let no hace falta con variables enteras
    done
done
print -- ${card[RANDOM%52]}
REY DE CORAZONES
# ###################################################################

# Lo siguiente ilustra la asignacion en los arrays indexados
ficheros=(*) # en Versiones posteriores a la de 16/Nov/1988
print -r -- "${#ficheros[@]}: ${ficheros[@]}"
7: fichero1 fichero2 foobar prog prog.c todo wishlist

# El siguiente ejemplo muestra los arrays asociativos
typeset -A color
color=( [manzana]=roja [mora]=violeta [banana]=amarilla )
print -- ${!color[@]}    # Imprime la lista de subindice
mora manzana banana
print -- ${color[@]}     # Imprime la lista de elementos
violeta roja amarilla
```

CREACIÓN Y UTILIZACIÓN DE MENÚS

Utilice **select** para visualizar una lista de alternativas para que el usuario elija una de ellas. **select** visualiza la lista de ítems que especificas en filas y columnas, con cada ítem precedido por un número y un paréntesis derecho. **select** utiliza las variables **COLUMNS** Y **LINES** para ayudar a formatear las selecciones del menú.

select visualiza la variable **PS3** como prompt cuando está preparado para leer la elección del usuario. La respuesta del usuario se guarda en la variable **REPLY**. Si el usuario introduce un número correspondiente a una de las opciones, a la variable de índice **select** se le asigna del valor correspondiente. De otra forma, la variable de índice de **select** toma el valor Nulo. Si el usuario introduce RETURN solamente, **ksh** vuelve a visualizar las elecciones y el prompt **PS3** y lee la selección de nuevo.

Versión: En versiones de **ksh** posteriores a la versión de 16 de Noviembre de 1988, si la variable **TMOUT** contiene un valor mayor que 0, **select** agotará su tiempo de elección una vez transcurran el número de segundo que tenga la variable **TMOUT** y el valor de retorno será Falso.

Normalmente utilizas un comando **case** dentro del cuerpo del **select** para especificar la acción apropiada a realizar.

Ejemplo:

```
PS3='Elige una de las opciones de arriba: '
TMOUT=10
select i in list edit quit
do   case $i in
     list) cat "$foo";;
     edit) ${EDITOR-vi} "$foo";;
     quit) break;;
     "") print –u2 Debes seleccionar una de las opciones de arriba;;
     esac
done
1) list
2) edit
3) quit
Elige una de las opciones de arriba:
```

USO DE "EVAL"

Un shell script es él mismo una secuencia de caracteres que pueden ser almacenados en una variable. El comando **eval** puede ser utilizado para procesar sus argumentos como si él (el comando **eval**) fuese un shell script. Debido a que **eval** es un comando, sus argumentos son expandidos antes de ejecutarse **eval**. **eval** crea un script concatenando sus argumentos cada uno separado por un espacio sin entrecomillar, y después ejecutando la cadena resultante como si fuese un shell script.

El capítulo "Procesamiento de comandos" explica el orden en el cuál **ksh** procesa un comando para construir el nombre de comando y los argumentos. Utilice **eval** cuando quieras que los resultados de la expansión sean aplicados a etapas anteriores a la expansión, o cuando quieras que una etapa del procesamiento se repita de nuevo. **ksh** expande los argumentos de **eval** con las reglas de procesamiento normal. **ksh** entonces forma una cadena concatenando los argumentos los argumentos de **eval**, separando cada uno por un Espacio. **ksh** lee y procesa la cadena resultante como un comando.

Utilice **eval** para:

- Procesar el resultado de una expansión o substitución por un paso que le preceda durante el procesamiento de comando
- Encontrar el valor de un parámetros cuyo nombre es el valor de otro parámetro
- Ejecutar una línea que leas

El valor devuelto por **eval** es el valor de retorno del comando formado por sus argumentos.

Ejemplos:

```
eval last='$'{$#}          # Último parámetro posicional
eval print '$'$nombre      # Visualiza la variable llamada $nombre

cmd='date | wc'
eval $cmd

read –r linea
eval "$linea"              # Procesa linea como un comando

if eval [ ! –d \${$#} ]
then print –r – "${0}: El último argumento no es un directorio."
     exit 1
fi
```

PROCESAMIENTO DE ARGUMENTOS

Es una buena práctica seguir las convenciones de argumentos de comando descritas en la página xxx cuando se diseñan programas de shell.

Todos los argumentos, tanto los que son opciones como los no lo son, son almacenados como parámetros posicionales cuando comienza el script. Se procesan las opciones leyendo cada uno de los argumentos opciones y creando variables para cada una de las opciones especificadas. Desplaza los parámetros posicionales de forma que después de que se haya completado el procesamiento de opciones, ellos contengan solamente argumentos que no sean opciones.

El comando **getopts** facilita el procesamiento de argumentos de la línea de comandos de una forma que sigue las convenciones de la línea de comando estándar descritas en la página xxxx. Con **getopts** especificas que opciones están permitidas, y qué opciones están permitidas que tomen argumentos. Las opciones que requieren argumentos son seguidas con unos **:** (dos puntos). Encierra información entre **[…]** seguida por **:** o un **Espacio** para tenerla incluida en los mensajes de uso generados automáticamente. Unos **:** (dos puntos) al principio de la cadena de opción produce que se devuelvan errores en la variable opción en vez de ser visualizada automáticamente por **ksh**. En este caso, una opción que requiere un argumento es devuelta como unos **:** (dos puntos), y las opciones desconocidas son devueltas como un **?**. También especificas el nombre de una variable que se utiliza para almacenar la opción que **getopts** encuentra. **getopts** procesa la lista de argumentos, una opción cada vez. Para que procese todos los argumentos, **getopts** necesita ser llamada desde dentro de un comando compuesto. Normalmente se utiliza un comando **while** para procesar el conjunto de opciones completo, y se utiliza un comando **case** dentro del proceso **while** para procesar cada opción.

Las variables **OPTARG** y **OPTIND** son fijadas por **getopts**. Después de procesar una opción que toma un argumento, **OPTARG** contendrá el valor del argumento opción. La variable **OPTIND** contendrá el índice del parámetro posicional que será procesado por la siguiente llamada a **getopts**. Normalmente correrás los parámetros posicionales mediante **OPTIND-1** después de procesar todas las opciones, de forma que los parámetros posicionales contengan solamente los argumentos que no son opciones.

El valor de retorno para **getopts** es Verdadero mientras existan más opciones para ser procesadas, de forma que todos los errores pueden ser fácilmente detectados y reportados. Un — en la lista de opciones significa el final de las opciones.

Versión: En versiones de **ksh** posteriores a la de 16 de Noviembre de 1988, un mensaje de Uso se generará con la opción **-?**.

Ejemplo:

```
prog=${0##*/}       # utiliza solamente el nombre de fichero, no el pathname
while getopts :abo: c
do  case $c in
    a)  aflag=1;;
    b)  bflag=1;;
    o)  oflag=$OPTARG;;
    :)  print –u2 “$prog: $OPTARG requiere un valor”
        exit 2;;
   \?)  print –u2 “$prog: opción desconocida $OPTARG”
        print –u2 “Uso: $prog [-a –b –o valor] fichero ...”
        exit 2;;
    esac
done
shift $((OPTIND-1))
```

COMANDOS BUILT-IN (EMPOTRADOS)

Los comandos built-in son similares a programas y scripts, pero son procesados por la shell en lugar de por un proceso separado. Esto hace posible que los comandos built-in hagan cosas que no pueden ser hechas como programas separados. Por ejemplo, el comando **cd** cambia el directorio

para el proceso actual. Si **cd** fuese escrito como un programa separado, entonces el directorio actual volvería a ser la localización original cuando **cd** se completase.

Los comandos built-in se ponen en marcha mucho más rápidamente de lo que lo hacen los programas separados. Los comandos que son built-in solamente para mejorar el desempeño pueden ser asociados con un pathname absoluto. Estos built-in se ejecutarán solamente después de hacer una búsqueda de path y encontrar el programa ejecutable de ese nombre. De otra forma, los built-ins son encontrados antes de buscar los programas.

El comando **builtin** sin argumentos lista el conjunto actual de built-ins, incluyendo el pathname de aquellos built-ins que tienen un pathname asociado. Es posible cambiar el pathname de un built-in o eliminar un built-in con **builtin**. Si el código para un comando built-in está ya cargado en la shell, entonces puedes hacerlo un built-in dándole el nombre del comando como un argumento a **builtin**. Puedes eliminar un comando built-in con la opción **–d** de **builtin**.

Además, en sistemas que tienen la habilidad de linkar código en tiempo de ejecución, el comando **builtin** puede ser utilizado para añadir librerías de código y comandos built-in. La descripción de cómo crear estos comandos está fuera del propósito de este manual.

Versión: El comando **builtin** está disponible solamente en versiones de **ksh** posteriores al 16 de Noviembre de 1988.

Ejemplo:

```
type wc test
wc is a tracked alias for /bin/wc
test is a shell builtin
builtin wc
builtin –d test
type wc test
wc is a shell builtin
test is a tracked alias for /bin/test
```

SCRIPTS DOT (PUNTO)

ksh ejecuta un script en un entorno separado tanto si llamas al script por su nombre como ejecutando **ksh** con el nombre del script como argumento. En cualquier caso, los cambios realizados al entorno del script, tales como cambiar el valor de cualquier variable, no afectará al entorno del programa que realiza la llamada.

Utilice el comando **.** (punto) para hacer que el script se ejecute en el entorno actual. Los argumentos al comando **.** (punto) son el nombre del script y sus argumentos. Si se le dan argumentos, ellos remplazan los parámetros posicionales dentro del script. **Versión**: Con la versión de 16 de Noviembre de 1988 de **ksh**, si se especificaron argumentos al script, los parámetros posicionales no fueron restaurados cuando finalizó el script.

El primer argumento para el comando **.** (punto) puede ser una función definida con la sintaxis **function *nombre*** descrita en la página xxxx. En este caso la función se ejecuta en el entorno actual.

Un script invocado por el comando **.** (punto) es llamado script punto. Un script punto es normalmente utilizado para inicializar el entorno de un programa. Una aplicación puede leer un script punto para crear un entorno por defecto antes de procesar los argumentos del comando.

Utilice **return** para finalizar el script punto. Utilice **exit** para finalizar el shell actual o shell script.

Ejemplos:

```
cat foo.init
# Este fichero fija valores por defecto
Logfile=$HOME/foo.log
Tmpdir=/usr/tmp
Prompt='dime! '
. foo.init
# El siguiente ejemplo no funciona en la versión de 16/Nov/1988 de ksh debido a
# la ausencia de entrecomillado de la salida de set
set > /tmp/save$$   # Salva las variables a un fichero temporal
. /tmp/save$$       # Restaura las variables de un fichero temporal
```

DEFINIENDO Y UTILIZANDO FUNCIONES

Las funciones proporcionan un medio eficiente para escribir comandos relativamente simples que se ejecutan en el entorno actual. También proporcionan una forma de dividir un script en trozos más pequeños y bien definidos que son más fáciles de escribir. Una función se diferencia de un script en que la función se ejecuta en el entorno actual y, por lo tanto, puede compartir variables y otros efectos laterales con el script que la invoca. Una función se ejecuta más rápidamente que un script punto debido a que **ksh** lee una función justamente en el momento en que es definida, en lugar de leerla cada vez que es referenciada.

Debido a las deficiencias del estándar de shell POSIX, existen dos tipos de funciones en **ksh** que difieren ligeramente en sintaxis y semántica. Las funciones POSIX son definidas con la sintaxis ***nombre(){cuerpo;}***, y no proporcionan ningún ámbito de variables o captura (traps). Esto las hace funciones difíciles de escribir que no tienen efectos laterales.

La otra forma de definir funciones en **ksh** es con el comando compuesto **function** el cuál utiliza la sintaxis ***function nombre {cuerpo;}***. Utilice **typeset**, con o sin opciones, dentro de la definición de función para declarar variables locales. Los nombres de las variables locales deben ser identificadores. Cambiar el valor de una variable local no tiene efecto sobre una variable del mismo nombre en el script que llama a la función. Si no declaras una variable para que sea local, entonces cualquier cambio que realices a la variable se mantendrá después de que la función vuelva al script que ha hecho la llamada.

Utilice **return** para volver al script que ha hecho la llamada. Si no utiliza **return**, la función finaliza después de que **ksh** procesa el último comando dentro de la función. No utilice **exit** a menos que desees realmente finalizar el script actual.

Llame a una función por su nombre, del mismo modo que invocas a un comando built-in, un script o programa. Una función debe estar definida, o encontrarse en un directorio nombrado por la variable **FPATH** antes de ser la referenciada. Especifique el nombre de función seguido por sus argumentos, si los tiene. Estos argumentos se convierten en los parámetro posicionales dentro de la función, pero no cambia los parámetros posicionales actuales del script que ha llamado a la función. El valor del parámetro posicional **0** se fija al nombre de la función, solamente cuando la función ha sido definida con el comando compuesto **function**. Cuando una función POSIX es invocada, **$0** no cambia.

Una función definida con el comando compuesto **function** puede ser dada como argumento al comando **.** (punto), el lugar del nombre de un fichero. En este caso la función se ejecuta sin un ámbito separado, y sin reasignar el parámetro **0**, como si hubiese sido definida con la sintaxis POSIX.
Versión: Esta característica está solamente disponible con versiones de **ksh** posteriores a la de 16/Nov/1988.

Después de expandir todas las palabras de un comando, **ksh** chequea primeramente el nombre de comando utilizando el orden de búsqueda definido en la página xxxx. Por lo tanto, no tiene significado definir una función cuyo nombre es el mismo que el de un comando built-in especial. El comando built-in **command** puede ser utilizado dentro de una función para invocar un comando built-in o un programa que tiene el mismo nombre que la función, en lugar de invocarse la función de forma recursiva.

Cuando escribes funciones, intenta evitar producir efectos laterales con cada una de las siguientes:

- Variables. Altera las variables globales solamente cuando sea esencial hacerlo. No cambies sus valores o sus atributos
- Funciones. Crea o borra solamente cuando sea esencial hacerlo así.
- Alias. Crea o borra solamente cuando sea esencial hacerlo así.
- Directorio de trabajo. Como las funciones se ejecutan como parte del mismo proceso, es posible cambiar el directorio de trabajo. El script que llama a la función no se debería encontrar de forma inesperada en otro directorio.
- Ficheros temporales. Antes de devolver el control al script que lo llamó, un función debería eliminar cualquier fichero temporal que la función hubiese creado para su uso propio. Ten especial cuidado cuando crees pathnames temporales formados utilizando el parámetro **$**, el cual se evalúa al identificador de proceso actual, porque tanto la función como el script que la llamó se ejecutan en el mismo proceso. Utilice el nombre de función, o una abreviatura, como parte del pathname.

Los nombres de función y definiciones no son heredados por los scripts o a través de invocaciones de **ksh** separadas. Deberías poner las definiciones de funciones en un fichero cuyo nombres es el nombre de la función, en un directorio definido en la variable **FPATH**. Adicionalmente, puedes poner definiciones de funciones en tu fichero de entorno si quieres que estén definidas cuando **ksh** se ejecute de forma interactiva.

Cuando las funciones son leídas por **ksh**, son copiadas al fichero histórico a menos que la opción **nolog** esté activa. Utilice el alias prefijado, **functions**, para visualizar la definición de una o más funciones. Para eliminar una función, utilice **unset –f** seguido del nombre de la función.
Versión: Con la versión de **ksh** de 16/Nov/1988, las funciones definidas con cualquiera de las dos sintaxis se comportaban como las funciones definidas con el comando **function**. Además, un comando built-in era encontrado antes que una función del mismo nombre; por lo que era necesario usar una combinación de alias y funciones para escribir una función que sustituyese a un comando built-in. Además, el built-in **command** no estaba disponible.

Ejemplo:

```
# isnum devuelve Cierto si su argumento es un número válido
# El primer patrón requiere un dígito antes del punto decimal, y el segundo
# después del punto decimal
function isnum # parámetro_cadena
{
  case $1 in
  ?([-+])+([0-9])?(.)*([0-9])?([Ee]?([-+])+([0-9])) )
    return 0;;
  ?([-+])*([0-9])?(.)+([0-9])?([Ee]?([-+])+([0-9])) )
    return 0;;
  *) return 1;;
  esac
  # Esta línea no se ejecuta
}
```

FUNCIONES AUTO-LOAD (AUTO-CARGA)

Una función que queda definida la primera vez que es referenciada es llamada una función auto-carga. Su ventaja principal es un mejor desempeño, ya que **ksh** no tiene que leer la definición de función si no referencias a la función nunca.

Puedes especificar que una función se auto-cargue con el alias prefijado **autoload**. Cuando **ksh** encuentra primero una función auto-carga, utiliza la variable **FPATH** para buscar un nombre de fichero cuyo nombre encaje con el de la función. Este fichero debe contener la definición de esta función. **ksh** lee y ejecuta este fichero en el entorno actual y entonces ejecuta la función.

Versión: Con versiones de **ksh** posteriores al 16/Nov/1988, las funciones son también auto-cargadas cuando se encuentra un comando en un directorio que está listado tanto en la variable **PATH** como en la variable **FPATH**.

Ejemplo:

```
FPATH=$HOME/funlib
                    # Directorios en los que buscar funciones
autoload foobar     # Especifica que foobar sea auto-cargado
foobar *.c          # Ejecuta foobar con los ficheros que acaban en .c.
```

Nota: Puedes cargar una librería de funciones relacionadas con la primera referencia a algunos de sus miembros, poniéndolos a todos en un único fichero. Para cada función definida, puedes utilizar **ln** para crear un nombre de fichero con el nombre de la función que se refiere a este fichero.

FUNCIONES DISCIPLINE (DISCIPLINA)

Versión: Las características descritas en esta sección están disponibles solamente en versiones de **ksh** posteriores al 16/Nov/1988.

Además de las funciones cuyos nombres son identificadores, cada variable puede tener una o más funciones asociadas con ella. Estas funciones tienen nombres de la forma *nombre_variable.acción*, donde *nombre_variable* es el nombre de una variable y *acción* es un identificador que puede ser aplicado a esa variable. Estas funciones son llamadas funciones disciplina para la variable dada. Cada variable puede definir una función disciplina para cada una de las siguientes acciones:

get Llamada cuando se referencia a la variable. Si se le da un valor a la variable **.sh.value** dentro de la función disciplina, entonces éste será el valor de la variable que fue

referenciada.

set Llamada cuando se le asigna un valor a la variable. La variable **.sh.value** se fija al valor que sería asignado. El valor después de que retorne la función disciplina será usado en la asignación. Si **.sh.value** es unset dentro de la función disciplina, entonces no se realizará ninguna asignación.

unset Llamada cuando la variable es desasignada (unset). La variable no será desasignada a menos que se desasigne dentro de la función disciplina.

Cuando quiera que se invoca a una función disciplina, la variable **.sh.name** se fija al nombre de la variable, y **.sh.subscript** es fijado al valor del subscript.

Estas funciones disciplina pueden ser utilizadas para hacer activas las variables en lugar de celdas de almacenamiento pasivo. Por ejemplo, si defines una variable **foo** para que sea un array asociativo, y defines una disciplina **get** para **foo**, entonces **${foo**[subíndice]} se comporta como una función cuyo argumento es subíndice.

Disciplinas adicionales pueden ser creadas para variables creadas por los comandos built-in que son añadidas a la shell.

Ejemplos:

```
function date.get
{
        .sh.value=$(date)
}
print –r –- “$date”
Sat Aug 28 12:48:25 2001
typeset –A linecount
function linecount.get
{
        .sh.value=$(wc –l < ${.sh.subscript} 2>/dev/null)
}
print –r -- $((linecount[fichero1] + linecount[fichero2]))
19
```

REFERENCIAS DE NOMBRES

Versión: Esta característica descrita en esta sección esta disponible solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

A menudo es deseable pasar el nombre de una variable en lugar de su valor, como argumento de una función. Una variable de referencia proporciona un mecanismo para proporcionar un nombre local para esta variable. Una variable de referencia se crea con **typeset –n**, o con el alias predefinido **nameref**. El nombre de una variable de referencia debe ser un identificador. El valor de la variable en el momento en que se especifica el atributo **typeset –n**, define la variable que será referenciada por la variable de referencia. Los siguientes usos de la variable de referencia (excepto **typeset +n**) hacen que la operación ocurra sobre la variable que está siendo referenciada por la variable de referencia. Por ejemplo, si se le asigna el valor **bar** a la variable **foo**, y a **foo** se le da el atributo de referencia de nombre, **-n**, entonces cada asignación a **foo** será una asignación a **bar**.

Ejemplo:

```
function reverse          # nombre de array
{
```

```
        nameref foo=$1
        typeset temp
        integer i n=${#foo[@]}   # número de elementos del array
        for  ((i=0; i < n/2; i++))
        do    temp=${foo[i]}
              foo[i]=${foo[n-1-i]}
              foo[n-1-i]=$temp
        done
}
```

VARIABLES COMPUESTAS

Versión: Las características descritas en esta sección están disponibles solamente en las versiones de **ksh** posteriores a 16/Nov/1988.

Una variable compuesta es una variable cuyo nombre consta de más de un identificador. Una asignación compuesta es una asignación de la forma ***nombre=(palabra ...)*** la cual es usada para asignación de arrays como se describió anteriormente, o ***nombre=(asignación ...)***, donde ***asignación*** puede ser una asignación simple o una asignación compuesta. Para cada ***asignación*** de la lista, es asignada la variable ***nombre.asignación***. Además, el valor de ***nombre***, ***$nombre***, consta de una lista de asignaciones encerradas entre paréntesis, que es equivalente a una asignación para todas las variables compuestas cuyo nombre comienza con ***nombre***.

Una vez que se ha creado la variable compuesta ***nombre***, puedes operar en cada miembro de forma independiente. Variables de la forma ***nombre.*** creadas o eliminadas, serán creadas o eliminadas de la variable compuesta ***nombre.*** A los miembros individuales de una variable compuesta no se les puede dar el atributo export, debido a que el atributo export no puede ser dado a ninguna variable cuyo nombre contenga un **.** (punto).

Ejemplos:

```
picture=(
              imagen=$PICTDIR/fruta
              color=([manzana]=roja [mora]=violeta [platano]=amarillo)
              tamanio=( typeset –E x=8.5  y=11.0 )
)
print –r –- “$picture”
(
        tamanio=(
              typeset –E x=8.5
              typeset –E y=11
        )
        imagen=/usr/local/pictures/fruta
        color=(
              [mora]=violeta
              [manzana]=roja
              [platano]=amarillo
        )
)
print –r –- “${picture.tamanio.x}”
8.5
unset picture.color
typeset –F2 picture.coste=22.50
print –r –- “${picture}”
(
        typeset –F 2 coste=22.50
        tamanio=(
              typeset –E x=8.5
              typeset –E y=11
        )
        imagen=/usr/local/pictures/fruta
```

)

El nombre de una variable compuesta es frecuentemente pasado como una referencia de nombre a una función. La función puede entonces utilizar, modificar, añadir o eliminar los miembros individuales.

Ejemplo:

```
function rotate     # ángulo de punto
{
             nameref point=$1
             float temp c=$((cos($2))) s=$((sin($2)))
             (( temp = c*point.x – s*point.y ))
             (( point.y = s*point.x + c*point.y ))
             (( point.x=temp ))
}
p=(x=1.0 y=0.0)
rotate p $(( 2*atan(1.0) ))
print p=$p
p=( y=1 x=0.000000000005 )
```

FIJANDO TRAPS PARA ATRAPAR LAS INTERRUPCIONES

Utilice **trap** para especificar una acción a ser ejecutada cuando alguna de un conjunto especificado de condiciones asíncronas se produce o levanta. Una condición se produce o levanta cuando tu script recibe una señal.

La acción que especificas se expande cuando **trap** se procesa, y de nuevo justamente antes de que **ksh** ejecute la acción. Por lo tanto, utilice comillas literales (simples) en lugar de comillas de agrupación (dobles) para evitar la expansión cuando especifiques la acción.

Un trap sobre **EXIT** se ejecuta cuando el script o función **ksh** finaliza su ejecución. Utilice este trap para especificar acciones de limpieza tales como eliminar ficheros temporales.

Si especificas un trap sobre **EXIT** dentro de una función que no es POSIX, entonces el trap se ejecuta justamente después de que la función finaliza. Un trap sobre **DEBUG** se ejecuta después de que cada comando es expandido y antes de ejecutarse. Utilice este trap como ayuda a la depuración de funciones y/o scripts.

Versión: Un trap sobre **DEBUG** era ejecutado después de la ejecución de cada comando con la versión de **ksh** de 16/Nov/1988.

Ejemplos:

```
trap ‘rm –f /tmp/fichero$$’ EXIT  # Elimina los ficheros temporales antes de salir
trap ‘mail morris <<\!
El programa abortó
!
‘  HUP TERM
```

DEBUGGING (DEPURACIÓN)

Puedes chequear la sintaxis de un script sin ejecutarlo realmente, invocando a **ksh** con la opción **noexec** o **–n**. Recomendamos especialmente que ejecutes cada script que escribas con esta opción. Puedes detectar a menudo errores que no encontrarías cuando ejecutases el script, porque el script podría no pasar por ciertas partes cuando lo ejecutases sin la opción **noexec**. Además, se producirán advertencias sobre características obsoletas y probablemente errores.

Puedes hacer que **ksh** visualice su entrada según la va leyendo con la opción **verbose**.

Un método simple para depurar un script es hacer que **ksh** tenga que visualizar una traza de ejecución. Una traza de ejecución es un listado de cada comando después de que **ksh** lo haya expandido, pero antes de que haya ejecutado el comando. Para hacer que **ksh** visualice una traza de ejecución de cada comando en un/a:

- Script: Llame a **ksh** con la opción **xtrace**, o active la opción **xtrace** con **set –x** (o **set –o xtrace**) dentro del script.
- Función. Si la función está definida con el tipo de sintaxis **function** *nombre*, especifique **typeset –ft** y el nombre de la función que deseas rastrear. De otra forma, la función comparte la opción **xtrace** con el entorno que produce la llamada.

En prompt **PS4** es evaluado para expansión de parámetro, expansión aritmética y substitución de comando y es visualizado delante de cada línea de una traza de ejecución. Utilice la variable de entorno **LINENO** dentro del prompt **PS4** para identificar el número de línea en el script o función que se corresponde con la línea que **ksh** visualiza. Por ejemplo, **PS4='[$LINENO]+'** visualizará el número de línea dentro de **[...]+** antes de cada comando de la traza.

Cuando **ksh** detecta un error de sintaxis mientras lee un script, visualiza el nombre del script, el número de línea que estaba leyendo dentro del script cuando encontró el error, y la causa de error. El número de línea que visualiza **ksh** para una comilla sin correspondencia no es a menudo la línea que causó el error, ya que las cadenas entrecomilladas se pueden extender a lo largo de varias líneas. Vuelve hacia atrás en el script hasta que encuentres la comilla omitida o la comilla extra que insertaste. También, **ksh –n** podría ayudarte a encontrar las comillas olvidadas.

Ejemplos:

```
# línea 1 – Este es el fichero bar.
function badquote
{
       print "Esta es la línea 1  # Aquí se encuentra el fallo
       for i in *
        do   foo
        done
        print "Última línea"
}
bar: syntax error al line 8 : `"' unmatched
ksh –n bar
bar: warning: line 4 : " quote may be missing
bar: syntax error at line 8: `"' unmatched
```

Cuando **ksh** detecta un error mientras ejecuta una función o script, visualiza un mensaje de error precedido por el nombre de función o nombre de script, y el número de línea del error encerrada entre **[ ]**. Los número de línea son relativos al principio del fichero para un script, y relativo a la primera línea de la función cuando el error se produjo dentro de la función.

Algunas veces es útil comentar porciones del código para localizar un error. Inserta un # al principio de cada línea que desees comentar.

Puedes insertar una llamada a una función, como la función definida en el siguiente ejemplo, en cualquier punto del script o función que te permita examinar de forma interactiva el entorno en ese punto.

Ejemplo:

```
function breakpoint
```

```
{
        while read -r line?"">> "
        do    eval "$line"
        done
}
```

Puedes utilizar el trap **DEBUG** para especificar que se ejecute un comando después de que **ksh** ejecute cada comando. Un depurador de punto de ruptura completo ha sido escrito para **ksh** por Bill Rosenblatt y está disponible a través de ftp anónimo en Internet en **ftp.uu.net** como el fichero /published/oreilly/nutshell/ksh/ksh.tar.Z.

INTERNACIONALIZACIÓN

Versión: Las características descritas en esta sección están disponibles solamente en las versiones de **ksh** posteriores a 16/Nov/1988.

Es posible escribir scripts cuyo comportamiento dependa del país o locale en el cual se ejecute. Locales diferentes podrían utilizar conjuntos de caracteres diferentes, clasificar los tipos de caracteres de forma diferente, comparar los caracteres de forma diferente, y utilizar un carácter diferente para representar los puntos decimales para los números. Además, los mensajes de cadenas y de error podrían hacer falta ser visualizados en el lenguaje definido en el locale actual.

ksh utiliza las variables **LANG, LC_ALL, LC_COLLATE, LC_TYPE** y **LC_NUMERIC** para proporcionar soporte a aplicaciones internacionalizadas. El valor de estas variables es el nombre del locale que será usado. Si **LC_ALL** está fijado, su valor se superpone al valor de todas las otras variables. De otro modo, cada una de las variables **LC_** se superpone al valor de la variable **LANG**.

Los mensajes de cadena y de error son traducidos al locale actual buscando el mensaje o cadena en un diccionario que está definido para el locale actual. Algunos mensajes de error son generados por el propio **ksh**. Puedes poner un **$** delante de cada cadena entrecomillada con comillas dobles para denotar cadenas que necesitan ser traducidas a una cadena en el locale actual. Puedes invocar a **ksh** con la opción **–D** y el nombre del script para generar la lista de cadenas que necesitan traducción. Puedes entonces hacer un diccionario de traducción para estas cadenas. El procedimiento para hacer diccionarios de lenguaje va más allá del ámbito de este manual.

6. PERSONALIZANDO TU ENTORNO

FICHERO HISTÓRICO

ksh utiliza un fichero para almacenar los comandos que introdujiste desde el terminal. Este fichero se conoce como el fichero histórico (history). Tanto el editor built-in emacs como vi tienen directivas de edición que te permiten recuperar comandos del fichero histórico para editarlas y reintroducirlas. Utilice **history** para visualizar los comandos de tu fichero histórico.

Si no especificas la opción **nolog**, entonces **ksh** también utiliza el fichero histórico para almacenar definiciones de funciones.

ksh abre el fichero histórico tan pronto como encuentra una definición de función y la opción **nolog** no está activada, o después de acabar de leer el fichero de entorno, lo que quiera que pase primero. El nombre del fichero histórico será el valor de la variable **HISTFILE** en el momento en que **ksh** lo abre. Si el fichero histórico no existe, o si **ksh** no puede abrirlo para leerlo y añadirle, entonces **ksh** crea un nuevo fichero histórico en un directorio temporal y fija sus permisos de forma que pueda leer de él y escribir en él.

Precaución: Cambiar **HISTFILE** después de que el fichero histórico haya sido abierto no afectará al **ksh** actual. Afectará solamente a invocaciones de **ksh** posteriores.

ksh siempre añade los comandos al fichero histórico. Puedes también utilizar **read –s** y **print –s** para añadir comandos al fichero histórico. Por ejemplo, para cambiar el valor de la variable **PATH**, ejecute **print –s "PATH=$PATH"** y después utilice uno de los editores built-in para editar el comando previo. **Precaución**: Es posible que la longitud de este comando sea más larga que el límite de tamaño de línea impuesto por los editores en línea. Con el modo de edición **vi** puedes utilizar la directiva **v** para invocar al comando **vi** sobre esta línea. De otra forma, puedes utilizar el comando **hist** para editar el comando.

ksh no impone límite al tamaño del fichero histórico. Sin embargo, el valor de la variable **HISTSIZE** en el momento que el fichero histórico es abierto especifica un límite para el número de comandos previos a los que **ksh** puede acceder.

ksh no elimina el fichero histórico al final de la sesión de conexión al sistema. Cuando se invoca **ksh** y determina que ninguna otra instancia de **ksh** tiene el fichero histórico abierto, y el fichero histórico no ha sido modificado en los últimos 15 minutos o más, **ksh** elimina los comandos de tu fichero histórico más viejos que los últimos **HISTSIZE** comandos.

Precaución: En algunos sistemas, **ksh** no elimina comandos antiguos del fichero histórico cuando te conectas al sistema, y el fichero histórico podría continuar creciendo. Esto hace que **ksh** utilice más tiempo para comenzar la ejecución y utilice espacio del sistema de ficheros. En algunos sistemas puedes evitar este problema especificando **ksh** como su shell de conexión al sistema. En otros sistemas tienes que eliminar tu fichero histórico de forma periódica.

Invocaciones separadas de **ksh** comparten el fichero histórico entre todas las instancias que especifican el mismo nombre y tienen el permiso apropiado. Los comandos tecleados desde una

invocación están accesibles por todas las instancias interactivas de **ksh** que se están ejecutando de forma concurrente y compartiendo el mismo fichero histórico. La variable **HISTSIZE** en cada instancia de **ksh** determina el número de comandos accesibles en cada instancia. En un terminal con múltiples ventanas, puedes teclear un comando en una ventana y acceder a dicho comando en otra ventana.

ENTORNO DE LOGIN (PROFILE)

Cuando utilizas **ksh** como tu shell de conexión al sistema, **ksh** ejecuta el script **/etc/profile** si existe, y después ejecuta el fichero con el pathname resultante de la expansión de parámetro en **${HOME: –.}/.profile**. Puedes usar este fichero para:

- Fijar y exportar valores de variables que desees tener fijadas para todos los programas que ejecutes, tales como la variable **TERM** para especificar el tipo de terminal que estás usando
- Fijar opciones como **ignoreeof** que deseas que se apliquen solamente a tu shell de conexión al sistema
- Especificar un script para que sea ejecutado cuando te salgas del sistema. Utilice **trap** sobre la condición **EXIT**.

Precaución: Por razones de seguridad, tu fichero profile no se procesa cuando la opción **privileged** está activada.

El capítulo “Programas y funciones de Shell” contiene un fichero **.profile** de ejemplo válido para la mayoría de los sistemas UNÍX.

FICHERO DE ENTORNO

Cuando **ksh** comienza una ejecución interactiva, expande la variable **ENV**, y ejecuta un script por este nombre si existe. Este fichero se llama fichero de entorno. Utiliza este fichero para:

- Definir aliases y funciones que se aplican solamente para uso interactivo
- Fijar opciones por defecto que quieres que se apliquen a todas las invocaciones de **ksh** interactivas
- Fijar las variables que quieres que se apliquen a la invocación de **ksh** actual.

Versión: El fichero definido por la variable **ENV** era también ejecutado para las invocaciones del shell no interactivas con la versión de **ksh** de 16 de Noviembre de 1988.

Precaución: Por razones de seguridad, tu fichero de entorno no se procesa cuando la opción **privileged** está activa.

El capítulo “Programas y funciones de Shell” contiene un fichero de entorno de ejemplo adecuado para la mayoría de los sistemas UNIX.

PERSONALIZANDO TU PROMPT

ksh visualiza el prompt primario cuando la opción **interactive** está activa y **ksh** está preparada para leer un comando.

La variable **PS1** determina tu prompt primario. **ksh** realiza la expansión de parámetros, expansión aritmética, y la substitución de comandos sobre el valor de **PS1** cada vez antes de

visualizar el prompt. Después de expandirse **PS1**, el carácter **!** es remplazado por el número de comando actual. Utilice **!!** si quieres que el prompt contenga el carácter **!**.

Utilice la expansión de parámetro **'${PWD#$HOME/}'** o **'${PWD/$HOME/\~/}'** dentro de tu prompt para visualizar tu directorio de trabajo relativo a tu directorio home.

Utilice la variable **SECONDS** para poner el momento del día en tu prompt. Utilice substituciones aritméticas para generar la hora y minuto del día de la variable **SECONDS**. Pon las siguientes líneas en tu fichero profile:

```
# Fija SECONDS al número de segundos desde la media noche
export SECONDS="$(date '+3600*%H+60*%M+%S')"
# La siguiente variable almacena horas y minutos
typeset -Z2 _h _m   # Dos columnas con ceros iniciales
# La siguiente expresión reformatea SECONDS
time='$((h=SECONDS/3600)%24)):$((_m=(SECONDS/60)%60))'
PS1="($_time)"'!$ '
# Advierta que $_time se remplaza por la expresión de encima
# La expresión obtiene su valor cuando PS1 se visualiza
```

CAMBIANDO DIRECTORIOS

ksh siempre recuerda tu directorio de trabajo anterior. Utilice **cd~** para volver al directorio de trabajo anterior.

Puedes fijar la variable **CDPATH** a una lista de prefijos de directorios que **ksh** buscará cuando teclees un pathname que no comience con un **/**. **ksh** visualiza el pathname del nuevo directorio de trabajo cuando el nuevo directorio no es un subdirectorio del directorio de trabajo actual.

Cuando ejecutas interactivamente, podría querer que **ksh** mantuviese un seguimiento de los directorios que visitas de forma que pudieses volver a un directorio sin tener que teclear su nombre de nuevo. Puedes escribir funciones y ponerlas en tu fichero de entorno para que hagan esto. El capítulo "Programas y funciones de la Shell" tiene el código de dos interfaces distintos para la manipulación de directorios.

MEJORANDO EL DESEMPEÑO

ksh lee el fichero histórico cuando comienzas una ejecución interactiva. Por lo tanto, deberías:

- Eliminar el fichero histórico si va teniendo un tamaño bastante grande
- Utilizar **set –o nolog** para evitar que sean guardadas las definiciones de funciones en el fichero histórico.

Para mejorar el desempeño interactivo de **ksh**:

- Minimice la cantidad la cantidad de información que pones en tu fichero de entorno
- Pon las funciones que utilices frecuentemente en un directorio llamado **FPATH**, en lugar de poner las definiciones de funciones en tu fichero de entorno
- Sigue las indicaciones del siguiente párrafo para escribir scripts eficientes, cuando escribas tu fichero de entorno

- Especifique la opción **trackall** en tu fichero de entorno. Esto reduce el tiempo que tardará en encontrar algunos comandos.
- No utilice la substitución de comandos en el valor de la variable de prompt **PS1**.
- En un sistema multi-usuario, si utilizas el editor built-in **vi**, no utilice la opción **viraw** a menos que lo necesites.
- No fije la variable **MAILCHECK** a un valor menor que su valor por defecto

Para decrementar el tiempo que **ksh** tarda en ejecutar un script o función:

- Utilice los comandos built-in cuando sea posible. Los comandos built-in normalmente se ejecutan más de una orden de magnitud más rápido que lo hacen los programas.
- Las funciones son más lentas que los comandos built-in pero son aún mucho más rápidas que los programas
- Evite la substitución de comandos completamente cuando puedes utilizar expansión de parámetros, evaluación aritmética, o encaje de patrones para obtener el mismo resultado.
- Mueve lo que no varíe en un bucle, especialmente la substitución de comandos, antes del bucle
- Utilice la construcción **$(<fichero)**, en lugar de **$(cat fichero**) o **`cat fichero`**.
- Evite el uso de **read** de la entrada estándar cuando la entrada estándar sea un pipe. En algunos sistemas, **ksh** podría tener que leer un byte cada vez para asegurarse el correcto posicionamiento de la entrada estándar
- Utilice **set –f (set –o noglob**) cuando no desees la expansión de pathname
- Fije **IFS** a Nulo si no necesitas el particionamiento de campos
- Utilice **{ }** en lugar de **( )** para agrupar comandos. **( )** podría crear un proceso para el nuevo entorno de subshell, mientras que **{ }** no crea un entorno de subshell.
- Es más rápido especificar la redirección para un bucle **for**, **while**, o **until** completo, en lugar de redireccionar los comandos dentro del bucle.
- Utilice la opción **–u** para **read** y **print** en lugar de redireccionar estos comandos
- Especifique el atributo entero o punto flotante sobre las variables numéricas. Si **foo** es una variable entera o punto flotante, es más rápido usar **foo** en lugar de **$foo** dentro de una expresión aritmética.
- No utilice el comando **exec** para evitar que **ksh** cree procesos extra. **ksh** ha sido optimizado para reducir el número de procesos que crea. Utilizando **exec** nunca ayudaría, y en algunos casos podría causar que el script se ejecutase más despacio.
- Es más rápido realizar varias asignaciones de variable, definiciones de alias, declaraciones de atributos, o evaluaciones aritméticas utilizando argumentos múltiples a un comando, que utilizando comando individuales para cada asignación, etc.

Puedes utilizar el script “line count profiler” de la página xxxx para ver cuantas veces se ejecuta cada línea. Puedes ver el tiempo de ejecución de cada comando incluyendo **SECONDS** como parte del prompt de traza de ejecución **PS4** y ejecutando el script con la opción **xtrace**

activada. Por ejemplo, **PS4='[$LINENO.$SECONDS]+'** visualiza el número de línea y el momento en que cada comando comienza su ejecución antes de cada comando. **Versión**: La granularidad de **SECONDS** es de un segundo con la versión de **ksh** de 16 de Noviembre de 1988, de forma que no te proporcionará información demasiado válida.

***PARTE III*: LENGUAJE DE PROGRAMACIÓN**

7. SINTAXIS

El análisis léxico es el proceso de particionar la entrada en unidades llamadas tokens. Este capítulo describe las reglas léxicas del lenguaje KornShell. La forma en la que se utilizan estos tokens para formar comandos se describe en el siguiente capítulo.

CARACTERES ESPECIALES

ksh procesa los siguientes caracteres de forma especial (debes utilizar uno de los mecanismos de entrecomillado si quieres que ellos se representen a ellos mismos):

| & ; < > ( ) $ ` \ " ' Espacio Tabulador Salto de línea

ksh procesa los siguientes caracteres de patrón de forma especial cuando se procesan patrones: *** ? [ ]**

ksh procesa los siguientes caracteres de forma especial cuando comienzan una nueva palabra: **# ~**

ksh procesa los siguientes caracteres de forma especial cuando se procesa la asignación de variables: **= [ ]**

NEWLINES (SALTOS DE LÍNEA)

Un newline o salto de línea es un token que es usado tanto para terminar un comando simple como para separar partes de un comando compuesto. Pulse la tecla RETURN para introducir un newline. Puedes utilizar múltiples newlines o saltos de línea donde es posible utilizar un newline.

Excepto dentro de una cadena entrecomillada con comillas simples, puedes continuar un comando en líneas siguientes, mediante la introducción de un carácter **** justo antes del newline. El **** y el newline son ambos eliminados del comando.

COMENTARIOS

Los comentarios comienzan con un signo **#** sin entrecomillar, y continúan hasta el siguiente salto de línea. Se puede usar un comentario en cualquier parte en la que se puede utilizar un token. Para un comentario multilínea, tienes que empezar cada línea con un #.

Algunos sistemas utilizan un comentario de la forma **#!pathname** en la primera línea de un shell script. El propósito de esto es definir el nombre del intérprete que procesará el script si invocas al script por su nombre, en lugar de especificar el script como un argumento de comando de **ksh**.

Ejemplos:

```
# Esto es una línea de comentario
# Esto es una segunda línea de comentario
ls –l   # Visualiza un listado en formato amplio del directorio actual.
```

OPERADORES

Los operadores son tokens. Son reconocidos en cualquier parte, a menos que los entrecomilles. Los espacios se permiten, pero no son necesarios, antes y después de un operador, excepto cuando un carácter adyacente pudiese ser concatenado para formar otro token de operador, por ejemplo, **((**. Sin espacios esto sería procesado como un único operador **((**.

Operadores de redireccionamiento de la entrada/salida: **> >> >& >| < << <<– <& <>**

Puedes preceder a cada operador de redireccionamiento de la entrada/salida con un único dígito que vaya desde 0 hasta 9, sin que intervenga ningún espacio o tabulador. El dígito se aplica al operador.

Operadores de control: **| & ; ( ) || && ;; (( )) |& ;&**

TOKENS PALABRA

Una palabra es un token que consiste en cualquier número de caracteres, separados de otras palabras por la parte izquierda y por la parte derecha por:

- Cualquier número distinto de cero de Espacios, tabuladores o saltos de línea sin entrecomillar.
- Cualquiera de los operadores listados más arriba. Un operador podría ir pegado a una palabra sin que hubiese ningún espacio, tabulador o salto de línea de por medio, pero aún así seguiría siendo un token de operador en lugar de ser parte de la palabra.

Sugerimos que indentes el cogido con Tabuladores y/o espacios para mejorar la legibilidad. Vea los ejemplos en el capítulo “Programas y funciones de shell” xxxx para una recomendación de un estilo de indentación.

ksh lee una secuencia de caracteres agrupados juntos con uno de los mecanismos de entrecomillado descritos en la página xxxx como una palabra, o parte de una palabra.

PALABRAS RESERVADAS

Las palabras reservadas son procesadas de forma especial solamente en el contexto descrito más abajo. Si utilizas cualquiera de estas palabras reservadas en cualquier otro lugar, **ksh** las procesará como palabras regulares. Las palabras reservadas son:

{ } case do done elif else esac fi for function if in select then time until while [[ ]] !.

Versión: **!** es una palabra reservada solamente en las versions de **ksh** posteriores a 16/Nov/1988.

ksh reconoce las palabras reservadas solamente cuando ellas aparecen:

- Como la primera palabra de una línea
- Después de un operador: **; | || & && |& ( )**

- Como la primera palabra después de una palabra reservada, excepto después de **case**, **for**, **in**, **select**, y **[[**.
- Como la segunda palabra después de **case**, **for**, y **select**. En este caso, **in** es la única palabra reservada permitida.

Ejemplo:

```
for i in *      # for y in son palabras reservadas
do  if foo; then bar
                # do, if y then son palabras reservadas
fi done         # fi y done son palabras reservadas
```

Adicionalmente, **ksh** no reconoce una palabra como palabra reservada:

- Cuando se usa como un patrón en un comando **case**, a excepción de **esac**.
- Cuando se usa como un patrón dentro de **( )**.
- Detrás del signo **=** dentro de una asignación de variable
- Dentro de un documento empotrado (here-document) (xxxx)
- Después de **[[** hasta el final del comando compuesto, a excepción claro está de **]]**.
- Si entrecomillas cero o más caracteres de la palabra

Ejemplo:

```
case for in         # for no es una palabra reservada
do|done)  [[ if –eq 0 ]]
                    # do, done e if no son palabras reservadas
      x=case “”do;;
                    # case y do no son palabras reservadas
esac <<!
while               # while no es una palabra reservada
!
```

NOMBRES ALIAS

Un nombre alias está formado por uno o más de cualquier carácter imprimible distinto de los caracteres especiales que se muestran en la página xxxx. **Precaución**: Ya que la substitución de alias (ver la página xxxx) se realiza después de que se procesan las palabras reservadas, un alias con el mismo nombre de una palabra reservada es normalmente ignorado.

Los aliases cuyos nombres son de la forma *_letra* definen macros para los editores empotrados **emacs** y **vi**.

IDENTIFICADORES

Use los identificadores como nombres de funciones y variables. Un identificador es una secuencia de caracteres consistente en uno o más caracteres de la clase de caracteres **alpha**, o un dígito, o un _ (subrayado). El primer carácter no puede ser un dígito. No hay límite en el número de caracteres de los que puede constar un identificador. **Nota**: En el locale POSIX, la clase de caracteres **alpha** consiste en **A–Z** y **a–z**.

Los caracteres en mayúsculas y minúsculas son distintos. Por ejemplo **ksh** procesa **A** y **a** como identificadores distintos.

Ejemplos:

```
PWD     X     x     x     Foo     Un_identificador_muy_largo
```

NOMBRES DE VARIABLES

Un nombre de variable es un nombre de variable simple o un nombre de variable compuesta. Un nombre de variable simple es un identificador.

Un nombre de variable compuesta es uno de los siguientes:

- Un identificador precedido por un **.** (punto)
- Más de un identificador cada uno de los cuales separados por un **.** (punto) y opcionalmente precedidos por un **.** (punto)

El formato de un nombre de variable es ***[.]identificador[.identificador] ...*** donde *identificador* debe presentar el formato de un identificador. Cada identificador que sigue a un **.** (punto) debe referirse a un nombre de variable definido previamente.

Los nombres de variables son también usados como parte de los nombres de las funciones disciplina, definidas en xxxx.

Versión: Los nombres de variables compuestas están solamente disponibles en las versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Ejemplos:

```
PWD   _foo   .foo   foo.bar   .sh.nombre
```

ASIGNACIÓN DE VARIABLES SIMPLES

El formato de una asignación de variable simple es

nombre_variable=valor o ***nombre_variable[subíndice]=valor***

nombre_variable debe presentar el formato descrito anteriormente. Para los arrays asociativos (ver la página xxxx), *subíndice* puede ser cualquier cadena. Si no, *subíndice* estará en el formato de expresión aritmética (ver la página xxxx). *valor* puede ser cualquier palabra.

No se permite ningún espacio o tabulador sin entrecomillar antes o después del signo **=**.

Ejemplos:

```
foo=bar   foo.bar='hello there'   .x[3]=yes   Z['hi there']=3
```

PATRONES

Los patrones de **ksh** está compuestos de los caracteres de patrón especiales especificados en la página xxx y descritos más abajo, y/o cuales quiera otros caracteres, llamados caracteres regulares. Los caracteres regulares encajan solamente con ellos mismos. Los caracteres de patrón podrían aparecer en cualquier parte en una palabra; al principio, en el medio o al final. También, pueden aparecer más de una vez en una palabra.

Entrecomille cualquiera de los caracteres especiales para quitarle su significado especial y hacer que se comporte como los caracteres regulares dentro del patrón.

Los patrones **ksh** son usados en expansión de pathname, patrones de encaje del comando **case**, patrón de encaje dentro de **[[...]]** y expansión de subcadena.

Los patrones **ksh** difieren de las expresiones regulares que son usadas en **ed**, **grep** y en otros comandos del sistema UNIX.

Caracteres de patrón

[...]

(Corchetes) Delimitan un conjunto de caracteres, ninguno de los cuales encajará con la posición de carácter identificada por los corchetes. Los siguientes caracteres son manejados de forma especial dentro de los corchetes:

- **-** (Menos) Indica un rango de caracteres. Por ejemplo, **a-z** especifica que **ksh** va a encajar con cualquiera de los caracteres comprendidos entre la **a** y la **z**. Para rangos de caracteres ASCII, vea *Conjunto de caracteres* xxxx. Si especificas los caracteres en el orden inverso al que aparece en el *Conjunto de caracteres* xxx (por ejemplo **z-a**), o si los caracteres son de dos conjuntos de caracteres distintos (por ejemplo, ASCII y Kanji), entonces **ksh** encaja solamente con el primer y el último carácter. La especificación **[a-x-z]** es equivalente a **[a-xx-z]**.
- **-** Se representa a si mismo cuando es el primer carácter después del corchete de apertura **[**, el carácter inmediatamente a continuación de un **!** que aparece justo después del **[**, o el último carácter antes de cerrar el corchete **]**.
- **[:clase:]** Indica una clase de caracteres según se define en el estándar ANSI C. Por ejemplo, con el conjunto de caracteres ASCII **[[:alpha:]]** se especifica que **ksh** encajará con cualquier carácter desde la **a** hasta la **z** o desde la **A** hasta la **Z**. **ksh** reconoce las siguientes clases de caracteres (ver *Conjunto de caracteres* xxxx) **[:alnum:] [:alpha:] [:blank:] [:cntrl:] [:digit:] [:graph:] [:lower:] [:print:] [:punct:] [:space:] [:upper:] [:xdigit:]**. **Versión**: Esta característica solamente está disponible en versiones de **ksh** posteriores a la de 16/Nov/1988.
- **[=c=]** Indica el conjunto de elementos de comparación que tienen el mismo peso primario que ***c***. **Versión**: Esta característica solamente está disponible en versiones de **ksh** posteriores a la de 16/Nov/1988.
- **[.símbolo.]** Indica el elemento de comparación definido por ***símbolo***; por ejemplo, **[.ch.]** en español. **Versión**: Esta característica solamente está disponible en versiones de **ksh** posteriores a la de 16/Nov/1988.
- **!** Inmediatamente después del corchete de apertura **[**, invierte la búsqueda o encaje. Es decir, encaja con cualquier carácter excepto los especificados. Por ejemplo, **[!a-z]** encaja con cualquier cosa que no sea una letra en minúsculas.
- **]** Se representa a si mismo cuando es el primer carácter después del corchete de apertura **[**, o el carácter inmediatamente a después de un **!** que siga al **[**.
- **** Elimina el significado especial de **–,], !**, y ****.

Ejemplos:

```
chap[259]
# Encaja con chap2, chap5, chap9
para[!1-3]
# Encaja con para4, para5, parax, etc.
```

```
chap[12][01]
# Encaja con chap10, chap11, chap20, chap21
para[1-3]
# Encaja con para1, para2, para3.
para[[:digit:]t]
# Encaja con para0, para1, ..., para9, y parat.
```

?

(Interrogación) Encaja con cualquier carácter (único). En conjunto de caracteres multibytes (e.g., Kanji), encaja con un carácter multitbyte completo, no con un único byte.

Ejemplo:

```
para?
# Encaja con todas las cadenas de 5 caracteres cuyos cuatro primeros caracteres
# son para.
```

*****

(Asterisco) Encaja con cero o más ocurrencias de alguno y de todos los caracteres

Ejemplos:

```
para*
# Encaja con todas las cadenas de caracteres que comiencen con para
x*y
# Encaja con todas las cadenas que comiencen con x y terminen en y.
```

Una **lista de patrones** son uno o más patrones separados por **|** o **&**. Un **|** entre dos patrones indica que uno de los dos patrones necesita encajar. Un **&** entre patrones indica que ambos patrones tienen que encajar. El **&** tiene precedencia sobre el **|**.

Un **subpatrón** es una lista de patrones encerrado entre paréntesis y precedido por uno de los caracteres que aparecen más abajo. Una lista de patrones puede contener uno o más subpatrones y cada subpatrón puede contener uno o más subpatrones. Un patrón puede estar formado de una de las siguientes expresiones de subpatrones:

?(lista de patrones)

Encaja con cero o una ocurrencia de *lista de patrones*.

Ejemplo:

```
para?([345]|99)1
# Encaja la cadena para1, para31, para41, para51, o para 991.
```

***(lista de patrones)**

Encaja con cero o más ocurrencias de *lista de patrones*.

Ejemplo:

```
para*([0-9])
# Encaja con la cadena para, y para seguida de cualquier número de dígitos
```

+(lista de patrones)

Encaja con una o más ocurrencias de *lista de patrones*.

Ejemplo:

```
para+([0-9])
# Encaja con para seguido de uno o más dígitos
```

@(lista de patrones)

Encaja exactamente con una ocurrencia de *lista de patrones*.

Ejemplo:

```
para@(chute|graph)
# Encaja con la cadena parachute o paragraph
```

!(lista de patrones)

Encaja con todas las cadenas excepto con las que encajan con *lista de patrones*.

Ejemplo:

```
para!(*.[0-9])
# Encaja cualquier cadena que empiece por para, y no acabe en un punto
seguido por un dígito
@(*.c&!(para*))
# Encaja con cualquier cadena que acabe en .c, y no empiece por para.
```

Los subpatrones pueden ser referenciados por número según están determinados por el orden de los paréntesis abiertos en el patrón, empezando por 1. Dentro de un patrón, puedes referirte a la cadena que encaja con un subpatrón completo previo con ***dígito***, donde *dígito* es el número del subpatrón. Esto se llama referencia hacia atrás. Son posibles hasta 9 referencias hacia atrás. Una referencia hacia atrás debe encajar con la misma cadena del subpatrón a la cual se refiere. **Versión**: Esta característica solamente está disponible en versiones de **ksh** posteriores a la de 16/Nov/1988.

Ejemplos:

```
@(?)*\1
# Encaja con cualquier cadena que comience y acabe por la misma letra
+(?)\1
# Encaja con cadenas dobles tales como fofo y barbar
```

EXPRESIONES ARITMÉTICAS

Utilice una expresión aritmética:

- Como subíndice de un array indexado
- Para cada argumento en **let**.
- Dentro de paréntesis dobles **((...))**
- Dentro de paréntesis dobles precedidos por un signo dólar. **$((...))** es remplazado por el valor de la expresión aritmética encerrada.
- Como contador de desplazamientos en **shift**.
- Como operandos de los operadores de comparaciones aritméticas de **test, [,** o **[[...]]**.
- Como límites de recursos en **ulimit**.
- Como el valor de la derecha de una asignación de variable a una variable entero o punto flotante.
- Como operandos a formateos aritméticos con **print** o **printf**.

Ejemplo:

```
print –f ‘%d\n’ 3+4
```

ksh realiza todos los cálculos usando el tipo de aritmética de punto flotante de doble precisión de tu sistema. **ksh** no realiza chequeos de desbordamiento. **Versión**: Esta característica solamente está disponible en versiones de **ksh** posteriores a la de 16/Nov/1988. Las versiones previas utilizan la aritmética entera.

Una constante entera tiene la forma ***[base#]número*** donde:

base: Un entero decimal entre 2 y 64 que define la base aritmética, La base por defecto es 10.

número: Cualquier número no negativo. Utilice el operador unario menos descrito más abajo para representar números negativos. Un número en una base mayor que 10 utiliza letra mayúsculas o minúsculas del alfabeto para representar un dígito cuyo valor es igual o mayor a 10. Por ejemplo, **16#b** o **16#B** representa 11 en base 16. Para bases mayores a 36, las mayúsculas y las minúsculas son distintas. @ y _ son los dos dígitos más grandes. Por ejemplo, **40#b** representa 11 mientras que **40#B** representa 37. Cualquier cosa después de un punto decimal es truncada.

Versión: Las bases mayores de 36 están solamente disponibles en versiones de **ksh** posteriores a la de 16/Nov/1988.

Una constante en punto flotante tiene la forma ***[±]número[.número][exponente]*** donde:

número: Cualquier número decimal no negativo

exponente: **E** o **e** opcionalmente seguidos por **+** o **–** y un número decimal no negativo.

El carácter que representa el punto decimal es una coma (**,**) en algunos locales.

Versión: Las constantes punto flotante están solamente disponibles en versiones de **ksh** posteriores al 16/Nov/1988.

Una variable es nombrada como *varname* o *nombrevariable*. Si una variable es una expresión aritmética tiene el atributo entero (ver los atributos en xxxx), entonces **ksh** utiliza el valor de la variable. De otra forma, **ksh** asume que el valor de la variable es una expresión aritmética, e intenta evaluarla. Una variable cuyo valor e Nulo se evalúa a **0**. Por ejemplo, si la variable **x** tiene el valor **y+1**, la variable **y** tiene el valor **z+2**, y **z** tiene el valor 3, entonces la expresión **2*x** se evalúa a **12**. **ksh** puede evaluar variables de hasta 9 niveles de profundidad.

Una función aritmética es denotada por ***función(expresión)***, donde *función* es una de las siguientes:

abs Valor absoluto

acos Arco coseno del ángulo en radianes

asin Arco seno

atan Arco tangente

cos Coseno

cosh Coseno hiperbólico

exp Exponencial con base *e*, donde $e \approx 2.718$

int Entero más grande menor o igual al valor de expresión

log Logaritmo

sin Seno

sinh Seno Hiperbólico

sqrt Raíz cuadrada

tan Tangente

tanh Tangente Hiperbólica

Los argumentos de cualquiera de las funciones de arriba que requieren ángulos son en radianes. **Versión**: Las funciones matemáticas que aparecen arriba solamente están disponibles en versiones de **ksh** posteriores a la de 16/Nov/1988.

Expresiones:

- Valor: El valor de una expresión con un operador de comparación o un operador lógico es **1** si no es cero (Verdadero), o **0** (falso) en caso contrario.
- Precedencia: Los elementos se listan más abajo en orden de precedencia, con los de precedencia mayor primeros. Los elementos con la mismo precedencia son listados bajo el mismo punto.
- Asociatividad: **ksh** evalúa todos los elementos de la misma precedencia de izquierda a derecha excepto para **=** y **op=** y otros operadores de asignación, los cuales son evaluados de derecha a izquierda.

Una *expresión* es una constante, una variable, o es construida con los siguientes operadores (listados aquí de mayor a menor precedencia):

- (expresión) Los paréntesis se utilizan para saltarse las reglas de precedencia.
- variable++ / ++variable Postincremento y preincremento
  variable– – / – –variable Postdecremento y predecremento
  +expresión Signo más unario
  -expresión Signo menos unario
  !expresión Negación Lógica. El valor es **0** para cualquier expresión cuyo valor no es **0**.
  ~expresión Negación a nivel de bits
- expresión * expresión Multiplicación
  expresión / expresión División
  expresión % expresión Resto de dividir la primera expresión entre la segunda expresión
- expresión + expresión Adición
  expresión – expresión Resta
- expresión << expresión Desplazamiento a la izquierda de la primera expresión en el número de bits dados por la segunda expresión
  expresión >> expresión Desplazamiento a la derecha de la primera expresión en el número de bits dados por la segunda expresión
- expresión <= expresión Menor o igual que
  expresión >= expresión Mayor o igual que
  expresión < expresión Menor que
  expresión > expresión Mayor que
- expresión == expresión Igual a
  expresión != expresión No igual a

- expresión & expresión And lógico a nivel de bits. El valor contendrá un **1** en cada bit donde haya un **1** en ambas expresiones, y un **0** el resto de posiciones de bit. Ambas expresiones son siempre evaluadas.
- expresión ^ expresión Or exclusivo a nivel de bits. El valor contendrá un **1** en cada bit en el que exactamente una de las expresiones sea **1**, y **0** en el resto de posiciones de bits.
- expresión | expresión Or a nivel de bits. El valor contendrá un **1** en cada bit en el cual hay un **1** en cualquiera de las dos expresiones o en ambas, y **0** en el resto de posiciones de bits. Las dos expresiones son evaluadas siempre.
- expresión && expresión And lógico. Si la primera expresión es cero, entonces la segunda expresión no se evalúa.
- expresión || expresión Or lógico. Si la primera expresión no es cero, entonces la segunda expresión no se evalúa.
- expresión ? expresión:expresión Operador condicional. Si la primera expresión no es cero, entonces se evalúa la segunda expresión, si no, se evalúa la tercera expresión.
- variable=expresión Asignación

  variable***op***=expresión Asignación compuesta. Este es el equivalente a **variable=variable op expresión**. *op* debe ser * / % + - << >> & ^ o |.
- expresión, expresión. Operador coma. Se evalúan ambas expresiones. El valor resultante es el valor de la segunda expresión. **Precaución**: La coma se utiliza como carácter para el punto decimal en algunos locales. Por lo tanto, para una portabilidad máxima deberías dejar un espacio antes de la coma cuando esté precedida por un número.

Versión: Los operadores unarios **+, ++, --, ?:** y **,** están solamente disponibles en versiones de **ksh** posteriores a la de 16/Nov/1988.

PRIMITIVAS DE EXPRESIÓN CONDICIONAL

Las primitivas de expresión condicional son expresiones unarias y binarias que se evalúan a Cierto o Falso. Son usadas dentro de expresiones con los comandos **test** y **[**, y dentro del comando compuesto **[[...]]**. Se necesitan espacios o tabuladores para separar los operadores de los operandos.

Una primitiva puede ser cualquiera de las siguientes expresiones de fichero unarias:

	Verdadero si:
–e fichero	el fichero existe. **Versión**: **–e** fichero está disponible solamente en versiones de **ksh** posteriores a la de 16/Nov/1988.
–a fichero	el fichero existe. **Precaución**: La opción **–a** está obsoleta y podría ser sustituida en el futuro. Debería usar **–e** en lugar de **–a**.
–r fichero	el fichero existe y se puede leer
–w fichero	el fichero existe y se puede escribir. Verdadero indica solamente que el bit de escritura está activo. El fichero no podrá ser escrito en un sistema de ficheros de solo lectura incluso aunque este chequeo devuelva Verdadero.

–x fichero	el fichero existe y es ejecutable. Verdadero indica solamente que el bit de ejecución está activo. Si el fichero es un directorio, Verdadero indica que podemos buscar en dicho directorio.
–f fichero	el fichero existe y es un fichero regular
–d fichero	el fichero existe y es un directorio
–c fichero	el fichero existe y es un fichero especial de tipo carácter
–b fichero	el fichero existe y es un fichero especial de tipo bloque
–p fichero	el fichero existe y es un pipe con nombre (fifo)
–u fichero	el fichero existe y su bit set-user-id está fijado
–g fichero	el fichero existe y su bit set-group-id está fijado
–k fichero	el fichero existe y su bit sticky está fijado
–s fichero	el fichero existe y tiene un tamaño mayor a 0 bytes
–L fichero	el fichero existe y es un enlace simbólico
–h fichero	el fichero existe y es un enlace simbólico
–O fichero	el fichero existe y su propietario es el identificador de usuario efectivo
–G fichero	el fichero existe y su grupo es el identificador de grupo efectivo
–S fichero	el fichero existe y es un fichero especial de tipo socket

ksh chequea el descriptor de fichero ***n*** cuando pathname de *fichero* es de la forma **/dev/fd/n**.

Una primitiva puede ser cualquiera de las expresiones unarias siguientes:

	Verdadero si:
–t descriptor_de_fichero	El fichero cuyo descriptor de fichero es *descriptor_de_fichero* está abierto y está asociado con un dispositivo de terminal.
–o opción	La opción está activa
–z cadena	La longitud de la cadena es 0
–n cadena	La longitud de la cadena no es 0

Una primitiva puede ser una cadena por si misma. En este caso la primaria es Verdadero si la cadena no es Nula. **Versión**: Con la versión de **ksh** de 16/Nov/1988, esta primaria está solamente disponible en **test** y **[**.

Con **test** y **[**, una primitiva puede ser cualquiera de las siguientes expresiones de cadena binarias:

	Verdadero si:
cadena1 = cadena2	Cadena1 es igual a cadena2
cadena1 != cadena2	Cadena 1 es distinta de cadena2

Con **[[...]]**, una primitiva puede ser cualquiera de las siguientes expresiones de cadena binarias:

	Verdadero si:
cadena == patrón	La cadena encaja con el patrón *patrón*. Entrecomilla en patrón para

	tratarlo como una cadena.
cadena = patrón	La cadena encaja con el patrón *patrón*. **Precaución**: = está obsoleto y podría ser remplazado en el futuro. Utilice == en lugar de =.
cadena != patrón	La cadena no encaja con el patrón *patrón*.
cadena1 < cadena2	La cadena1 viene antes que la cadena2 en el orden de comparación definido por el locale actual.
cadena1 > cadena2	La cadena1 viene después que la cadena2 en el orden de comparación definido por el locale actual.

Una primitiva puede ser cualquiera de las siguientes expresiones de fichero binarias:

	Verdadero si:
fichero1 **–nt** fichero2	El fichero fichero1 es más nuevo que el fichero fichero2 o si el fichero fichero2 no existe.
fichero1 **–ot** fichero2	El fichero fichero1 es más antiguo que el fichero fichero2 o si el fichero fichero2 no existe.
fichero1 **–ef** fichero2	Fichero1 es otro nombre para el fichero fichero2. Esto será cierto si fichero1 es un enlace duro (hard) o un enlace simbólico de fichero2.

Ejemplo:

```
[[ /dev/fd/2 –ef /dev/null ]]
# Cierto si el error estándar está redireccionado a /dev/null
```

Una primitiva puede ser cualquiera de las siguientes expresiones que comparan dos expresiones aritméticas:

	Verdadero si:
expr1 **–eq** expr2	expr1 y expr2 son iguales
expr1 **–ne** expr2	expr1 y expr2 no son iguales
expr1 **–gt** expr2	expr1 es mayor que el valor de expr2
expr1 **–ge** expr2	expr1 es mayor o igual que el valor de expr2
expr1 **–lt** expr2	expr1 es menor que el valor de expr2
expr1 **–le** expr2	expr1 es menor o igual que el valor de expr2

ENTRECOMILLADO

El entrecomillado es la forma de negar el procesamiento normal de muchos elementos. Puedes aplicar cualquiera de los mecanismos de entrecomillado para:

- Utilizar cualquiera de los caracteres especiales con sus significados literales
- Evitar que las palabras reservadas sean reconocidas como palabras reservadas. Entrecomille cero o más caracteres de la palabra reservada, por ejemplo **“for”**, **\for**, **“”for**, o **for””**.
- Evitar que los nombre de alias sean reconocidos como tales. Entrecomillar cero o más caracteres del nombre de alias.

- Evitar la expansión de parámetros y sustitución de comandos dentro del procesamiento de un documento empotrado o here-document. Entrecomillar cero o más caracteres de la palabra delimitadora de un documento empotrado.

\ Carácter Escape

(barra invertida). Cuando **** está:

- Dentro de un comentario, tiene su significado literal
- Si entrecomillar, la **** se elimina y el carácter único que viene a continuación, distinto del salto de línea, es tratado por su significado literal. Un salto de línea a continuación del **** también es eliminado junto con el ****.
- Dentro de comillas literales, tiene su significado literal
- Dentro de cadenas ANSI C (ver la página xxxx) formatea de acuerto a las convenciones de escape listadas en el comando **print** en la página xxxx.
- Dentro de comillas de agrupación (dobles), tiene su significado literal excepto cuando va seguido de un **$**, **`**, ****, o **"**.
- Dentro de la sustitución de comandos antigua, tiene su significado literal excepto cuando va seguida de un **$**, **`**, o ****.

Ejemplo:

```
print –r \#  \\\\ $'don\'t'
# \\ don't
```

\Newline Continuación de línea

(barra invertida seguida de un salto de línea) Une dos líneas juntas. Este es un caso especial de ****, en el cual el **** el significado normal del carácter Newline o salto de línea. No tiene este efecto si el **** está dentro de comillas simples, o si sigue a un signo # (comentario).

Ejemplos:

```
print esto es una continuación de \
línea
esto es una continuación de línea
# Lo siguiente funciona pero no se recomienda
wh\
ile ((x<3)); do ((x++));command; do\
ne
```

'...' Entrecomillado Literal (simple)

(par de comillas simples) Elimina el significado especial de todos los caracteres encerrados. Una comilla simple no puede aparecer dentro de comillas simples debido a que una comilla simple representa el final de la cadena. Esto es, incluso **\'** no es legal dentro de las comillas simples. Utilice **\'** o **"'"** fuera de las comillas simples para referirse al carácter **'** literal.

Ejemplo:

```
print –r '!*+\'"'"        # Concatena ' a un literal
!*+\'
```

$'...' Cadenas de ANSI C

(par de comillas simples precedidas por un signo de dólar) Elimina el significado especial de todos los caracteres encerrados a excepción de las secuencias de escape, las cuales son precedidas por el carácter escape (****) y son seguidas de caracteres con un significado especial, descritos en el comando **print** en la página xxxx.

Versión: Las cadenas ANSI C están disponibles solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Ejemplo:

```
print $'hola\n\tmundo'
# \t es un carácter tabulador
# \n es un carácter de salto de línea
hola
        mundo
```

"..." Entrecomillado de agrupación (doble)

(par de comillas dobles) Elimina el significado especial de todos los caracteres encerrados, excepto de **$**, **`**, **"** y ****.

Dentro de las comillas dobles, la barra invertida seguida de uno de los cuatro caracteres de más arriba hace que se elimine el **** y el carácter sea interpretado con su significado literal. Cuando no está precedido de una barra inversa, los cuatro caracteres se interpretan de la siguiente forma:

$ Expansión de parámetro

$(...) Nueva sustitución de comando. Todos los tokens entre el (y el) forman el comando.

` Antigua sustitución de comando

$((...)) Expansión aritmética

" Final de esta cadena

**** Como se dijo anteriormente, seguido de uno de estos cuatro caracteres, **** elimina el significado especial del siguiente carácter. Si no, se interpreta como un literal ****.

Ejemplos:

```
print –r "$PWD  \"  \$PWD  \\$PWD  \\\$PWD"
/usr/dgk   "   $PWD   \/usr/dgk   \$PWD
```

$"..." Entrecomillado de agrupación de mensaje (doble)

(par de comillas dobles precedidas de un signo dólar) Lo mismo que las comillas dobles excepto que la cadena será buscada en un diccionario de mensajes específicos al locale y remplazada si se encuentra. El **$** también es eliminado.

Versión: Esta característica está disponible solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Ejemplo:

```
LANG=french
print –r $"hello world"
bonjour monde
```

`...` Substitución de comandos antigua

(par de comillas inversas, también conocidas como acentos graves) **Bourne shell**: Esta es la sintaxis de la Bourne Shell, y es aceptada también por **ksh** pero está obsoleta y en desuso.

Sin embargo, sus reglas de entrecomillado son complejas, de forma que **ksh** ofrece una sintaxis más simple que realiza lo mismo (ver Nueva sustitución de comandos más abajo xxxx)

ksh construye el comando a ser ejecutado a partir de los caracteres que quedan después de que **ksh** hace los siguientes borrados. Un **** que es seguido por un **$**, **`** o **** es eliminado. El siguiente ejemplo ilustra la complejidad de entrecomillado con ``.

Ejemplo:

```
print –r “`print –r \$PWD “\$PWD” ‘)a\`\$\\’`”
/home/dgk /home/dgk )a`$\
```

Puedes incluir comillas inversas, **`...`**, dentro de las comillas de agrupación (dobles). Si haces eso, y quieres también incluir una o más comillas dobles dentro de las comillas inversas, entonces debes preceder a cada comilla doble incluida con una barra invertida. **ksh** elimina las barras invertidas cuando construye el comando.

$(...) Nueva sustitución de comandos

Todos los tokens (no los caracteres, como para la sustitución de comandos antigua vista más arriba), entre el (y el) correspondiente, forman el comando. Por lo que no tienes que cambiar un comando para ponerlo dentro de **$(...)** como tenías que hacer a menudo con **`...`**.

El anidamiento está permitido. Esto es, puedes incluir **$(...)** dentro de las comillas de agrupación (dobles)

Puedes usar paréntesis sin balancear dentro del comando, siempre que los entrecomilles.

Versión: Con la versión de **ksh** de 16/Nov/1988, debes poner un (delante de cada lista de patrón de cualquier comando **case** contenida dentro de **$(...)** para mantener los paréntesis balanceados.

Ejemplo:

```
print –r “$(print –r \$PWD “\$PWD” ‘)a\`\$\\’)”
$PWD $PWD )a\`\$\\
```

${...} Expansión de parámetros

La expansión de parámetros es descrita al principio de la página xxxx. Los caracteres entre las llaves son procesados como parte del mismo token.

Ejemplo:

```
foo=${foo:-<Este es el valor por defecto>}
```

((...)) Evaluación aritmética

Expresiones aritméticas. Los caracteres entre los paréntesis son procesados como si estuviesen contenidos dentro de comillas dobles, ya que esto es equivalente a **let “...”**.

Ejemplo:

```
while (( (X*=2) < 100 )); do print $X; done
```

$((...)) Expansión aritmética

Expresiones aritméticas. **$((...))** es remplazado por el valor de la expresión aritmética dentro de los paréntesis dobles.

Ejemplo:

```
print -- $((Celsius * 9./5 + 32))
# utilice –- en caso de que el resultado sea negativo
```

variable[...]= Asignación a una variable array

Subscript para la asignación de una variable array. Los caracteres dentro de los corchetes siguen las reglas de las comillas de agrupación (dobles).

Ejemplo:

```
X[2*(i+1)]=6
X[hola mundo]=hi
```

REDIRECCIÓN DE LA ENTRADA/SALIDA (I/O)

Para redireccionar la entrada y/o salida de un comando, utilice la notación de esta sección en cualquier parte en un comando simple, o siguiente un comando compuesto.

Ejemplo:

```
while read –r linea
do print –r "$linea"
done <ficherodesde >ficherohasta
```

Puedes también usar esta notación con **exec** para abrir y cerrar ficheros en el entorno actual.

Ejemplo:

```
exec 3< $infile 4>&-
```

Para los sistemas que soportan conexiones a socket, los pathnames de la forma **/dev/tcp/***hostid/portid* y **/dev/udp/***hostid/portid* son tratados de forma especial como conexiones de socket **tcp** y **udp** respectivamente. *hostid* es el identificador del host actual, y portid es el número de puerto. **Versión**: Esta característica no está disponible en todas las versiones de la versión de **ksh** de 16/Nov/1988.

Vea la tabla de la página xxxx para ver cómo se expanden las palabras. En particular, resaltar como se realiza la expansión de pathname solamente si resulta en un único pathname.

Cualquiera de los operadores de más abajo podría ser precedido por un único dígito, sin que se permita la intervención de ningún espacio o tabulador. En este caso el dígito especifica el número de descriptor de fichero, en lugar del valor por defecto 0 o 1.

El orden en el cual especificas la redirección es significativo. **ksh** evalúa cada redirección de izquierda a derecha en cuanto a la asociación de descriptor de fichero en el momento de evaluación.

Ejemplo:

```
cat 1>fname 2>&1
# Primero asocia el descriptor de fichero 1 con el fichero fname. Después asocia
# el descriptor de fichero 2 con el fichero asociado con el descriptor de fichero
# 1 (este es, fname). Si el orden de redireccionamiento fuese el inverso, el
# descriptor de fichero 2 sería asociado al terminal (asumiendo que el descriptor
# de fichero 1 hubiese sido especificado anteriormente), y después el descriptor
# de fichero 1 sería asociado con el fichero fname.
```

Nota: En los siguientes formatos, uno o más espacios o tabuladores son opcionales entre el operador y *palabra* y ningún espacio o tabulador está permitido entre el *dígito* y el operador de Entrada/salida.

Lectura <palabra n<palabra

Abre el fichero con el nombre que resulte de la expansión de *palabra*, para lectura como entrada estándar (o descriptor de fichero *n*). Cuando se usa con un comando, redirecciona la entrada (o descriptor de fichero *n*) del comando, desde el fichero expandido desde *palabra*.

Ejemplo:

```
mail abc <F2
# Envia el mensaje en el fichero F2 al usuario cuyo nombre de login es abc
```

Here-Document <<palabra n<<palabra

Crea un fichero here-document, y lo abre como la entrada estándar (o el descriptor de fichero *n*).

Cualquier entrada en la misma línea después del delimitador *palabra*, se lee normalmente. Puedes incluso tener otro comando en la misma línea. Una práctica común es poner *palabra* al final del comando al que se refiere. El here-document comienza justamente después del siguiente salto de línea incluso cuando el salto de línea es parte de un comando compuesto. Continúa hasta una línea que encaje con *palabra* carácter a carácter (debe existir únicamente la *palabra* en la línea para que encaje), o hasta el final del fichero. Si especificas más de un here-document en una línea de comando, entonces **ksh** los lee en el orden dado.

ksh no realiza ninguna expansión de parámetro, sustitución de comando o expansión de pathname sobre *palabra*. Si entrecomillas cero o más caracteres de *palabra* con un \, comilla simple o comilla doble, entonces **ksh** no realiza ninguna expansión sobre los caracteres del here-document. De otra forma, **ksh** lee y procesa el here-document simplemente como si fuese una comilla entrecomillada con comillas dobles, excepto que **ksh** no procesa las comillas dobles de forma especial.

Ejemplos:

```
cat <<!* | tr ‘[a-z]’ ‘[A-Z]’
este es un here-document
$HOME es mi directorio home
!*
ESTE ES UN HERE-DOCUMENT
/USR/DGK ES MI DIRECTORIO HOME
cat << “”EOF
este es un here-document
$HOME es mi directorio home
EOF
este es un here-document
$HOME es mi directorio home
```

Here-Document <<-palabra n<<-palabra

De la misma forma que arriba, excepto que **ksh** quita los tabuladores de principio de línea del here-document y la línea que contiene la palabra delimitadora que encaja.

Ejemplo:

```
cat <<-\!@
      este es un here-document
      $HOME es mi directorio home
      !@
este es un here-document
$HOME es mi directorio home
```

Duplicando, moviendo y cerrando la entrada <&palabra n<&palabra

Palabra se debe expandir a:

- Un dígito, en cuyo caso **ksh** hace a la entrada estándar (o al descriptor de fichero *n*) un duplicado del descriptor de fichero cuyo número viene dado por el dígito.
- Un dígito –, en cuyo caso **ksh** mueve el descriptor de fichero cuyo número está dado por el dígito a la entrada estándar (o al descriptor de fichero *n*). **Versión**: dígito – está disponible solamente en versiones de **ksh** posteriores a la de 16/Nov/1988.
- – (menos), en cuyo caso **ksh** cierra la entrada estándar (o el descriptor de fichero *n*)
- **p**, en cuyo caso **ksh** conecta la salida del co-proceso (ver la página xxxx) a la entrada estándar (o descriptor de fichero *n*). Esto hace posible la creación de otro co-proceso y/o pasar la salida del co-proceso a otro comando.

Ejemplos:

```
exec 3<&4     # Abre el descriptor de fichero 3 como una copia del descriptor de
              # fichero 4
exec 3<&4-    # Mueve el descriptor de fichero 4 al 3.
exec 4<&-     # Cierra el descriptor de fichero 4
```

Lectura/Escritura <>palabra n<>palabra

Abre el fichero con el nombre que resulta de la expansión de *palabra*, para lectura y escritura como la entrada estándar (o el descriptor de fichero *n*). Cuando se usa con un comando, redirecciona la entrada (o descriptor de fichero *n*) del comando, desde el fichero expandido desde *palabra*.

Ejemplo:

```
exec 3<> /dev/tty
# Abre /dev/tty sobre el descriptor de fichero 3 para lectura y escritura
```

Escritura >palabra n>palabra >|palabra n>|palabra

ksh abre el fichero que resulta de la expansión de *palabra*, para escribir como salida estándar (o como el descriptor de fichero *n*). Cuando se usa con un comando **ksh** redirecciona la salida (o el descriptor de fichero *n*) del comando, al fichero resultado de la expansión de *palabra*.

Si el fichero no existe, **ksh** lo crea si es posible. Si el fichero existe y la opción **noclobber**:

- Está fijada, **ksh** visualiza un mensaje de error. La sintaxis >|*palabra* (o n>|*palabra*) hace que **ksh** trunque la longitud del fichero a 0 incluso si fijas la opción **noclobber**.
- No está fijada, **ksh** trunca el tamaño del fichero a 0.

Ejemplos:

```
exec 3> foobar      # Abre el fichero foobar para escritura
cat F1 F2 > F3      # Concatena los ficheros F1 y F2, y coloca el resultado en F3
cat F1 F2 >| F3     # Lo mismo que arriba excepto que también funciona si F3 existe
                    # y la opción noclobber está activa.
```

Adición >>palabra n>>palabra

ksh abre el fichero que resulta de la expansión de *palabra* para añadir como salida estándar (o el descriptor de fichero *n*). Cuando se usa con un comando, **ksh** redirecciona la salida estándar (o el descriptor de fichero *n*) del comando, y la añade al final del fichero expandido desde *palabra*.

Si el fichero no existe, **ksh** lo crea si es posible.

Ejemplo:

```
cat F1 >> F2
# Añade el fichero F1 a F2
# Crea F2 si no existía realmente.
```

Duplicando, moviendo y cerrando la salida >&palabra n>&palabra

Palabra se debe evaluar a:

- Un dígito, en cuyo caso **ksh** hace a la salida estándar (o al descriptor de fichero *n*) un duplicado del descriptor de fichero cuyo número viene dado por el dígito.
- Un dígito –, en cuyo caso **ksh** mueve el descriptor de fichero cuyo número está dado por el dígito a la salida estándar (o al descriptor de fichero *n*). **Versión**: dígito – está disponible solamente en versiones de **ksh** posteriores a la de 16/Nov/1988.
- – (menos), en cuyo caso **ksh** cierra la salida estándar (o el descriptor de fichero *n*)
- **p**, en cuyo caso **ksh** conecta la entrada del co-proceso (ver la página xxxx) a la salida estándar (o descriptor de fichero *n*). Esto hace posible dirigir la salida desde cualquier comando al co-proceso.

Ejemplo:

```
foobar |&     # Crea proceso cooperativo foobar
exec 3>&p     # Mueve write end del proceso cooperativo al descriptor de fichero 3
date >&3      # Direcciona la salida de date al proceso cooperativo
exec 3>&-     # Cierra la conexión al proceso cooperativo
```

8. PROCESAMIENTO DE LOS COMANDOS

Este capítulo presenta el orden lógico que sigue **ksh** para leer y procesar un comando. Esto no es necesariamente el orden real dentro del código de cualquier implementación de **ksh** dada.

Los comandos son procesados en dos etapas. En la primera etapa, **ksh** lee cada comando y lo parte en tokens. **ksh** determina si el comando es un comando simple o un comando compuesto para determinar cuanto leer.

En la segunda etapa, **ksh** expande y ejecuta un comando cada vez que es usado. Un comando compuesto tal y como un bucle **while-do-done** se expande y ejecuta cada vez que se repite el bucle. Una función se expande y ejecuta cada vez que es referenciada.

Este capítulo describe como lee **ksh** su entrada, y como expande y ejecuta **ksh** los comandos simples. Vaya a la sección "*Comando compuestos*" xxxx para ver como **ksh** los procesa.

LEYENDO COMANDOS

ksh primero lee los comandos y luego los ejecuta. Esta sección describe cuanto lee **ksh** cada vez, y como **ksh** parte la entrada en comandos.

Partiendo la entrada en Comandos

Es importante que comprendas como los alias y **set –k** interactúa con **ksh** en la lectura de comandos. **set –k** es una característica obsoleta que hace que todos los argumentos de la forma *nombre=valor* sean tratados como asignaciones sin importar donde aparezcan en la línea. Puedes evitar los problemas causados por el orden en el cual ksh lee y procesa los comandos, si tú:

- Usas **alias** y **unalias** sin ningún otro comando en la misma línea
- No utilizas **alias** y **unalias** en los comandos compuestos
- No usas **set –k** nunca. **Bourne shell**: **set –k** es una característica de **ksh** solamente para compatibilidad con la Bourne shell.

ksh lee al menos una línea cada vez. Por lo tanto, alias nuevos y **set –k** no afectan a los comandos siguientes de la misma línea, pero afecta solamente a las líneas siguientes. Esto significa que si tienes dos o más comandos simples o compuestos en una única línea, **ksh** lee todos los comandos de la línea antes de ejecutarlos.

Ejemplo:

```
alias foo=bar; foo
# En este caso, foo no se convertirá en bar. Si foo aparece en alguna línea
# posterior, entonces si se convertirá en bar.
```

ksh lee comandos enteros de una vez. Por lo tanto, si usas alias o **set –k** dentro de un comando compuesto, ellos no afectan a los comandos dentro del comando compuesto. Ellos afectan solamente a la lectura de comandos que son leídos después de la ejecución del comando compuesto. **Nota**: Una definición de función (ver la página xxxx) es un comando compuesto. Por lo tanto, la definición de alias dentro de ellas no afecta a la forma en que se lee la función. Además, una

función podría ser referenciada y por lo tanto ejecutada muchas líneas después del código correspondiente al comando de definición de la función, o nunca más. Por lo que un comando **alias** o **unalias** dentro de comando **function** podría tener efectos solamente muchas líneas después de la definición de la función, o incluso nunca.

Ejemplo:

```
for  i in 0 1
do   if   ((I==0))
     then alias print=date
     fi
     print +%H:%M:%S
done
print +%H:%M:%S
+%H:%M:%S
+%H:%M:%S
07:05:15
```

ksh lee un script punto completo y lo divide en tokens antes de ejecutar ninguno de los comandos del fichero. El script punto es el fichero especificado como el primer argumento a el comando **.** (punto). Por lo tanto, si usas alias o **set –k** en un script punto, no afectan a los comandos del script punto. Afectan solamente a la lectura de comandos posteriores.

ksh lee el número más pequeño de líneas completas que constituyen un comando completo cuando lee su entrada del terminal, un shell script distinto de un script punto, ficheros profile o tu fichero de entorno.

Cuando tecleas un comando de **ksh** incompleto en el terminal, y pulsas RETURN, **ksh** visualiza el prompt secundario, **PS2** (por defecto **>**), para indicar que espera que continúes introduciendo el comando.

Ejemplo:

```
for i in One two three
> do print $i
> done
One
two
three
```

Partiendo la entrada en Tokens

El proceso de partir tus comandos de entrada de **ksh** en tokens se llama análisis léxico y es fue descrito en el capítulo anterior. Un token es uno de los siguientes:

- Operador de redirección de entrada/salida
- Operador de control
- Salto de línea
- Palabra reservada
- Asignaciones simples
- Palabra
- Documento empotrado o here-document.

Cuando tecleas un token incompleto desde un terminal, y después pulsas RETURN, **ksh** visualiza su prompt secundario, **PS2** (por defecto **>**), para indicar que espera que continúes introduciendo el token.

Ejemplo:

```
print “Uno dos tres
> cuatro cinco seis
> siete”
Uno dos tres
cuatro cinco seis
siete
```

ksh realiza sustitución de alias (ver la página xxxx) según va partiendo la entrada en tokens.

ksh convierte cada cadena de mensaje en un cadena entrecomillada con comillas dobles buscando el mensaje en un catálogo de mensajes específico del locale. Si **ksh** no puede encontrar el catálogo de mensaje, o el mensaje no se encuentra en el catálogo, la cadena de mensaje no es traducida.

Determinando el tipo de un Comando

Si el primer token del comando es una de las siguientes palabras reservadas, entonces **ksh** lo lee como un comando compuesto: **{ case for function if until select time while [[! Nota**: **ksh** procesa éstas como palabras reservadas solamente si no existen comillas de ningún tipo, o alguna barra invertida, en o alrededor de ellas, y son el primer token del comando.

Si el primer token del comando es uno de los siguientes operadores:

(**ksh** lo lee como un comando compuesto, hasta el paréntesis correspondiente de cierre **)**.

((**ksh** lo lee como una expresión aritmética hasta los dos paréntesis de cierre correspondientes **))**. **ksh** procesa esto como si fuese un argumento entrecomillado con comillas dobles de **let**.

Si el primer token del comando es uno de los siguientes, entonces **ksh** visualiza un error de sintaxis y, cuando no se está en modo interactivo, se sale:

- Palabras reservadas:
  do done elif else esac fi in then }]]
- Operadores: **| || & && ; ;; |& ;&**

ksh lee cualquier otro token al principio de un comando, incluyendo los operadores de redirección de entrada/salida, como el primer token de un comando simple.

Asignación de variables

La asignación compuesta es un mecanismo para asignar valores a una o más variables. Una asignación compuesta puede aparecer donde quiera que pueda aparecer un token de asignación simple. Una asignación compuesta está en una de las siguientes formas:

variable=(valor ...) para asignar valores a un array indexado *variable* (página xxxx)

variable=([expresión]=valor ...) para asignar valores a un array asociativo *variable* (ver la página xxxx)

variable=(asignación ...) para asignar valores a un conjunto de variables cuyos nombres son de la forma *variable.nombre*. Cada asignación puede ser una de las siguientes:

- Una asignación simple
- Una asignación compuesta
- El comando **typeset** definido en la página xxxx. Utilice un salto de línea o **;** para separar cada comando **typeset**.

En este caso, el valor de *variable* después de una asignación está en el formato de una asignación de variable compuesta que consta de todas las variables cuyos nombres comienzan con *variable.*.

No se permiten espacios o tabuladores antes del **=**. El **(** debe aparecer en la misma línea que el **=**. Se permiten saltos de línea entre los valores o asignaciones y antes del **)**.

Versión: Las asignaciones compuestas están disponibles solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Ejemplos:

```
files=(*)
age= ( [Adam]=18 [Jeff]=21 [Flip]=23 )
point=( x=3.5 y=4.2 )
record= ( name=joe ; typeset –i age=31 )
```

Leyendo comandos simples

ksh lee todos los tokens hasta alguno de los siguientes como un comando simple:

; | & || && |& Salto de línea

ksh organiza los tokens en comandos simples de tres clases:

- Redirecciones de entrada/salida (el operador de redirección de entrada/salida, más la palabra que sigue al operador; e.g., **>x**). **ksh** lee y procesa las redirecciones de entrada/salida de izquierda a derecha. Puedes mezclar redireccionamientos de entrada/salida y palabras de comandos en cualquier orden, aunque no se considere que sea buena práctica hacer eso. Sin embargo, esta habilidad de "*mezclar*" hace posible que le especifiquemos redireccionamientos de entrada/salida como parte de una definición de alias, lo cuál es en algunas ocasiones útil.
- Asignaciones de variable. Si la opción **keyword** (**set –k**) está activa, **ksh** reconoce cualquier palabra en la sintaxis de una asignación de variable (página xxxx) como una asignación de variable. Si la opción **keyword** está desactivada, **ksh** reconoce cualquier palabra en la sintaxis de una asignación de variable como una palabra de asignación de variable, solamente hasta que encuentra un token el cuál no es de este formato y no es una palabra de redireccionamiento de entrada/salida. Si **ksh** encuentra la sintaxis de asignación de variable más adelante, lee y procesa la palabra como una palabra comando.
- Palabras comando. Las palabras restantes son llamadas palabras comando. **ksh** construye el nombre de comando y los argumentos de comando expandiendo las palabras comando. **ksh** chequea la primera palabra comando cuando la lee, para ver si existe un alias de dicho nombre para expandir (ver la siguiente sección). Si el nombre de comando es **alias,**

readonly, typeset o **export**, **ksh** procesa cada argumento del formato de asignación de variable de forma especial, sin tener en cuenta si **keyword** está activa o desactiva. Estos argumentos se leen y expanden con las mismas reglas que las asignaciones de variable, excepto que son expandidas y evaluadas cuando se procesan los argumentos de comando.

Substitución de alias

ksh chequea la palabra de nombre de comando de cada comando simple para ver si es un nombre de alias legal. Si no está entrecomillado de ninguna forma, y es un nombre de alias legal, y existe un alias no-tracked de ese nombre definido, entonces **ksh** chequea para ver si está procesando en ese momento un alias como el mismo nombre. Si lo está procesando, **ksh** no remplaza el nombre de alias. Esto previene la recursividad infinita. Si no está procesando en ese momento un alias con el mismo nombre, **ksh** remplaza el nombre de alias por el valor del alias.

ksh realiza la sustitución de alias como parte del proceso de partir un comando en tokens. Cuando **ksh** realiza la sustitución de alias, el token que contiene el alias se remplaza por los tokens definidos por el valor del alias.

Puedes crear y visualizar los alias definidos con **alias**, y eliminar alias con **unalias**.

Versión: La versión de 16/Nov/1988 tenía una opción **–x** de **alias** para permitir que las definiciones de alias fuesen heredadas por los scripts invocados por nombre. La opción **–x** de **alias** no tiene ningún efecto con las versiones más nuevas de **ksh**.

Si el valor de un alias finaliza con un espacio o un tabulador, **ksh** chequea la siguiente palabra de comando para sustitución de alias. Por ejemplo, el alias prefijado **nohup**, definido más abajo, finaliza en espacio. Por lo que la palabra después de **nohup** será procesada para sustitución de alias.

Ejemplos:

```
alias foo='print '
alias bar='hola mundo'
foo bar
hola mundo
alias od=done
for i in foo bar
do    print $I
od
foo
bar
```

Las definiciones de alias no son heredadas por scripts ejecutados por **ksh** o a través de invocaciones de **ksh** diferentes.

Aliases prefijados

Los alias prefijados son alias que son predefinidos por **ksh**. Puedes quitar su definición o cambiarlos si lo deseas. Sin embargo, recomendamos que nos los cambies, porque esto podría confundirte a ti y/o a otros que esperasen que el alias funcionase de la forma que está predefinida por **ksh**:

```
autoload='typeset -fu'
command='command '
fc=hist
float='typeset –E'
```

```
functions='typeset -f'
hash='alias -t--'
history='hist -l'
integer='typeset -i'
nameref='typeset -n'
nohup='nohup '
r='hist -s'
redirect='command exec'
stop='kill -s STOP'
times='{ {time;}2>&1;}'
type='whence -v'
```

Substitución de mensajes

ksh chequea cada cadena entrecomillada con comillas dobles que está precedida por un **$** sin entrecomillar para realizar una sustitución de mensaje. El **$** es eliminado. Si no está en el locale POSIX, **ksh** busca la cadena en un catálogo de mensajes específico del locale y la remplaza con una cadena contenida en el catálogo de mensaje. En el locale POSIX, o si no se encuentra en el catálogo de mensajes específico del locale, la cadena entrecomillada con comillas dobles no es cambiada.

EXPANDIENDO UN COMANDO SIMPLE

Anteriormente a la ejecución, **ksh** procesa los tokens de palabra de los comandos simples para generar el nombre de comando y/o argumentos de comando según se ha descrito en esta sección.

ksh realiza primero la expansión de la tilde (**~**), sustitución de comando, expansión de parámetro, y expansión aritmética sobre una palabra de izquierda a derecha. Esto es seguido por la partición en campos, y después por la expansión de pathname. La eliminación de comillas siempre se realiza lo último.

Como se expuso más arriba, los comandos simples están compuestos de tres tipos de tokens: palabras de asignación de variables, palabras comando, y redireccionamientos de entrada/salida. Las palabras comando se expanden primero de izquierda a derecha. La siguiente tabla resumen que realiza el procesamiento de **ksh** a cada tipo.

	Asignación de Variable	**Palabra comando**	**Redireccionamiento de Entrada/salida**
Lectura de comandos			
Sustitución de alias	No	Nota 1	No
Sustitución de mensaje	Sí	Sí	Sí
Ejecución de comandos			
Expansión de la tilde	Nota 2	Sí	Sí
Sustitución de comando	Sí	Sí	Sí, nota 3
Expansión de parámetro	Sí	Sí	Sí, nota 3
Expansión aritmética	Sí	Sí	Sí, nota 3
Particionar en campos	No	Nota 4	No
Expansión de pathname	No	Sí, nota 5	Nota 6
Eliminación de comillas	Sí	Sí	Sí

Sí Esto se hace

No Esto no se hace

Nota 1 Siempre se aplica a la primera palabra. Si el valor **alias** finaliza con un espacio o un tabulador, entonces la sustitución de alias también se aplica a la siguiente palabra y así sucesivamente.

Nota 2 Se hace después del **=** y después de cada **:**.

Nota 3 Excepto después de los operadores **<<** y **<<-**.

Nota 4 Se hace solamente sobre porciones de palabras resultantes de la sustitución de comando y expansión de parámetros.

Nota 5 Se hace, a menos que **set –f (set –o noglob)** esté activa

Nota 6 Se hace para shells interactivos solamente si la expansión lleva a un pathname único con las versiones de **ksh** posteriores a la de 16/Nov/1988.

Expansión de la tilde

ksh chequea cada palabra para ver si comienza con un **~** sin entrecomillar. Si es así, entonces **ksh** chequea la palabra hasta un carácter **/** para ver si encaja con:

~ por el mismo	Se remplaza por el valor de la variable **HOME**.
~ seguido por un +	Se remplaza por el valor de la variable **PWD**.
~ seguido por un – (menos)	Se remplaza por el valor de la variable **OLDPWD.**
~ seguido por un nombre de usuario	Se remplaza por el directorio home (de login) del usuario en cuestión
~ seguido por cualquier otra cosa	La palabra original se deja sin cambiarse.

También **ksh** chequea el valor de cada asignación de variable para ver si aparece un **~** detrás de **=** o detrás de un **:** en el valor asignado. Si aparece, **ksh** intenta hacer la expansión de la tilde.

Versión: La expansión de la tilde se hacia mientras se leía el comando con la versión de 16/Nov/1988. La expansión de la tilde se realiza sobre la palabra que sigue al modificador de la expansión de parámetro (ver la página xxxx), cuando la expansión de parámetro no está dentro de comillas dobles en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Substitución de comandos

ksh chequea cada palabra para ver si contiene un comando encerrado en **$(...)** (nueva sustitución de comando), o en un par de comillas simples inversas **`...`** (antigua sustitución de comando. Si es así, **ksh** hace lo siguiente:

- Si utiliza la forma antigua de sustitución de comando, **ksh** procesa la cadena entre las comillas inversas utilizando las reglas de entrecomillado de la página xxxx para construir el comando real.
- Si utilizas la forma de sustitución de comando nueva, **ksh** ejecuta el comando representado por los puntos suspensivos **(...)**.

- **$(...)** o **`...`** son remplazados por la salida del comando representado por los **...**, con los saltos de línea finales (si los tuviese) eliminados. **ksh** no procesa esta salida para expansión de parámetros, expansión aritmética o sustitución de comando.

  Ejemplo:

  ```
  x=$(date)  # A x se le asigna la salida del comando date.
  ```
- Si la sustitución de comandos aparece dentro de unas comillas de agrupación (dobles), entonces **ksh** no procesa la salida del comando para particionamiento de campos o expansión de pathname.

$(<fichero) es equivalente a **$(cat fichero)**, pero se ejecuta más rápido debido a que **ksh** no crea un proceso separado.

La sustitución de comando se realiza en un entorno de subshell. No ocurren efectos laterales en el entorno de **ksh** actual como resultado de ejecutar la sustitución de comando. Por ejemplo, **$(cd)** no cambia el directorio de trabajo del entorno actual.

Expansión aritmética

Cada **$((...))** se remplaza por el valor de la expresión aritmética dentro de los paréntesis dobles.

Ejemplos:

```
x=$((RANDOM%5))
# x recibe un número aleatorio entre 0 y 4
```

Expansión de parámetros

ksh chequea todas las palabras que contienen un **$** sin entrecomillar para ver si el **$** especifica una expansión de parámetros. Si es así, entonces **ksh** remplaza la porción de parámetro de la palabra.

Si el parámetro no está fijado y especificas la opción **nounset (set –u)**, **ksh** visualiza un mensaje de error en el error estándar. Si el error ocurre dentro de un script, el programa termina con un valor de retorno Falso.

Ejemplo:

```
set –u
unset foobar
rm $foobar
ksh: foobar: parameter not set
```

Si incluyes una expansión de parámetro dentro de unas comillas de agrupación (dobles), entonces **ksh** no procesa el resultado de la expansión de parámetros para el particionamiento en campos y expansión de pathname.

Partición de campos

ksh rastrea el resultado de la sustitución de comando, expansión aritmética y expansión de parámetro de las palabras comando en busca de los caracteres delimitadores de campo encontrados en el valor de la variable **IFS**, teniendo en cuenta que la expansión no está dentro de comillas dobles. Si el valor de **IFS** es ***Nulo***, entonces **ksh** no realiza particionamiento de campos. Si no, **ksh** parte el

resultado de la sustitución de comando y la expansión de parámetros en campos distintos según se describe en la página xxxx.

Ejemplo:

```
IFS=: foobar="foo:bar"
set foo:bar $foobar "$foobar"
for i
do print "$i"
done
foo:bar
foo
bar
foo:bar
```

Expansión del pathname

Siguiendo al particionamiento de campos, **ksh** busca en cada campo los siguientes caracteres sin entrecomillar *****, **?**, **[** y **(**, a menos que utilices la opción **noglob (set –f)**. Cualquier carácter (o) que resulta de una expansión está sin entrecomillar, de forma que los patrones que usan **(...)** deben ser presentados como parte de la entrada de comando. Por ejemplo, **print !(*.o)** encajará e imprimirá nombres de ficheros que no acaben en **.o**, mientras que **x='(*.o)'; print $x** no lo hará.

Si ninguno de estos caracteres aparece, el campo permanece sin cambios. De otro modo, **ksh** procesa el campo como un patrón. **ksh** remplaza el patrón que todos los pathnames que encajen, ordenados alfabéticamente, con las siguientes reglas adicionales:

- ***** y **?** en un patrón no encajan con **/**. El pathname debe encajar explícitamente con cada **/** del patrón.
- El pathname debe encajar explícitamente con **.** (punto) cuando **.** es el primer carácter del pathname, y cuando el **.** sigue inmediatamente a un **/**. Si la variable **FIGNORE** está fijada, entonces los nombres de ficheros que encajen con el patrón definido por el valor de la variable **FIGNORE** son excluidos del encaje en lugar de los nombres de ficheros que contienen un **.** (punto) inicial. **Versión**: **FIGNORE** está solamente disponible en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Si no existen encajes, **ksh** deja la palabra tal cuál, sin cambios.

Eliminación de entrecomillado

Los caracteres especiales ****, **"** y **'** son eliminados por **ksh** a menos que ellos mismos estén entrecomillados. **ksh** no elimina las comillas que son el resultado de las expansiones.

Los argumentos ***Nulos*** que son:

- Explícitos (dentro de comillas simples o dobles, por ejemplo **"$nombre"** donde *nombre* no tiene valor), son retenidos.
- Implícitos (por ejemplo, **$nombre** donde *nombre* tiene un valor *Nulo* o no tiene valor), son eliminados

Ejemplos:

```
x='"'  y=""
print "'hello'" ${x}there${x}
```

```
'hello' "there"
set "$y" $y
print $#
1
```

EJECUTANDO UN COMANDO SIMPLE

Esta sección describe como **ksh** busca y ejecuta un comando simple. Un comando simple puede ser un comando de asignación a variable, redirecciones de Entrada/Salida, un comando especial built-in, una función, un comando built-in regular, o un programa.

No Command Name or Arguments

ksh hace cada uno de los redireccionamientos de Entrada/Salida en un entorno de subshell. Por lo tanto, los únicos operadores de redireccionamiento que son útiles en este contexto son > y >|, los cuales puedes usar para crear ficheros.

ksh realiza asignación de variables de izquierda a derecha en el entorno actual.

El valor devuelto por una asignación de variable es *Verdadero* en todos los casos excepto cuando:

- La última asignación de variable implicaba una sustitución de comando que falló.
- El redireccionamiento falló dentro de un script o función
- Intento de asignar a una variable que tiene el atributo readonly (sólo lectura)
- Intento de asignar una expresión aritmética inválida a una variable entera o punto flotante.

Comandos especiales built-in

ksh ejecuta los comandos especiales built-in (ver el capítulo Comandos Built-in xxxx) en el entorno actual. Muchos de estos comandos tienen efectos laterales. Son listados más abajo:

. (punto) **:** (dos puntos) **alias break continue eval exec exit export newgrp readonly return set shift trap typeset unalias unset**

Excepto para **exec**, el redireccionamiento de la entrada/salida se aplica solamente al propio comando built-in. No afecta al entorno actual. Sin embargo, el redireccionamiento de la entrada/salida aplicado a **exec** sin argumentos afecta a los ficheros abiertos en el entorno actual.

ksh procesa estos built-ins de forma especial del modo que se anota más abajo:

- **ksh** evalúa las listas de asignación de variables especificadas con el comando antes del redireccionamiento de la entrada/salida. Estas asignaciones mantienen su efecto cuando el comando acaba.
- Los errores en estos built-ins hacen que el script que los contienen finalice.

El valor de retorno: Se especifica debajo de cada comando en el capítulo de Comandos Built-in (xxxx).

Versión: Los comandos built-in **times** y **wait** no seguirán considerados como tales en versiones de **ksh** posteriores a la versión de 16/Nov/1988. Los comandos built-in **set**, **unset** y **unalias** se han

convertido en comandos built-in especiales en las versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Funciones

Si el comando no es un built-in especial, **ksh** chequea para ver si existe una función definida. Si una función no ha sido definida, pero ha sido marcada como una función sin definir, con **typeset –fu**, **ksh** buscará en **FPATH** para encontrar la definición de función y ejecutarla antes de hacer una búsqueda del **PATH**.

Los redireccionamientos de entrada/salida especificados con la referencia de función se aplican solo a la propia función. No afectan al entorno actual. Los redireccionamientos de entrada/salida especificados dentro de la función con **exec** no afectan al entorno actual.

Los siguientes son compartidos por la función y el script que la llama, de forma que pueden producir efectos laterales:

- Valores de variables y atributos, a menos que utilices **typeset** dentro del cuerpo de la función de una función no-POSIX para declarar una variable local.
- Directorio de trabajo
- Aliases, definiciones de funciones, y atributos
- Parámetro especial **$**. **Precaución**: Ten cuidado de que las funciones no creen un fichero temporal con el mismo nombre que un fichero temporal creado por el script que la llama.
- Ficheros abiertos

Si declaras un función con la sintaxis **function** *nombre*, entonces **ksh** ejecuta la función en un entorno de función separado. Los siguientes no son compartidos entre este tipo de función y el script que la invoca, y por lo tanto no puede causar efectos laterales:

- Parámetros posicionales
- Parámetro especial #.
- Variables en una lista de asignación de variable cuando se invoca la función
- Variables declaradas utilizando **typeset** dentro de la función
- Opciones
- Traps o trampas. Sin embargo, las señales ignoradas por el script que hace la llamada también serán ignoradas por la función.

ksh ejecuta una trap o trampa sobre **EXIT** que se fije dentro de este tipo de funciones justo después de que finaliza la ejecución de la función, pero en el entorno del script que realiza la llamada.

Si declaras una función con el formato *nombre ( )* entonces **ksh** ejecuta la función en el entorno actual del mismo modo que se hace con un script punto. Las siguientes no son compartidas entre este tipo de funciones y el script que hace la llamada, y no pueden causar por lo tanto efectos laterales:

- Parámetros posicionales

- Parámetro especial #

Versión: **ksh** ejecuta una función *nombre ( )* en el entorno actual como un script punto solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

El valor de retorno de una función es el valor de retorno del último comando ejecutado dentro de la función.

Utilidades built-in regulares

ksh ejecuta las siguientes utilidades built-in regulares en el entorno actual:

bg builtin cd command disown echo false fg getconf getopts hist jobs kill print printf read test true umask wait whence.

Versión: **builtin**, **command**, **disown**, **getconf**, **hist** y **printf** están disponibles solamente en versions de **ksh** posteriores al 16/Nov/1988.

Son ejecutadas sin tener en cuenta el contenido de la variable **PATH**.

El comando **builtin** puede ser usado para añadir o eliminar built-ins regulares.

Muchos de estos comandos tienen efectos laterales sobre el entorno actual. Estos efectos laterales son listados en el capítulo Comandos Built-ins xxxx.

Búsqueda de PATH

Si el comando dado contiene un carácter */*, entonces **ksh** ejecuta el comando con el pathname especificado. De otra forma, **ksh** chequea cada uno de los directorios localizados en el parámetro de búsqueda de path **PATH**. La búsqueda tiene lugar en el orden en el que se listan los directorios en la variable **PATH**. Si **ksh** no localiza el comando dado en ninguno de los directorios del parámetro **PATH**, entonces **ksh** visualiza un mensaje de error y el valor de retorno es 127.

Si **ksh** localiza el comando dado en un directorio en el parámetro **PATH**, entonces **ksh** busca ese directorio en el parámetro de búsqueda de path de funciones **FPATH**. Si el directorio se localiza en **FPATH**, entonces **ksh** lee este fichero en el entorno actual y después ejecuta la función. **Precaución**: Un directorio que está en **PATH** y **FPATH** no debería contener otros ficheros ejecutables que definiciones de funciones.

De otro modo, si **ksh** determina que el comando dado ha sido implementado como un built-in, entonces **ksh** ejecuta el built-in en el entorno actual.

De otro modo, **ksh** ejecuta el comando dado como un programa en un entorno separado. Por lo tanto los programas no pueden tener efectos laterales sobre el entorno actual.

ksh recuerda los pathnames de cuales quiera comandos ejecutados y no los buscará de nuevo a menos que la variable **PATH** haya sido asignada de nuevo.

Si el programa termina con una valor de vuelta Falso (distinto de cero) y si se especifica una captura (trap) sobre **ERR**, **ksh** ejecuta la acción asociada con ella. Si la opción **errexit (set –e**) está activa, **ksh** se sale con un valor de retorno del comando que terminó con el valor Falso. Si no, **ksh** ejecuta el siguiente comando.

Si el programa termina con un valor de retorno Verdadero, **ksh** procesa el siguiente comando.

Versión: En versiones de **ksh** anteriores y la de 16/Nov/1988 incluida, los built-ins eran siempre ejecutados primeros. También, los pathnames de comandos no eran recordados a menos que la opción **trackall** estuviese activa (lo cual era por defecto).

9. COMANDOS COMPUESTOS

Este capítulo define el formato y significado de los comandos compuestos de **ksh**. Un comando compuesto es un pipeline o una lista, o incluso el comando compuesto comenzar con una palabra reservada o el operador de control **(**.

El redireccionamiento de entrada/salida después de cualquier comando compuesto (excepto del pipeline, del comando **time**, o de una lista) se aplica al comando completo. El redireccionamiento de la entrada/salida después de un pipeline, el comando **time** o una lista se aplica solamente al último comando. Utilice el comando de agrupación que empieza con la palabra reservada **{** alrededor de cualquier comando si necesitas especificar un redireccionamiento de entrada/salida para la secuencia completa de comandos. El redireccionamiento de entrada/salida no afecta al entorno en el cuál **ksh** ejecuta el comando.

No puedes especificar asignaciones de variables con un comando compuesto.

Vea el capítulo de "*Referencia rápida*" xxxx para una descripción mas concisa de la gramática del lenguaje KornShell.

COMANDO PIPELINE

comando [| [Newline...] comando]...

Un pipeline es una secuencia de uno o más comandos simples o compuestos, cada uno separado por un **|**. Como especifican el formato y definición de más arriba, un pipeline puede constar de un comando simple o compuesto sin operador de pipeline **|**. En la práctica no nos referiremos a esto como pipeline (ya que no existe operador de pipeline **|**). Definimos esto así de forma que no tenemos que designar especialmente este caso para comandos que permiten pipelines dentro de ellos.

La salida estándar de cada comando a excepción del último, se conecta con la entrada estándar del comando siguiente.

ksh ejecuta cada comando excepto posiblemente el último como un proceso separado. Si la opción **monitor** está desactivada, **ksh** espera a que termine el último comando. De otro modo, **ksh** ejecuta cada pipeline como un trabajo separado. Por ejemplo, **date | wc** sería ejecutado como un trabajo. **ksh** espera a que todos los procesos de un pipeline se completen.

Valor de retorno: El valor de retorno del último comando especificado.

Ejemplo:

```
grep foo bar | sort | uniq
food menu
```

COMANDO TIME

time [*pipeline****]***

Si se especifica *pipeline*, **ksh** ejecuta *pipeline*, y visualiza en el error estándar el tiempo transcurrido, tiempo de usuario y tiempo de sistema. De otro modo, **ksh** visualiza el tiempo

cumulativo para el shell y sus hijos. **Versión**: Con la versión de 16/Nov/1988 de **ksh**, **time** necesitaba *pipeline*, y el comando built-in **times** se usaba para visualizar el tiempo cumulativo del shell y sus hijos.

Los redireccionamientos a continuación de *pipeline* se aplican al último comando del *pipeline* en lugar de aplicarse a **time**. Para salvar la salida de **time** a un fichero, encierre el comando **time** en uno de los comandos de agrupación (vea la página xxxx) y redirija el error estándar del comando agrupado al fichero.

Valor de retorno: Devuelve el valor de *pipeline*.

Ejemplos:

```
time grep foo bar | sort | uniq
food menu

real  0m2.03s
user  0m0.85s
sys   0m0.49s

time
user  0m0.85s
sys   0m0.49s
```

COMANDO NEGACIÓN

! pipeline

ksh ejecuta *pipeline*. El valor de retorno de **!** es:

- Verdadero (cero) si el valor de retorno de *pipeline* es Falso (distinto de cero)
- Falso (distinto de cero) si el valor de retorno de *pipeline* es Verdadero (cero)

Ejemplo:

```
if     ! grep –c ^$user: /etc/passwd
then   print No existe una cuenta para $user
fi
```

Versión: **!** está disponible solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

COMANDOS DE LISTA

lista

Una *lista* puede ser un *pipeline*, o cualquier combinación de los siguientes formatos. Esto es, donde quiera que aparezca *lista* en un formato más abajo, puedes sustituir *pipeline* o un formato completo de forma recursiva, hasta el nivel de profundidad que desees.

Los operadores de lista tienen precedencia menor que el operador pipeline **|**. Si especificas dos operadores o más en la misma *lista*, **ksh** los evalúa de izquierda a derecha, y utiliza la siguiente precedencia:

- Mayor: **&& ||**
- Menor: **; & |&**

lista [&& [newline...] pipeline]... Lista And

ksh ejecuta el primer *pipeline*. Si su valor de retorno es:

- Verdadero: **ksh** ejecuta el segundo *pipeline*, y así sucesivamente con los siguientes *pipelines*, mientras que el valor de retorno del *pipeline* ejecutado anteriormente sea *Verdadero*.
- Falso (distinto de cero): **ksh** no ejecuta los *pipelines* restantes.

Valor de retorno: El valor de retorno del último *pipeline* ejecutado por **ksh**.

Ejemplo:

```
cd foobar && print –r $PWD
```

lista [|| [newline…] pipeline]... Lista Or

ksh ejecuta el primer *pipeline*. Si su valor de retorno es:

- Verdadero: **ksh** no ejecuta los *pipelines* restantes.
- Falso (distinto de cero): **ksh** ejecuta el segundo *pipeline*, y así sucesivamente con los siguientes *pipelines*, mientras que el valor de retorno del *pipeline* ejecutado anteriormente sea *Falso*.

Valor de retorno: El valor de retorno del último *pipeline* ejecutado por **ksh**.

Ejemplo:

```
read –r linea || error_exit 'Final de fichero inexperado'
```

lista [; pipeline]... Lista Secuencial

ksh ejecuta cada uno de los *pipelines* secuencialmente.

Valor de retorno: El valor de retorno del último *pipeline* ejecutado por **ksh**.

Ejemplo:

```
who|wc ; date
```

lista & [pipeline &]... Procesos desatendidos o background

ksh ejecuta cada uno de los *pipelines* sin esperar a que termine ninguno de ellos. Si la opción **monitor** está activa, **ksh** ejecuta cada *pipeline* como un trabajo separado.

Valor de retorno: Verdadero

Ejemplo:

```
nohup find / -name foobar –print &
```

lista |& Co-Procesos

ksh ejecuta *lista* como un trabajo separado con su entrada y salida estándar conectadas a **ksh**.

Para escribir en la entrada estándar de este proceso, utilice **print –p**. Para leer la salida estándar de este proceso, utilice **read –p**.

Valor de retorno: Verdadero

Ejemplo:

```
ed – foobar |&
```

***[newline...] lista [newline...]* Lista Compuesta**

Cuando aparece una lista dentro de alguno de los comandos compuestos listados más abajo, puede ser opcionalmente precedida y seguida por uno o más Newline (saltos de línea)

Puedes usar uno o más saltos de línea en lugar de **;** para especificar una lista secuencial dentro de una lista compuesta.

COMANDOS CONDICIONALES

[[expresión-test [newline…]]]

Nota: Debes teclear los corchetes que aparecen en negrita.

expresión-test debe ser una de las primitivas de expresión condicional definidas en la página xxxx, o alguna combinación de éstas (las de la página xxxx) formadas mediante combinación de una o más de ellas con una de las siguientes. Las siguientes se listan en orden de precedencia, de mayor a menor.

- (expresión-test). Su valor es el valor de *expresión-test*. Los **( )** se usan para saltarse las reglas de precedencia normales
- ! expresión-test: Negación lógica de expresión-test
- expresión-test && expresión-test: Toma el valor Verdadero si ambas expresiones-test son ciertas. La segunda expresión-test se expande y evalúa solamente si la primera expresión-test es cierta.
- expresión-test || expresión-test: Toma el valor Verdadero si una de las dos expresiones-test o ambas son verdaderas. La segunda expresión-test se expande y evalúa solamente si la primera expresión-test es falsa.

ksh expande el o los operadores de cada primitiva de expresión condicional para hacer sustitución de comando, expansión de parámetros, expansión aritmética, y eliminación de comillas según se requiera para evaluar el comando. **ksh** chequea la expresión primitiva para determinar si es Verdadera o Falsa.

Valor de retorno: El valor de *expresión-test*.

Ejemplo:

```
[[ foo > bar && $PWD –ef . ]]  &&  print foobar
foobar
```

if ... then ... elif ... else … fi

if *lista-compuesta*
then *lista-compuesta*
[elif *lista-compuesta*
then *lista-compuesta*
...
[else *lista-compuesta***]**

fi

ksh ejecuta la *lista-compuesta* de **if**. Si el valor de retorno es:

- Verdadero: **ksh** ejecuta la *lista-compuesta* del **then**.
- Falso: **ksh** ejecuta cada *lista-compuesta* de **elif** (si existe alguna) en orden, hasta que alguna de ellas tenga un valor de retorno de Verdadero. Si no existen *lista-compuestas* de **elif**, o si ninguna tiene un valor de retorno de Verdadero, **ksh** ejecuta la *lista-compuesta* del **else**, si existe.

Valor de retorno:

- Valor de retorno de la última *lista-compuesta* de **then** o **else** que se haya ejecutado
- Verdadero si no se ejecutó ninguna *lista-compuesta* ni de **then** ni de **else**.

Ejemplo:

```
if    ((score < 65))
then  grade=F
elif  ((score < 80))
then  grade=C
elif  ((score < 90))
then  grade=B
else  grade=A
fi
```

case ... esac

case *palabra* **in**
[[(] patrón [| patrón]...) *lista-compuesta* **;;]**
[[(] patrón [| patrón]...) *lista-compuesta* **;&]**
...
esac

ksh ejecuta la primera *lista-compuesta* de comandos para la cual *palabra* encaje con *patrón*.

ksh expande *palabra* para sustitución de comando, expansión de parámetros, expansión aritmética y eliminación de comillas.

ksh expande cada lista de patrones separados por **|**, en orden, para sustitución de comando, expansión de parámetro, y eliminación de comillas. El orden de evaluación de patrones dentro de la misma lista no está definido. Si el valor expandido de *palabra* encaja con el patrón resultante de expandir el valor de *patrón*, la *lista-compuesta* correspondiente se ejecuta.

Una vez que **ksh** encaja un patrón, no ejecuta ninguna *lista-compuesta* más, a menos que la *lista-compuesta* vaya seguida de un **;&**. En este caso, **ksh** will fall through (pasará) a la siguiente *lista-compuesta*.

El paréntesis antes de cada lista de *patrones* es opcional. **Versión**: El paréntesis es necesario cuando un comando **case** aparece dentro de una sustitución de comando **$(...)** con la versión de **ksh** de 16/Nov/1988.

Valor de retorno:

- Si la *palabra* encaja con algún *patrón*, el valor de retorno de la última *lista-compuesta* que **ksh** ha ejecutado.

- Si *palabra* no encaja con ningún *patrón*, Verdadero.

Ejemplo:

```
case $x in
-d*)  dflag=1;&          # continua
-e*)  eflag=1;;
“”)   print –r –u2 -- "x debe tener un valor.”;;
*)    if  [[ ! –r $x ]]
      then print –ru2 -- "$x: no tiene permiso de lectura.”
      fi;;
esac
```

COMANDOS ITERATIVOS

select ... do ... done

select *variable [***in** *palabra…]*
do *lista-compuesta*
done

ksh realiza sustitución de comandos, expansión de parámetros, expansión aritmética, particionamiento de campos, expansión de pathname y eliminación de comillas para cada *palabra* para generar una lista de items, antes de procesar el comando *lista-compuesta* de **do**. Si no especificas **in** *palabra*, **ksh** usa los parámetros posicionales empezando en el **1** como la lista de items, como si hubieses especificado **in “$@”**.

ksh visualiza los items en una o más columnas en el error estándar, cada uno precedido por un número, y después visualiza el prompt **PS3**. El número de columnas está determinado por el valor de la variable **LINES** y el valor de la variables **COLUMNS**.

ksh entonces lee una línea de selección de la entrada estándar. Si la línea es el número de uno de los items visualizados, **ksh** fija el valor de la variable *variable* al ítem correspondiente a este número. Si la línea está vacía, **ksh** visualiza de nuevo la lista de ítems y el prompt **PS3**; **ksh** no ejecuta *lista-compuesta*. De otro modo, **ksh** fija la variable *variable* a Nulo.

ksh salva los contenidos de la línea de selección leída de la entrada estándar en la variable **REPLY**.

ksh ejecuta *lista-compuesta* para cada selección hasta que **ksh** encuentra un comando **break**, **return** o **exit** en *lista-compuesta*. El comando **select** también termina cuando encuentra un *Final-de-fichero*.

Si el valor de la variable **TMOUT** es mayor que cero, **select** finalizará su ejecución una vez transcurridos el número de segundos especificado en **TMOUT**. **Versión**: Disponible solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Ejemplo:

```
PS3=’Por favor introduzca un número: ‘
select i in fo*
do    case $i in
      food|fool|foot)
            print Buena elección
            break;;
      for|foxy)
            print Poble elección;;
```

```
      *)    print –u2 ‘Número inválido’;;
      esac
done
1) food
2) fool
3) foot
4) for
5) foxy
Por favor introduzca un número:
```

Valor de retorno:

- Verdadero si no se ejecutó ninguna *lista-compuesta*.
- Falso si se agota el tipo de elección del **select**
- Valor devuelto por la última *lista-compuesta* ejecutada.

for ... do ... done

for *variable [***in** *palabra…]*
do *lista-compuesta*
done

ksh realiza sustitución de comandos, expansión de parámetros, expansión aritmética, particionamiento en campos, expansión de pathname y eliminación de comillas para cada *palabra* para generar una lista de items, antes de procesar el comando *lista-compuesta* de **do**. Si no especificas **in** *palabra*, **ksh** utiliza los parámetros posicionales empezando en **1** como la lista de items tal y como si hubieses especificado **in “$@”**.

ksh le asigna a *variable* cada ítem por turno, y ejecuta *lista-compuesta*. La ejecución termina cuando no haya más items. Si *variable* tiene el atributo referencia fijado, entonces se creará una nueva referencia para cada *palabra*.

Ejemplo:

```
for i in fo*
do  print “$i”
done
food
fool
foot
for
foxy
```

Valor de retorno:

- El valor de retorno de la última *lista-compuesta* ejecutada.
- Verdadero si no se ejecutó ninguna *lista-compuesta*.

for **((***[expresión_inicial];[condición_bucle];[expresión_bucle]***))**
do *lista-compuesta*
done

El comando **for** aritmético es muy similar a la sentencia **for** del lenguaje de programación C. *expresión_inicial*, *condicion_bucle y expresión_bucle* son expresiones aritméticas. **ksh** evalúa *expresión_inicial* antes de ejecutar el bucle **for**. Utilice el operador coma dentro de *expresión_inicial* para especificar inicializaciones múltiples. **ksh** evalúa *condición_bucle* antes de cada iteración. Si

condición_bucle es distinto de cero, entonces *lista-compuesta* se ejecuta de nuevo. *expresión_bucle* especifica una expresión que se ejecuta después de cada iteración.

Si se omite *condición_bucle* su valor por defecto es **1**.

Versión: El **for** aritmético esta disponible solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Ejemplo:

```
for    ((i = 0; i < 5; i++))
do     print -- ${array[i]}     # Imprime los primeros 5 elementos del array
done
```

Valor de retorno:

- Valor de retorno de la última *lista-compuesta* ejecutada.
- Verdadero si no se ejecutó ninguna *lista-compuesta*.

while ... do ... done

while *lista-compuesta*
do *lista-compuesta*
done

El comando **while** ejecuta repetidamente la *lista-compuesta* de **while**. En cada repetición, si el valor de retorno de *lista-compuesta* es:

- Verdadero: Ejecuta la *lista-compuesta* del **do**
- Falso (distinto de cero): El bucle termina

Un comando **break** dentro de la *lista-compuesta* del **do** hace que el comando **while** termine con un valor de verdadero. Un comando **continue** hace que termine la *lista-compuesta* del **do** y se ejecute de nuevo la *lista-compuesta* del **while**.

Valor de retorno:

- Valor de retorno de la última *lista-compuesta* de **do** ejecutada.
- Verdadero si no se ejecutó ninguna *lista-compuesta* de **do**.

until ... do ... done

until *lista-compuesta*
do *lista-compuesta*
done

El comando **until** ejecuta repetidamente la *lista-compuesta* del **until**. Para cada repetición, si el valor de retorno de *lista-compuesta* es:

- Falso (distinto de cero): Ejecuta la *lista-compuesta* de **do**.
- Verdadero: Finaliza el bucle

Un comando **break** dentro de la *lista-compuesta* del **do** hace que el comando **until** finalice con un valor de retorno de verdadero. Un comando **continue** hace que finalice la *lista-compuesta* del **do** y se ejecute de nuevo la *lista-compuesta* del **until**.

Valor de retorno:

- Valor de retorno de la última *lista-compuesta* de **do** ejecutada
- Verdadero si no se ejecutó ninguna *lista-compuesta* de **do**.

Ejemplo:

```
until cc –c foo.c
do     vi foo.c
done
```

COMANDOS DE AGRUPACIÓN

(lista-compuesta) Agrupación de subshell

ksh ejecuta *lista-compuesta* en un entorno de subshell. Por lo tanto, no existirán efectos laterales con el entorno actual.

Precaución: Si necesitas anidar este comando, debes insertar Espacios, tabuladores o saltos de línea entre los dos paréntesis abiertos para evitar la evaluación aritmética.

Valor de retorno: Valor de retorno de *lista-compuesta*.

Ejemplo:

```
( find . –print | wc ) >foobar 2>&1 &
```

{lista-compuesta} Agrupación por llaves

ksh ejecuta la *lista-compuesta* en el entorno actual.

Precaución: **{** y **}** son palabras reservadas aquí. Ver la página xxxx para consultar las reglas que gobiernan las palabras reservadas.

Valor de retorno: Valor de retorno de la *lista-compuesta*.

Ejemplo:

```
{ time foobar ;} 2> savetimes
```

COMANDOS ARITMÉTICOS

((palabra...))

ksh realiza la sustitución de comando, expansión de parámetros, expansión aritmética y eliminación de comillas para cada *palabra* para generar una expresión aritmética que se evalúa tal y como se describe en la página xxxx.

Valor de retorno: Verdadero si la expresión aritmética se evalúa a un valor distinto de cero; si no Falso.

DEFINICIÓN DE FUNCIONES

function *[variable.]identificador*
{
    lista-compuesta
}

[variable.]identificador()

{

lista-compuesta

}

Cualquiera de estas declaraciones define una función la cual es referenciada por *variable.identificador* o *identificador*. El cuerpo de la función es la *lista-compuesta* de comandos entre el **{** y el **}**. Si declaras una función con el formato **function *nombre***, entonces **ksh** ejecuta la función en un entorno de función separado descrito en la página xxxx. Si declaras una función con el formato ***nombre()***, entonces **ksh** ejecuta la función en el entorno actual. Esta forma es conocida por función POSIX y es incluida por compatibilidad con el estándar POSIX. Vea la página xxxx para una descripción de ejecución de función.

Los nombres de función de la forma *variable.identificador* son llamados funciones ***disciplina***. Está permitida la definición de tres funciones *disciplina* especiales para todas las variables: **get**, **set** y **unset**. Se usan para definir que una función se ejecute cuando una variable es expandida, asignada y unset (desasignada) (respectivamente). Se pueden definir funciones *disciplina* adicionales para variables definidas mediante built-ins añadidos a **ksh** mediante una librería de aplicación o un usuario.

Versión: **ksh** ejecuta una función *identificador ( )* en el entorno actual solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988. Con la versión de **ksh** de 16/Nov/1988, el nombre de una función estaba limitado a un *identificador*.

Precaución: **{** y **}** son palabras reservadas aquí. Vea la página xxxx para consultar las reglas que gobiernan las palabras reservadas.

Valor de retorno: Verdadero

Ejemplos:

```
function afirmativo # pregunta
{
       typeset –l reply
       while true
       do     read –r “reply?$1? “ || return 1
              case $reply in
              y|yes)  return 0;;
              n|no)   return 1;;
              *)      print ‘Por favor responda y o n’;;
              esac
       done
}
# Lo siguiente realiza una llamada a la función
while afirmativo ‘Quieres continuar?’
do     foobar
done

# Lo siguiente visualiza un mensaje cuando se realiza una asignación a foo
function foo.set
{
       print “La variable foo[${.sh.subscript}] vale ${.sh.value}”
}
foo[3+4]=bar
La variable foo[7] vale bar
```

10. PARÁMETROS

Las entidades de **ksh** que almacenan valores son llamadas parámetros. Algunos parámetros tienen nombres predefinidos por **ksh**, y valores que son fijados por **ksh**. Otros parámetros también tienen nombres predefinidos por **ksh**, pero valores que son fijados por ti. Y otros parámetros (llamados variables de usuario) tienen nombres que tú eliges y valores que tu les das y utilizas.

Los parámetros nombrados, esto es, los parámetros denotados por un *nombredevariable*, tienen atributos que puedes fijar. Puedes usar atributos para formatear el dato, y para otros propósitos.

Existen varios modificadores que puedes usar para alterar el valor de un parámetro cuando es referenciado. Un ejemplo importante es las operaciones de subcadenas.

Todas estas operaciones se discuten en este capítulo.

CLASES DE PARÁMETROS

Parámetros nombrados (Variables)

Los parámetros nombrados son llamados variables y son denotados por un nombre de variable. Tú:

- Asignas el valor de variables con una lista de asignación de variable
- Asignas / desasignas atributos con **typeset**
- Desasignas los valores y atributos de variables con **unset**.

Precaución: Por convención, **ksh** (y otros comandos) utilizan nombres de variables que contienen tres o más caracteres en mayúsculas (por ejemplo, **CDPATH**) para su propio uso. Por lo tanto, te sugerimos que no utilices nombres identificadores de tres o más caracteres en mayúsculas para variables de usuario. Si lo haces, corres el riesgo de una release futura de **ksh** que utilice el mismo nombre para su propio uso. Además, los nombres de variables que comienzan con **.sh.** están reservadas para ser usadas por **ksh**.

Muchas variables son heredadas a través del entorno del proceso padre.

Las variables pueden también ser arrays (vea la página xxxx)

Parámetros Posicionales

Los parámetros posicionales son parámetros que están denotados por uno o más dígitos. Se les asignan valores inicialmente cuando invocas a **ksh**, o cualquier shell script o función, tal y como sigue:

- Parámetro **0** es el nombre del shell, script o función
- Parámetro **1, 2, ...** son los valores de cada uno de los argumentos del shell, script o función según fue invocado.

Puedes reasignar o desasignar todos los parámetros todos los parámetros posicionales desde **1** en adelante con **set**. No puedes volver a fijar el valor de un parámetro posicional individual.

Puedes desplazar los parámetros posicionales (excepto el **0**) solamente a la izquierda (por ejemplo, puedes mover **3, 4, 5, ...** a **1, 2, 3, ...**), con **shift 2**.

Parámetros Especiales

Le puedes asignar a cada variable uno o más de los siguientes atributos. Cuando cambias el atributo de una variable, el valor de la variable se expande todo lo que puede para adecuarse al nuevo atributo. Utilice **typeset** para activar y desactivar atributos, o para listar los atributos.

Nota: No tienes que especificar el atributo entero o punto flotante para usar variables dentro de expresiones aritméticas. Sin embargo, el desempeño podría ser mejor si especificas estos atributos.

–u Mayúsculas

Cuando quiera que **ksh** expande la variable, **ksh** cambia los caracteres en minúsculas a mayúsculas. **ksh** desactiva el atributo de minúsculas.

Ejemplo:

```
typeset –u x=abc
print $x
ABC
```

–l Minúsculas

Cuando quiera que **ksh** expande la variable, **ksh** cambia los caracteres en mayúsculas a minúsculas. **ksh** desactiva el atributo de mayúsculas.

Ejemplo:

```
typeset –l x=ABC
print $x
abc
```

–E o **–E***n* **Punto flotante – Notación científica**

Si después de **–E**, tú:

- No especificas *n*, el valor por defecto es 10.
- Especificas *n*, **ksh** expande el valor a ese número de dígitos significativos

Cuando quiera que asignes un valor a la variable, el valor se evalúa como una expresión aritmética de punto flotante. Cuando imprimes el valor de la variable, será visualizado en el formato **printf %g** (vea la página xxxx)

Puedes usar el alias prefijado **float** para declarar variables punto flotante.

Versión: El atributo **–E** está disponible solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

–F o **–F***n* **Punto flotante – Precisión fija**

Si después de **–F**, tú:

- No especificas *n*, el valor por defecto es 10.
- Especificas *n*, **ksh** expande el valor a ese número de dígitos después del punto decimal.

Cuando quiera que asignes un valor a la variable, el valor se evalúa como una expresión aritmética de punto flotante. Cuando imprimes el valor de la variable, será visualizado en el formato **printf %f** (vea la página xxxx)

Versión: El atributo **–F** está disponible solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

–i o **–i***base* **Entero**

Si después de **–i**, tú:

- No especificas *base*, el valor por defecto es 10 (decimal).
- Especificas *base*, **ksh** expande el valor en esa base aritmética. No puedes especificar una base mayor a 64. **Versión**: en la versión de **ksh** de 16/Nov/1988, la base máxima era 36.

Cuando quiera que asignes un valor a la variable, el valor se evalúa como una expresión aritmética entera.

Si la base es otra distinta de 10, **ksh** preposiciona el número de base seguido de un signo **#**, al valor de la variable cuando es expandida.

Puedes usar el alias prefijado **integer** para declarar variables enteras.

Ejemplo:

```
integer x=6
typeset –i8 y=x+x
print $y
8#14
```

–A Array asociativo

Define un array asociativo, como opuesto a un array indexado. El subscript para un elemento de un array asociativo es una cadena arbitraria.

Versión: el atributo **–A** está disponible solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Ejemplo:

```
typeset –A color
color[manzana]=roja color[platano]=amarillo color[mora]=violeta
for i in ${!color[@]}    # lista de subíndices del array
do    print $i ${color[$i]}
done
manzana roja
mora violeta
platano amarillo
```

–L o **–L***ancho* **Justificado a la izquierda**

ancho es un número. Si no especificas *ancho*, entonces **ksh** utiliza el número de caracteres de la primera asignación a la variable.

Cuando **ksh** expande la variable, justifica los caracteres a la izquierda para adecuarse a *ancho*, y rellena con espacios en blanco al final, si son necesarios, hasta alcanzar una longitud total de *ancho*.

Si asignas un valor a la variable que es demasiado grande para adecuarse a *ancho*, **ksh** trunca los caracteres que sobran por la derecha.

ksh desactiva el atributo justificado a la derecha.

Dependiente de la implementación: En la versión multibyte de **ksh**, *ancho* se refiere al número de columnas (en lugar de al número de caracteres). Durante la expansión, si no existe hueco suficiente para el último carácter completo sin sobrepasar *ancho*, **ksh** no incluye ese carácter en la expansión, y utiliza espacios al final hasta completar y llegar a *ancho*.

Ejemplo:

```
typeset –L3 x=abcd y
y=3
print “$y-$x”
3  -abc
```

–LZ o **–LZ***ancho* **Quita los ceros iniciales**

Este es similar al atributo justificado a la izquierda anterior, excepto que cuando quiera que **ksh** expande la variable, elimina los ceros a la izquierda (iniciales).

ksh desactiva el atributo justificado a la derecha.

Ejemplo:

```
typeset –LZ3 x=abcd y z=00abcd
y=03
print “$y-$x z=$z”
3  -abc z=abc
```

–R o **–R***ancho* **Justificado a la derecha**

ancho es un número. Si no especificas *ancho*, entonces **ksh** utiliza el número de caracteres de la primera asignación a la variable.

Cuando **ksh** expande la variable, justifica los caracteres a la derecha para adecuarse a *ancho*, y rellena con espacios en blanco iniciales (a la izquierda), si son necesarios, hasta alcanzar una longitud total de *ancho*.

Si asignas un valor a la variable que es demasiado grande para adecuarse a *ancho*, **ksh** trunca los caracteres que sobran por la izquierda.

ksh desactiva el atributo justificado a la izquierda.

Dependiente de la implementación: En la versión multibyte de **ksh**, *ancho* se refiere al número de columnas (en lugar de al número de caracteres). Durante la expansión, si no existe hueco suficiente para el primer carácter completo sin sobrepasar *ancho*, **ksh** no incluye ese carácter en la expansión, y utiliza espacios al principio hasta completar y llegar a *ancho*.

Ejemplo:

```
typeset –R3 x=abcd y
y=3
print “$y-$x”
  3-bcd
```

–Z o **–Z***ancho* **Relleno con ceros**

–RZ o **–RZ***ancho* **Relleno con ceros**

Esto es similar al atributo justificado a la derecha. Sin embargo, cuando **ksh** expande la variable le preposiciona ceros iniciales por la izquierda. **ksh** realiza esto solamente si es necesario, y solamente si el primer carácter (distinto de espacio o tabulador) es un dígito. Si el primer carácter no es un dígito, entonces **ksh** lo rellena con espacios iniciales.

ksh desactiva el atributo justificado a la izquierda.

Ejemplo:

```
typeset –Z3 x=abcd y
y=3
print "$y-$x"
003-bcd
```

–r Solo lectura

Una vez que fijas este atributo, obtendrás un mensaje de error si intentas cambiar el valor de esta variable, desactiva el atributo solo lectura, o desasignalo. Sin embargo, **ksh** puede todavía cambiar el valor si es una variable que **ksh** cambia automáticamente, tal y como **PWD**.

Puedes utilizar **readonly** o **typeset –r** para fijar este atributo. Dentro de una función, **typeset** crea una variable local mientras que **readonly** no lo hace.

Ejemplo:

```
readonly foo=bar
foo=nobar
ksh: foo: is read only
unset foo
ksh: foo: is read only
```

–n Referencia de nombre

Hace que el valor de una variable sea tratado como una referencia a otra variable de forma que las variables pueden ser nombradas o referidas indirectamente. Las variables de referencia son usadas a menudo dentro de funciones cuyos argumentos son los nombres de variables de shell. El atributo es particularmente útil cuando la variable de shell es un array.

Los nombres de variables de referencia no pueden contener un **.** (punto). Cuando se referencia una variable shell, la parte de variable hasta el primer **.** se chequea para ver si encaja con el nombre de una variable de referencia. Si lo hace, entonces el nombre de la variable utilizada realmente consiste de la concatenación del nombre de la variable definida por la referencia, más la parte restante del nombre de variable original.

Versión: El atributo **–n** está solamente disponible en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Ejemplo:

```
typeset –n z=foo.bar   # z es un nombre de referencia para foo.bar
z.bam="hola mundo"     # foo.bar.bam se cambia en consecuencia
print ${foo.bar.bam}
hola mundo
```

–x Exportada

Este atributo hace que le nombre y valor de la variable sea pasado al entorno de cada proceso hijo. **ksh** automáticamente fija este atributo para todas las variables heredadas del entorno padre. Por lo tanto, si cambias el valor de cualquier variable que es heredada del entorno, entonces **ksh** automáticamente exporta el nuevo valor al entorno de cada proceso hijo.

Bourne shell: Esto es diferente de la shell de Bourne, donde debes exportar la variable explícitamente para exportar el nuevo valor al entorno de cualquier proceso hijo.

ksh también pasa los atributos de las variables exportadas al entorno de cada proceso hijo.

Los nombres de las variables exportadas no pueden contener un **.** (punto).

Utilice **export** o **typeset –x** para fijar este atributo. Dentro de una función, **typeset** crea una variable local mientras que **export** no lo hace.

Ejemplo:

```
export foo=bar PATH
```

–H Mapeo de pathname del sistema operativo anfitrión

Aplicable solamente a sistemas que no sean UNIX. **ksh** ignora esto si lo especificas en sistemas UNIX.

Cuando quiera que **ksh** expande la variable, **ksh** cambiar el formato del valor de una pathname de un sistema UNIX a un pathname del sistema operativo anfitrión.

Ejemplo:

En este ejemplo se asume que cada nombre de fichero en un pathname está delimitado por un **** en lugar de por un **/** en el sistema operativo anfitrión.

```
typeset –H fichero=/system/win32
print “$fichero”
c:\system\win32
```

–t Etiquetado

ksh no utiliza este atributo. Está pensado con la intención de que lo uses como quieras.

Ejemplo:

```
typeset –t PWD foo=bar
typeset +t     # Visualiza los nombres de todas las variables con
               # el atributo etiquetado
PWD
foo
```

ARRAYS

Puedes usar cualquier variable como un array unidimensional, con el formato *variable[subíndice]*. Utilice la sintaxis de la página xxxx para asignar valores a elementos individuales del array. Existen dos tipos de arrays en **ksh**: arrays indexados y arrays asociativos.

Para los arrays indexados, el *subíndice* debe ser una expresión aritmética que se evalúe a un número en el rango **0-4095**. **Dependencia de la implementación**: Algunas implementaciones de **ksh** podría tener un límite mayor para subíndices enteros.

Versión: El rango de los subíndices estaba limitado a **0-1023** para algunas versiones de la versión de **ksh** de 16/Nov/1988. Vea el programa de la página xxxx para ver como chequear si un subíndice es válido.

Para arrays asociativos, el subíndice puede ser cadena de texto arbitraria. Los arrays asociativos deben ser declarados con **typeset –A**. No tienes que declarar los arrays indexados. Sin embargo, si conocer el tamaño del array indexado, y/o quieres especificar un atributo para el array, utilice **typeset** para declarar el tamaño y o atributos, por ejemplo, **typeset –u x[100]**. Cuando referencias cualquier variable con un subíndice válido, se creará una array si no lo declaraste. Cada atributo se aplica a todos los elementos del array.

La forma de cualquier referencia de elemento de array es **${nombre[subíndice]}** tanto para arrays indexados como para arrays asociativos.

Para referenciar todos los elementos de un array indexado o un array asociativo, utilice el subíndice ***** o @. Los subíndices ***** y @ difieren solamente cuando la expansión está contenida dentro de comillas dobles. En este caso ellos difieren de la misma forma que la expansión de los parámetros especiales ***** y @ difieren (vea la página xxxx)

Para referenciar todos los subíndices de un array, utilice el operador prefijo **!** en la expansión de parámetro. Por ejemplo, **${!foo[@]}** se expande a la lista de subíndices del array para la variable **foo**.

Precaución: Las implementaciones actuales de **ksh** no te permiten exportar variables array a invocaciones de **ksh** separadas.

Versión: Los arrays asociativos y el hecho de referenciar a todos los subíndices con **${!foo[@]}** están solamente disponibles en versiones de **ksh** posteriores a la de 16/Nov/1988.

EXPANSIÓN DE PARÁMETROS – INTRODUCCIÓN

$** dispara la expansión de parámetro. Lea **$ como "*el valor de*".

Cuando decimos que un parámetro no está fijado, queremos decir que nunca se le ha dado un valor al parámetro o que el parámetro ha sido desasignado (unset); por ejemplo, utilizando **unset** o **shift**.

ksh expande los parámetros incluso dentro de las comillas dobles (agrupación). **Nota**: Si pones comillas dobles alrededor de la expansión de parámetro de una palabra comando, o una palabra en la lista de palabras del comando compuesto **for** o el comando compuesto **select**, **ksh** no realiza particionamiento en campos o expansión de pathname sobre el resultado de la expansión.

ksh expande los parámetros dentro de los here-documents si no entrecomillas la palabra delimitadora.

EXPANSIÓN DE PARÁMETROS – BÁSICA

${parámetro}

ksh expande este formato al valor del *parámetro*.

Si la opción **nounset** está activa y el parámetro no está fijado, entonces **ksh** visualiza un mensaje de error. Si esto ocurre dentro de un script, **ksh** termina la ejecución de este script con un valor de retorno Falso. Si **nounset** no está activa, entonces **ksh** trata los parámetros sin fijar como si su valor fuese Nulo.

Nota: Si el valor del parámetro es Nulo, y si tú:

- Pones comillas de agrupación (dobles) alrededor del parámetro, entonces **ksh** hace eliminación de comillas, retiene la expansión de *parámetro* y lo cuenta como un argumento Nulo. **Excepción**: Cuando no existen parámetros posicionales (o elementos de array), **"$@"** (o **"${variable[@]}"**) no cuenta como un Nulo.
- No pones comillas dobles alrededor de ${parámetro}, entonces cuando **ksh** haga la eliminación de las comillas, ignora la expansión de parámetro y no lo cuenta como un argumento.

Las llaves (**{ }**) son opcionales, excepto para un:

- parámetro seguido de una letra, dígito o subrayado que no va a ser interpretado como parte de su nombre
- Variable que está subindexada
- Parámetro posicional de más de un dígito
- Nombres de variables con **.** (punto)

Si *parámetro* es uno o más dígitos, entonces es un parámetro posicional.

Ejemplo:

```
print $PWD ${11} $$
/usr/dgk arg11 1234
```

ksh evalúa el subíndice antes de expandir una variable array. Deberías tener en cuenta esto, en caso de que la evaluación de subíndice tenga efectos laterales.

Ejemplo:

En la expansión **${x[y=1]}**, **ksh** primero evalúa la asignación **y=1** y después expande **x[1].**

EXPANSIÓN DE PARÁMETROS – MODIFICADORES

${parámetro:-palabra} Utilizando valores por defecto

Nota: Los dos puntos son opcionales

Si parámetro es:

- No fijado, **ksh** expande el formato de arriba al valor expandido de *palabra*.
- Nulo, y especificaste **:**, lo mismo que arriba
- Si no, **ksh** expande el formato de arriba al valor de *parámetro*. **ksh** no realiza ninguna expansión sobre *palabra*.

Ejemplo:

En este ejemplo, **ksh** ejecuta **date** solamente si **d** es Nulo o no está fijada.

```
print ${d:-$(date)}
```

En este ejemplo, **ksh** realiza una expansión de tilde si **d** es Nulo o no está fijada.

```
print ${d:-~}
/usr/dgk
```

${parámetro:=palabra} Asignación de valores por defecto

Nota: Los dos puntos son opcionales.

Si parámetro es:

- No fijado, **ksh** asigna el valor expandido de *palabra* a *parámetro*. y entonces expande el formato de más arriba al valor de *parámetro*.
- Nulo, y especificaste **:**, los mismo que arriba

- Si no, **ksh** expande el formato de arriba al valor de *parámetro*. **ksh** no realiza ninguna expansión sobre *palabra*.

  Solamente las variables podrían ser asignadas de esta manera.

Ejemplo:

```
unset X
typeset –u X
print ${X=abc}
ABC
```

${parámetro:?palabra} Visualiza un error si es Nulo o no está fijado

Nota: Los dos puntos son opcionales

Si parámetro es:

- No está fijado, **ksh** expande y visualiza *palabra* en el error estándar, y hace que tu shell script, si existe, finalice con un valor de retorno Falso (1). Si omites *palabra*, entonces **ksh** visualiza un mensaje en el error estándar.
- Nulo, y especificaste **:**, lo mismo que arriba
- Si no, **ksh** expande el formato superior al valor de *parámetro*. **ksh** no realiza ninguna expansión sobre *palabra*.

Ejemplo:

```
print ${foo?}
ksh: foo: parameter null or not set
```

${parámetro:+palabra} Uso de un valor alternativo

Nota: Los dos puntos son opcionales

Si parámetro es:

- No está fijado, **ksh** expande el formato de más arriba a Nulo. **ksh** no expande *palabra*.
- Nulo, y especificaste **:**, lo mismo que arriba
- Si no, **ksh** expande el formato de más arriba al valor expandido de *palabra*.

Ejemplo:

```
set a b c
print ${3+foobar}
foobar
```

EXPANSIÓN DE PARÁMETROS – SUBCADENAS

Nota: patrón representa a cualquier patrón

${parámetro#patrón} Elimina el patrón pequeño de la izquierda

El valor es esta expansión es el valor del *parámetro* con la porción más pequeña que encaje por la izquierda con *patrón* eliminada. Si el *patrón* empieza con un # debe estar entrecomillado.

Ejemplo:

```
cd $HOME/src/cmd
print ${PWD#$HOME/}
src/cmd
```

${parámetro##patrón} Elimina el patrón grande de la izquierda

El valor es esta expansión es el valor del *parámetro* con la porción más grande que encaje por la izquierda con *patrón* eliminada.

Ejemplo:

```
x=/uno/dos/tres
print ${x##*/}
tres
```

${parámetro%patrón} Elimina el patrón pequeño de la derecha

El valor es esta expansión es el valor del *parámetro* con la porción más pequeña que encaje por la derecha con *patrón* eliminada. Si el *patrón* empieza con un **%** debe estar entrecomillado.

Ejemplo:

```
x=fichero.c
print ${x%.c}.o
fichero.o
```

${parámetro%%patrón} Elimina el patrón grande de la derecha

El valor es esta expansión es el valor del *parámetro* con la porción más grande que encaje por la derecha con *patrón* eliminada.

Ejemplo:

```
x=foo/fun/bar
print ${x%%/*}
foo
```

${parámetro:posición:longitud } Subcadena empezando en posición

${parámetro:posición}

Vea la página xxxx para la descripción cuando parámetro es @ o *****. El valor de esta expansión es una subcadena del *parámetro* empezando en la posición de carácter definida por la expresión aritmética *posición*, y terminando después de *longitud* caracteres, donde *longitud* es una expresión aritmética. Si ***:longitud*** no se especifica o si el valor de *longitud* hará que la expansión exceda la longitud de *parámetro*, entonces se producirá la subcadena que finalizará en el último carácter de *parámetro*. El primer carácter de *parámetro* está definido por una posición de **0**.

Ejemplos:

```
x=foo/fun/bar
print ${x:4:3}
fun
print ${x:3}
/fun/bar
```

Versión: Las expansiones de parámetro *${parámetro:posición:longitud}* y *${parámetro:posición}* está disponibles solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

${parámetro/patrón/cadena} Sustituye cadena por patrón

${parámetro/#patrón/cadena}

${parámetro/%patrón/cadena}

${parámetro//patrón/cadena}

El valor de esta expansión es el valor de *parámetro* con una o más ocurrencias del encaje más grande de *patrón* sustituidas por *cadena*. Cualquier **/** contenido dentro del patrón debe ser entrecomillado. Cada ocurrencia de **\dígito** en *cadena* que no es el resultado de una expansión, se sustituye por la correspondiente referencia atrás en *patrón*. Para:

- **/patrón/** La primera ocurrencia de *patrón* en parámetro es sustituida
- **/#patrón/** Si *patrón* ocurre al principio de *parámetro*, entonces es sustituido por *cadena*.
- **/%patrón/** Si *patrón* ocurre al final de *parámetro*, entonces es sustituido por *cadena*.
- **//patrón/** Cada ocurrencia de *patrón* es sustituida por *cadena*.

Si no se se especifica ***/cadena*** se elimina cada porción de texto que encaje.

Ejemplos:

```
x=foo/fun/bar
print ${x/fun/bam}
foo/bam/bar
print ${x//f?/go}
goo/gon/bar
print ${x/%n*r/mble}
foo/fumble
print ${x/*+(o)*/\1}
oo
```

Versión: Estas expansiones de parámetros están disponibles solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

EXPANSIÓN DE PARÁMETROS – OTRAS

${#parámetro} Longitud de la cadena

Si *parámetro* es ***** o @, **ksh** sustituye por el número de parámetros posicionales. Si no, **ksh** sustituye por la longitud del valor de *parámetro*.

Ejemplo:

```
HOME=/usr/dgk
print ${#HOME}
8
```

${#variable[*]} Número de elementos de un array

${#variable[@]}

ksh expande el formato de arriba al número de elementos del array *variable* que están fijados.

Ejemplo:

```
unset x
x[1]=5 x[3]=8 x[6]=abc x[12]=
print ${#x[*]}
4
```

${!variable} Nombre de variable

ksh expande el formato de arriba al nombre de la variable a la cuál *variable* está fijada. En la mayoría de los casos, *variable* se expandirá simplemente a *variable*. Sin embargo, si *variable* es una variable de referencia, este formato se expandirá al nombre de variable que cuál se refiere *variable*.

Ejemplo:

```
nameref foo=bar
print ${!foo}
bar
```

Versión: La expansion de parámetro *${!variable}* está solamente disponible en versiones de **ksh** posteriores al 16/Nov/1988.

${!prefijo@} Nombres de variables que empiezan con prefijo

${!prefijo*}

ksh expande el formato de arriba a la lista de nombres de variables que comienzan con *prefijo* y están definidas.

Ejemplo:

```
print ${!HIST@}
HISTEDI HISTCMD HISTFILE HISTSIZE
```

Versión: Las expansiones de parámetros *${!prefijo@}* y *${!prefijo*}* están disponibles solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

${!variable[@]} Nombres de subíndices de un array

${!variable[*]}

ksh expande el formato de arriba a la lista de subíndices del array *variable* que están definidos. Para una variable que no es un array, el valor es **0** si la variable está definida. Si no, es nulo.

Cuando se usa con comillas dobles, **ksh** expande el formato **${!variable[*]}** a un argumento, y el formato **${!variable[@]}** a argumentos separados para cada subíndice de array.

Ejemplo:

```
x[1]=5 x[3]=8 x[6]=abc x[12]=
print ${!x[@]}
1 3 6 12
typeset –A class
class[Lunes]=algebra class[Jueves]=historia
print ${!class[@]}
Lunes Jueves
```

Versión: Estas expansiones de parámetros están disponibles solamente en versiones de **ksh** posteriores al 16/Nov/1988.

${@:posición:longitud} Expande parámetros posicionales, sub-array

${*:posición:longitud}

${@:posición}

${*:posición}

${variable[@]:posición:longitud}

${variable[*]:posición:longitud}

${variable[@]:posición }

${variable[*]:posición }

El valor de esta expansión es un sub-array de *variable*, o un subconjunto de los parámetros posicionales (ver la sección “Parámetros Posicionales” xxxx justo más abajo), comenzando en la posición definida por la expresión aritmética *posición* y terminando después de *longitud* parámetros, donde *longitud* es una expresión aritmética. Si ***:longitud*** no se especifica o si el valor de *longitud* provoca que la expansión exceda el número de elementos del array o parámetros posicionales, se utilizarán los elementos restantes a partir de *posición*.

Ejemplos:

```
set foo/fun/bar hello.world /dev/null
print ${@:2}
hello.world /dev/null
print ${*:2:4}
hello.world /dev/null
print ${@:0:2}
ksh foo/fun/bar
x=( foo/fun/bar hello.world /dev/null)
print ${x[@]:1:2}
hello.world /dev/null
```

Versión: Estas expansiones de parámetros están solamente disponibles en versiones de **ksh** posteriores al 16/Nov/1988.

PARÁMETROS ESPECIALES FIJADOS POR KSH

Nota: No puedes inicializar o asignar valores a estos parámetros.

@ Parámetros posicionales

ksh expande los parámetros posicionales, comenzando con **$1**, separándolos con caracteres espacio.

Si incluyes esta expansión de parámetro en comillas de agrupación (dobles), entonces **ksh** incluye el valor de cada parámetro posicional como una cadena entrecomillada doble separada. Esto contrasta con la forma en que ***** es manejado, donde **ksh** incluye el valor entero en una única cadena entrecomillada con comillas dobles (la forma usual en la que se expanden los parámetros). Por lo tanto, **“$@”** es equivalente a **“$1” “$2”** hasta **“$n”** donde *n* es el valor de **$#**. Cuando no existen parámetros posicionales, **ksh** expande **“$@”** a una cadena nula sin entrecomillar.

Ejemplo:

```
set “hello there” world
for i in $@
do print “$i"
done
hello
there
world
for i in “$@”
do print “$i"
done
hello there
world
```

* Parámetros posicionales

ksh expande los parámetros posicionales, comenzando con **$1**, separándolos con el primer carácter del valor de la variable **IFS**.

Si incluyes esta expansión de parámetros entre comillas de agrupación (dobles), entonces **ksh** incluye, entre comillas dobles, el valor de todos los parámetros posicionales, separados por *d*, donde *d* es el primer carácter de la variable **IFS**. Por lo tanto, **"$*"** es equivalente a **"$1*d*$2*d*..."**.

Ejemplo:

```
set "hello there" world
IFS=,$IFS
for i in $*
do print "$i"
done
hello
there
world
for i in "$*"
do print "$i"
done
hello there,world
```

Número de Parámetros posicionales

Esto es inicialmente el número de argumentos de **ksh**. Su valor puede ser cambiado por los comandos **set**, **shift**, y **.** (punto), y llamando a una función.

Ejemplo:

```
set a b c
print $#
3
```

- Indicadores de opciones

(menos) Estas son las opciones suministradas a **ksh** en su invocación o a través de **set**. **ksh** también fija automáticamente algunas opciones. Cada opción se corresponde con una letra en el valor de este parámetro.

Ejemplo:

```
case $- in
*i*) print interactivo ;;
*)   print no interactivo ;;
esac
interactivo
```

? Valor de retorno

El valor es el devuelto por el último comando, función o programa ejecutado. Si el valor es:

- Cero, indica una finalización con éxito
- Distinto de cero, indica una condición de error o inusual. Un comando que es terminado por una señal tiene un valor de retorno de 256 más el número de señal.

Versión: La versión de **ksh** de 16/Nov/1988 usada 128 en lugar de 256.

Ejemplo:

```
let 0
print $?
```

1

$ Identificador de proceso de esta Shell

El identificador de proceso es un entero único que garantiza que sean diferentes todos los procesos activos en un determinado momento. Se obtiene normalmente del sistema operativo. Este parámetro se expande al identificador de proceso.

Supón que estás escribiendo un shell script que podría ser usado por dos o más usuarios a la vez, y que el script crea un fichero temporal para su propio uso. Puedes usar este parámetro como parte del pathname, para hacer que cada uso del script genere un nombre distinto para el fichero temporal.

El valor de **$** no cambia cuando **ksh** crea una subshell, incluso en sistemas que crean un proceso separado para una subshell.

Ejemplo:

```
exec 3> /tmp/foo$$
print /tmp/foo$$
( print /tmp/foo$$ )
/tmp/foo1234
/tmp/foo1234
```

! Identificador de proceso background

El número de identificación de proceso del último comando background o co-proceso generado.

Ejemplo:

```
sleep 30 &
print $!
784
```

VARIABLES FIJADAS POR KSH

Si asignas valores a la mayoría de estas variables, eliminarás su significado normal. Por ejemplo, si cambias **PWD**, no cambiará tu directorio de trabajo actual. Además, si pones **print $PWD**, **ksh** no imprimirá tu directorio de trabajo. Sin embargo, la siguiente vez que **ksh** ejecute **cd**, se fijará **PWD** a el pathname de tu directorio de trabajo.

_ Variable temporal

(subrayado) Esta variables tiene varias funciones. Contiene:

- El último argumento del anterior comando simple ejecutado en el entorno actual
- Nombre del fichero **MAIL** que encaja cuando se chequea el correo.
- Valor del pathname de cada programa que **ksh** invoca. Este valor es pasado en el entorno. El valor de _ en un script es inicializado al pathname del script.

Precaución: Si fijas _, **ksh** elimina su significado especial incluso si lo fijas subsecuentemente.

Ejemplo:

```
print hello world
hello world
print $_
```

```
world
```

HISTCMD Número de comando histórico

El valor de **HISTCMD** es el número del comando actual en el fichero histórico.

Versión: La variable **HISTCMD** está disponible solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

LINENO Número de línea actual

ksh fija **LINENO** a el número de línea actual dentro de un script o función antes de ejecutar cada comando.

Cuando le asignas un valor a **LINENO**, solamente afecta a los números de línea de los comandos que **ksh** no ha leído todavía.

Precaución: Si eliminas la definición de **LINENO**, **ksh** elimina su significado especial incluso si lo defines subsecuentemente.

Ejemplo:

```
function foobar
{
        print $0 linea LINENO
}
foobar
foobar linea 2
```

OLDPWD Último directorio de trabajo

Directorio de trabajo anterior fijado por **cd**.

Ejemplo:

```
print $PWD $OLDPWD
/usr/dgk /usr/src
cd –
/usr/src
print $OLDPWD
/usr/dgk
```

OPTARG Argumento opción

ksh fija el valor de la variable **OPTARG** cuando **getopts** encuentra una opción que requiere un argumento.

Precaución: Si eliminas la definición de **OPTARG**, **ksh** elimina su significado especial incluso si lo vuelves a definir a continuación.

OPTIND Índice opción

getopts fija el valor de la variable **OPTIND** al índice del argumento en el que buscar la siguiente opción.

OPTIND es inicializado a **1** cuando se invoca **ksh**, un script o una función. Puedes asignar **1** a **OPTIND** para hacer que **getopts** procese otra lista de argumentos.

Precaución: Si eliminas la definición de **OPTIND**, **ksh** elimina su significado especial incluso aunque lo vueltas a definir a continuación.

Ejemplo:

```
OPTIND=1
while getopts u:xy:z foo –y bar –zx –unew foobar
do print OPTIND=$OPTIND OPTARG=$OPTARG foo=$foo
done
OPTIND=3 OPTARG=bar foo=y
OPTIND=3 OPTARG= foo=z
OPTIND=4 OPTARG= foo=x
OPTIND=5 OPTARG=new foo=u
```

PPID Identificador de proceso del padre

Identificador de proceso del proceso que invocó a esta shell. **Precaución**: Si eliminas la definición de **PPID**, **ksh** elimina su significado especial incluso si lo vuelves a definir posteriormente.

El valor de **PPID** no cambia cuando **ksh** crea una subshell, incluso en sistemas que crean un proceso separado para una subshell.

Ejemplo:

```
print $PPID $(print $PPID)
777 777
```

PWD Directorio de trabajo

Directorio de trabajo fijado por **cd**.

Ejemplo:

```
cd /usr/src
print $PWD
/usr/src
```

RANDOM Generador de números aleatorios

RANDOM tiene el atributo entero. **ksh** le asigna a **RANDOM** enteros aleatorios uniformemente distribuidos desde **0** hasta **32767** cada vez que es referenciado.

Puedes inicializar la secuencia de números aleatorios asignando un valor numérico a **RANDOM**.

Precaución: Si eliminas la definición de **RANDOM**, **ksh** elimina su significado especial incluso si vuelves a definirlo posteriormente.

Ejemplo:

```
RANDOM=$$    # Inicializa el generador de número aleatorios
print $RANDOM $RANDOM
7269 32261
```

REPLY Variable de respuesta

Cuando usas el comando compuesto **select**, los caracteres que teclees son almacenados en esta variable.

Cuando usas el comando built-in **read**, y no le especificas ningún argumento, los caracteres que son leídos se almacenan en esta variable.

Ejemplo:

```
read; print “$REPLY”
hola   mundo
hola  mundo
```

SECONDS Tiempo transcurrido en segundos

Por defecto, **SECONDS** tiene el atributo punto flotante visualizado a tres lugares después del punto decimal. Su valor es el número de segundos desde que invocaste **ksh**. Si asignas un valor a **SECONDS**, entonces el valor de **SECONDS** es el valor que le asignaste, más el número de segundos transcurridos desde que hiciste la asignación. Vea “Personalice su prompt” xxxx

Precaución: Si eliminas la definición de **SECONDS**, **ksh** elimina el significado especial incluso aunque vuelvas a definirlo posteriormente.

Versión: La variable **SECONDS** era un número entero de segundos con la versión de **ksh** de 16/Nov/1988.

Ejemplo:

```
SECONDS=35
print $SECONDS; sleep 10; print $SECONDS
35.069
45.103
```

.sh.edchar Carácter que causó una captura de KEYBD

.sh.edchar contiene el valor del carácter de teclado (o secuencia de caracteres si el primer carácter es Escape) que se introdujo cuando se procesó una captura (Trap) de **KEYBD**. Si el valor se cambia como parte de la acción de captura, entonces el nuevo valor remplaza la tecla (o secuencia) que causó la captura.

.sh.edcol Posición del cursor antes de una captura de KEYBD

.sh.edcol se fija a la posición de carácter actual del cursor dentro del buffer de entrada cuando se introdujo una captura de **KEYBD**.

.sh.edmode Carácter de escape para vi

.sh.demodé se fija al carácter *Escape* cuando en el modo de entrada de **vi** se introduce una captura de **KEYBD**.

.sh.edtext Contenidos del buffer de entrada antes de capturar KEYBD

.sh.edtext se fija a los contenidos actuales del buffer de entrada cuando se introduce una captura de **KEYBD**.

.sh.name Nombre de variable en Función disciplina

.sh.name se fija dentro de una función disciplina al nombre de la variable asociada con la función disciplina.

.sh.subscript Subíndice en Función Disciplina

.sh.subscript recibe el valor dentro de la función disciplina del nombre del subíndice de la variable asociada con la función disciplina.

.sh.value Valor de variable en Función Disciplina

.sh.value se fija, dentro de una función disciplina **set**, al valor que se intenta asignar a la variable asociada con la función disciplina **set**. El valor de **.sh.value** cuando se completa la función disciplina será el valor de la asignación.

Si se le da un valor a **.sh.value** dentro de una función disciplina **set**, se convertirá en el valor de la variable que está siendo expandida.

Ejemplo:

```
unset foo
function foo.set
{
  print “old: ${.sh.name}[ ${.sh.subscript}]=${.sh.value}”
  .sh.value=good
}
foo[3]=bad
old: foo[3]=bad
print “new: foo[3]=${foo[3]}”
new: foo[3]=good
```

Ejemplo:

```
function bar.get
{      .sh.value=$(date)
}
print “$bar”
Sat Aug 28, 2001 15:32 EST
```

Ejemplo:

```
foo=( float x=1.0 y=0 sum=1.0 )
function foo.x.set
{      (( foo.sum = foo.y + .sh.value ))
}
function foo.y.set
{      (( foo.sum = foo.x + .sh.value ))
}
foo.x=3.5  foo.y=5
print “${foo.sum}”
8.5
```

.sh.version Versión de la shell que se está ejecutando

.sh.version se fija a la cadena de versión de la shell actualmente en ejecución.

Ejemplo:

```
print “${.sh.version}”
Version 12/28/93b
```

Versión: Las variables que comienzan por **.sh** están disponibles solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

VARIABLES USADAS POR KSH

Esta sección describe las variables que puedes fijar para influir en el comportamiento de **ksh**. Mucha gente asigna valores a estas variables en sus fichero profile, de forma que no lo tienen que hacer cada vez que se conectan al sistema.

Los administradores de sistema a menudo asignan valores a algunas de estas variables en el fichero **/etc/profile**. Puedes usar **cat** para visualizar el contenido de **/etc/profile**. Sin embargo, normalmente solamente los administradores del sistema podrán hacer cambios en este fichero.

Por defecto: Puedes asignar valores a estas variables. Si no puedes, **ksh** hace algo de lo siguiente:

- Asigna un valor por defecto explícito. En este caso, podrías imprimir el valor de la variable
- Utiliza un valor por defecto implícito. En este caso, no podrías imprimir el valor de la variable
- No tiene valor por defecto. En este caso, **ksh** no hace nada de lo anteriormente especificado, y realmente no usa la variable de ninguna forma. Por ejemplo, si no asignas un valor a **CDPATH**, entonces si especificas **cd**, no haría realmente una búsqueda.

CDPATH Path o camino de búsqueda para el Built-in cd

Lista de directorios separados por **:** (dos puntos) utilizada por el comando **cd** como se ha describe más abajo. El directorio de trabajo se especifica mediante **.** (punto) o un nombre de directorio Nulo, el cuál puede aparecer antes del primer **:**, después del último **:**, o entre delimitadores **:**.

Si el directorio que especificas para **cd** no comienza con **/**, **./** o **../**, entonces **ksh** busca cada uno de los directorios de **CDPATH** (en el orden que aparezcan) para encontrar el directorio especificado, e intenta realizar el **cd** (cambio de directorio) a dicho directorio.

No tiene valor por defecto

Ejemplo:

```
CDPATH=$HOME:/usr/src
cd cmd
/usr/src/cmd
```

COLUMNS Número de columnas en el Terminal

Si fijas **COLUMNS**, **ksh** utiliza el valor para definir el ancho de la ventana de edición para el modo de edición de **ksh**, y para imprimir las listas **select**.

Junto a **ksh**, algunos otros programas también utilizan esta variable.

Valor implícito por defecto: 80

EDITOR Pathname de tu editor

Si fijas el valor de **EDITOR** a un pathname que finaliza en **emacs**, **gmacs**, o **vi** y la variable **VISUAL** no está fijada, entonces **ksh** activa la opción correspondiente.

Valor implícito por defecto: **/bin/ed**

ENV Fichero de entorno de usuario

Cada vez que invocas a **ksh** de forma interactiva, él expande esta variable y realiza expansión de parámetros, sustitución de comandos, y expansión aritmética para generar el pathname del shell script, si existe alguno, que será ejecutado cuando se invoque **ksh**. Los usos

típicos son para definiciones de **alias** y **funciones** y para ajuste de opciones con **set**. **Versión**: **ENV** era también expandido y ejecutado para shells no interactivos con la versión de **ksh** de 16/Nov/1988.

Cuando la opción **privileged** está activa, **ksh** no expande esta variable y no ejecuta el script resultante.

No presenta valor por defecto

Ejemplo:

```
ENV=$HOME/envfile
```

FCEDIT Editor para el Built-in hist

Asignas un valor a **FCEDIT** si quieres cambiar el editor que **hist** utilizará, cuando no especificas un editor en la línea de comando de **hist**.

Valor por defecto implícito: /bin/ed

Precaución: La variable **FCEDIT** está obsoleta. Es preferible la variable **HISTEDIT**.

FIGNORE Ignora nombres de ficheros que encajen con este patrón

ksh ignora cada nombre que encaja con el patrón definido por el valor de **FIGNORE**, cuando se lee un directorio durante la expansión de fichero.

No tiene valor por defecto

Versión: La variable **FIGNORE** está disponible solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Ejemplo:

```
print /usr/*/include/*      # Visualiza una lista de fichero en todos los
                            # directorios include.
/usr/local/include/sys  /usr/gcc/include/sys
FIGNORE=gcc                 # Patrón que encaja con los ficheros fuente de C
print /usr/*/include/*
/usr/local/include/sys
```

FPATH Camino de búsqueda para Funciones de auto carga

Lista de directorios separados por **:** (dos puntos) que **ksh** busca para encontrar un fichero de definición de función. El formato de la variable **FPATH** es el mismo del de la variable **PATH**.

No presenta valor por defecto

Ejemplo:

```
FPATH=$HOME/fundir      # Busca las definiciones de funciones aquí
autoload foobar         # Especificación de que foobar es una función
foobar                  # Carga $HOME/fundir/foobar para definir la función
                        # y la ejecuta.
```

HISTEDIT Editor para el Built-in hist

Asignas un valor a **HISTEDIT** si quieres cambiar el editor que utilizará **hist**, cuando no especificas un editor en la línea de comando de **hist**.

Valor por defecto implícito: /bin/ed

Versión: La variable **HISTEDIT** está disponible solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988. En versiones anteriores, la variable se llama **FCEDIT** y el comando **hist** se llama **fc**.

HISTFILE Pathname del histórico

Si **HISTFILE** está fijado cuando **ksh** accede por primera vez al fichero histórico, entonces **ksh** utiliza este valor como el nombre del fichero histórico. **ksh** realiza el primer acceso al fichero histórico cuando encuentra la primera definición de función que no tiene la opción **nolog** activada, o después de completar el procesamiento del fichero de entorno, lo que ocurra primero.

Si el fichero histórico no existe y **ksh** no puede crearlo, o si existe pero **ksh** no tiene permiso para añadir, entonces **ksh** utiliza un fichero temporal como fichero histórico.

Valor por defecto implícito: $HOME/.sh_history

Ejemplo:

Este ejemplo muestra como crear un fichero histórico separado para cada "ventana" en sistemas que tiene ventanas (algunas veces llamadas "capas"), y que también tienen un programa llamado **tty** que devuelve un pathname separado para cada ventana.

```
nombretty=$(tty)
HISTFILE=$HOME/${nombretty##/}.hist
```

HISTSIZE Número de comandos históricos

Si **HISTSIZE** está fijado cuando **ksh** accede por primera vez al fichero histórico, entonces el máximo número de comando introducidos previamente a los que puedes acceder a través de **ksh** será igual a este número.

Precaución: Fijando este valor a un valor extrañamente grande, tal y como 10000, podría llevar a un inicio lento en una nueva invocación de **ksh**.

Valor por defecto implícito: 128

HOME Tu directorio de inicio en el sistema

El valor de la variable **HOME** es el argumento por defecto usado por **cd**.

Por defecto: El valor de **HOME** se fija automáticamente cuando te conectas al sistema, según fue asignado por tu administrador del sistema.

IFS Separador de campos interno

ksh utiliza cada uno de los caracteres en el valor de la variable **IFS** para partir en campos:

- El resultado de una sustitución de comando o expansión de parámetros de palabras comando
- El resultado de una sustitución de comando o expansión de parámetros de palabras después de **in** con **for** y **select**.
- Los caracteres que **ksh** lee con el comando **read**.

ksh utiliza el primer carácter del valor de **IFS** para separar argumentos cuando **ksh** expande el parámetro **, y cuando ksh expande ${variable[*]}**.

Cualquier secuencia de caracteres que pertenezcan a la clase **espacio** que esté contenida en **IFS** delimita un campo, excepto para cualquier carácter que aparezca en **IFS** más de una vez consecutiva. Si no, cada ocurrencia de un carácter **IFS** y caracteres **espacio** adyacentes contenidos en **IFS**, delimitan un campo. Si un carácter **espacio** se repite de forma consecutiva en **IFS**, el carácter se comporta como si no fuese un carácter **espacio** de forma que cada ocurrencia delimita un campo. Si **IFS** se fija a nulo, **ksh** no realiza particionamiento en campos. **Versión**: La repetición de caracteres **espacio** para hacer que cada instancia se comporte como un delimitador está disponible solamente en las versiones de **ksh** posteriores a la versión del 16/Nov/1988.

El valor de **IFS** es vuelto a fijar al valor por defecto después de ejecutar el fichero de entorno y antes de que comience la ejecución de un script. Por lo tanto, no puedes influir en el comportamiento de un script exportando la variable **IFS**.

Valor por defecto: Espacio-Tabulador-Salto de línea, en ese orden

Ejemplos:

```
x=$'foo\t\tbar'
IFS=$'\t'
set -- $x
print second arg=$2
second arg=bar
IFS=$'\t\t'
set -- $x
print second arg=$2
second arg=
IFS=:
read name passwd  uid gid rest
root::0:0:superuser:
print "Usuario $name tiene userid $uid y groupid $gid"
Usuario root tiene userid 0 y groupid 0
```

LANG Lenguaje del Locale

Determina la categoría de locale para aquellas categorías que no se haya seleccionado específicamente a través de una variable que comience por **LC**_.

Valor por defecto implícito: POSIX

LC_ALL Ajuste del locale

Sobrescribe el valor de la variable **LANG** y cualquier otra variable que comience por **LC**_.

No presenta valor por defecto

LC_COLLATE Comparación en el locale

Determina la categoría de locale para la ordenación y clases de caracteres.

No presenta valor por defecto.

LC_CTYPE Clases de caracteres en el locale

Determina la categoría del locale para la funciones de manejo de caracteres tales como la clasificación de caracteres.

No presenta valor por defecto.

LC_NUMERIC Formato numérico del locale

Determina la categoría del locale para el carácter del punto decimal.

No presenta valor por defecto.

LINES Número de líneas del terminal

Si fijas la variable **LINES**, **ksh** utiliza el valor para la impresión de las listas de **select** (comando compuesto **select**). Las listas de **select** se visualizarán verticalmente hasta que se rellenan alrededor de las dos terceras partes de **LINES** líneas.

Además de **ksh**, otros programas también utilizan esta variable.

Valor implícito por defecto: 24

MAIL Nombre de tu fichero de correo

Si a **MAIL** se le asigna el nombre de un fichero que ha crecido, y la variable **MAILPATH** no está definida, entonces **ksh** te informa cuando se produce un cambio en el momento de modificación del fichero cuyo nombre es el mismo que el valor de **MAIL**. **ksh** realiza esto periódicamente según está determinado por la variable **MAILCHECK** (ver más abajo).

MAILCHECK Frecuencia para chequear el correo

MAILCHECK presenta el atributo entero. Puedes fijar el valor de **MAILCHECK** para especificar cómo de a menudo (en segundos) **ksh** chequeará los cambios en la hora de modificación de cualquiera de los ficheros especificados por el valor de **MAIL** o **MAILPATH** (ver más abajo). Cuando ha transcurrido el tiempo, **ksh** realizará el chequeo antes de visualizar el prompt siguiente.

Si **MAILCHECK** no está fijada o es cero, entonces **ksh** chequea el fichero antes de mostrar cada prompt.

Precaución: Si quitas la definición de **MAILCHECK**, **ksh** elimina su significado especial incluso aunque posteriormente le vuelvas a asignar un valor.

Valor por defecto: 600 segundos.

MAILPATH Lista de ficheros de correo

Lista de pathnames separados por dos puntos (**:**). Puedes poner a continuación de cada pathname un **?** y un mensaje para que sea visualizado por **ksh**. El mensaje por defecto es **You have mail in $_**.

Antes de que **ksh** visualice un mensaje, **ksh** fija la variable _ (subrayado) al nombre del fichero que cambió y realiza una expansión de parámetros en el mensaje.

No presenta un valor por defecto.

Ejemplo:

```
MAILPATH=~uucp/dgk:/usr/spool/mail/dgk
```

PATH Camino de búsqueda de los comandos

Lista de directorios separados por dos puntos (**:**). El directorio de trabajo se especifica mediante un punto (**.**) o un nombre de directorio nulo, el cual puede aparecer antes de los primeros **:**, después de los últimos **:** o entre **:** delimitadores.

Utilice la variable **PATH** para especificar donde debería buscar **ksh** el comando que quieres ejecutar. La búsqueda se aplica solamente a programas que no contienen una **/** en su nombre. **ksh** busca el programa en cada uno de los directorios especificados en **PATH** en el orden especificado en dicha variable, hasta que encuentre el programa a ejecutar.

Cuando asignas un valor a la variable **PATH**, **ksh** elimina la definición de los valores de los alias tracked.

No podrías cambiar el valor de **PATH** si la opción **restricted** estuviese activa.

Valor por defecto dependiente de la implementación: **/bin:/usr/bin**

El path en el siguiente ejemplo hace que **ksh** busque primero en tu directorio bin, después en **/bin**, después en **/usr/bin** y finalmente en el directorio de trabajo actual.

Ejemplo:

```
PATH=$HOME/bin:/bin:/usr/bin:
```

PS1 Cadena del prompt primario

Si la opción **interactive** está activada, **ksh** realiza expansión de parámetro, sustitución de comando y sustitución aritmética sobre el valor de **PS1** y lo visualiza a través del error estándar cuando **ksh** está preparado para leer un comando.

ksh sustituye el carácter **!** en **PS1** por el número de comando. Si quieres incluir el carácter **!** en tu prompt deberías utilizar **!!**.

Para más información y ejemplos, como por ejemplo como personalizar tu prompt, vea Personalizando tu Prompt xxxx.

Valor por defecto: $Espacio (#Espacio para el súper usuario)

PS2 Cadena del prompt secundario

ksh visualiza **PS2** a través del error estándar después de que has pulsado el RETURN y esto comienza una nueva línea, sin que hayas acabado de introducir el comando.

Valor por defecto: >Espacio

Ejemplo:

```
$ print Tod\
> ay is Tuesday
Today is Tuesday
```

PS3 Cadena del prompt select

ksh realiza expansión de parámetros, sustitución de comando y sustitución aritmética sobre el valor de **PS3** y lo visualiza en el error estándar para solicitarte que elijas una de las opciones que especificaste con el comando compuesto **select**.

Valor por defecto: **#?**

Ejemplo:

```
PS3='Por favor introduzca un número: '
select i in foo bar1 bar2 bar3
do  command
done
1) foo
2) bar1
3) bar2
4) bar3
Por favor introduzca un número:
```

PS4 Cadena del prompt de depuración

ksh realiza expansión de parámetros, sustitución de comandos y sustitución aritmética sobre el valor de **PS4**, y lo visualiza en el error estándar cuando **ksh** está preparado para visualizar un comando durante las trazas de su ejecución.

Valor por defecto: +

Ejemplo:

```
PS4='[$LINENO]+ '
set -x
print $HOME
[3]+ print /usr/dgk
/usr/dgk
```

SHELL Pathname de la Shell

Si el último componente del valor de **SHELL** es **rsh**, **rksh**, o **krsh** cuando se invoca **ksh**, entonces **ksh** fija la opción **restricted**. Algunos comandos del sistema UNIX utilizan la variable **SHELL** para invocar una nueva shell.

Por defecto: Podría estar fijada al pathname de **ksh** durante la conexión al sistema o login.

TERM Tipo de Terminal

Especifica que terminal estas usando. No es usada por **ksh**. Sin embargo, se usa en algunos ejemplos de este manual y algunos otros programas utilizan esta variable.

No presenta valor por defecto

TMOUT Variable Timeout (tiempo agotado)

TMOUT tiene el atributo entero.

Si fijas **TMOUT** a un valor mayor que cero, **ksh** termina si no introduces un comando dentro del número de segundos prescritos después de que **ksh** muestre el prompt **PS1** (más un periodo de gracia adicional de 60 segundos).

Dependiente de la implementación: **ksh** podría haber sido compilado en tu sistema con un límite máximo para este valor que tú no puedes exceder.

Precaución: Si eliminas la definición de **TMOUT**, **ksh** elimina su significado especial incluso aunque vuelvas a darle un valor posteriormente. Sin embargo, el valor de **TMOUT** en el momento en que eliminas la definición continúa siendo válido para que finalice **ksh**.

Versión: La variable **TMOUT** se usa como tiempo límite para introducir una elección en el comando **select** en las versiones de **ksh** posteriores a 16/Nov/1988.

Valor por defecto: Cero, lo que significa ilimitado. O fijado por el administrador del sistema.

Ejemplo:

```
# En el ejemplo, se asume que han pasado 5 minutos desde que apareció el prompt $
TMOUT=300
$
shell timeout in 60 seconds
```

VISUAL Editor Visual

Si fijas el valor de **VISUAL** a un pathname que finaliza en **emacs**, **gmacs**, o **vi**, entonces **ksh** activa la opción correspondiente sin tener en cuenta el valor de la variable **EDITOR**.

No presenta valor por defecto.

11. COMANDOS BUILT-IN

INTRODUCCIÓN

Comandos Built-in

Los comandos built-in (nos referimos a ellos como "*built-ins*") son procesados por el propio **ksh**. En la mayoría de los sistemas **ksh** hace que sea creado un proceso separado para los programas pero no para los built-ins. Las mayoría de los built-ins se comportan del mismo modo que lo hacen los programas. Sin embargo algunos built-ins defieren en la forma en que funciona la redirección de la entrada/salida y la asignación de variable. Estos built-ins especiales son denotados con un símbolo ( ) al lado de los formato para el comando.

La razón para tener built-ins:

- Puedes cambiar el entorno actual con ellos, directamente o a través de sus efectos laterales
- **ksh** los ejecuta mucho más rápido que otros comandos que hacen lo mismo
- Siempre se comportan tal y como se documenta en este capítulo. El comportamiento de programas podría diferir en sistemas distintos.

Valores de retorno

El valor de retorno normal se especifica en la descripción de cada uno de los built-ins. El valor de retorno es Falso (1) para todos aquellos built-ins para los que has especificado:

- Una opción inválida
- Un número de argumentos incorrecto
- Un argumento incorrecto (por ejemplo, un argumento que debería ser numérico pero que no se lo pones numérico)
- Redirección de la entrada/salida inválida
- Asignación de variable inválida, tales como un identificador inválido o una asignación no numérica a una variable entera
- Nombre de alias inválido
- Expansión de un parámetro que no ha sido fijado y la opción **nounset** está activa.

Salida

A menos que se especifique otra cosa, **ksh** escribe la salida de un comando built-in en la salida estándar, descriptor de fichero 1. Por lo que puedes utilizar los built-ins en pipelines.

ksh escribe los mensajes de error en el error estándar.

Ejemplo:

```
set | wc # set es un comando buil-in
      83      92    7354
```

Precaución sobre efectos laterales:

- **ksh** ejecuta cada elemento de un pipeline, a excepción del último, en un entorno de subshell. Por lo tanto, cualquier efecto lateral de los built-ins, a excepción del último elemento del pipeline, no afecta al entorno actual como hacen normalmente los built-ins.
- Debes encerrar el comando entre paréntesis, si necesitas garantizar que el built-in se ejecute en un entorno de subshell, como se ilustra en el segundo ejemplo de más abajo.
- **Ejemplo**:

```
word=murrayhill
print foobar | read word
print $word
foobar
```

- **Ejemplo**:

```
word=murrayhill
print foobar | (read word)
print $word
murrayhill
```

Notación

Cada comando built-in descrito más abajo utiliza la notación de sintaxis de comando especificada en el capítulo segundo. Cuando se especifica que un comando permite las opciones de la forma –letra, puedes especificar cada una de estas opciones separadamente, precediendo cada una con un –. Puedes especificar – – para hacer que el siguiente argumento sea procesado como un argumento en lugar de como una opción. El – – es requerido cuando el primer argumento que no sea opción comienza con un –. Cuando se especifica que un comando permite las opciones de la forma ±letra, entonces el + se procesa de la misma forma que el –.

Este signo designa built-ins especiales que son tratados de forma diferente tal y como sigue:

- **ksh** procesa las listas de asignación de variables especificadas con el comando antes de la redirección de la entrada/salida. Estas asignaciones permanece su efecto cuando el comando finaliza.
- Los errores en estos built-ins hacen que el script que los contienen finalice.

DECLARACIONES

ksh expande los argumentos del comando a comandos de declaración que están en el formato de una asignación de variable de un modo especial. **ksh** realiza la expansión de la tilde (~) después del signo igual, y no realiza particionamiento en campos o expansión del pathname en estos argumentos.

Ejemplo:

```
bar='uno dos'
export foo=~morris/$bar
print –r -- "$foo"
/usr/morris/uno dos
```

alias [-pt] [nombre[=valor]…]

Versión: La versión de **ksh** de 16/Nov/1988 soportaba una opción **–x** que exportaba las definiciones de alias a scripts ejecutados por **ksh**. La opción **–x** no ha vuelto a tener efecto y no aparece aquí.

Utilice **alias** para definir y visualizar aliases o sinónimos.

No especifique ningún argumento nombre si quieres visualizar los aliases. **ksh** visualiza la lista de aliases, uno por línea, en la salida estándar, en la forma *nombre*=*valor*. La opción **–p** hace que la palabra **alias** preceda a cada línea de forma que se visualice en un formato reutilizable. Si especificas **–t**, entonces **ksh** visualiza solamente los alias tracked. **Versión**: La opción **–p** está disponible solamente en versiones de **ksh** posteriores a 16/Nov/1988

Utilice **–t** para fijar y listar los aliases tracked. El valor de un alias tracked es el pathname competo correspondiente al programa el *nombre* dado. Los aliases tracked se quedan indefinidos cuando el valor de la variable **PATH** es redefinido.

Si se especifica *nombre*, debe ser un nombre de alias válido, o **alias** visualiza un mensaje de error. En un nombre de alias válido, el primer carácter es un carácter imprimible en lugar de ser uno de los caracteres especiales que aparecen en xxxx, y los otros caracteres son alfanuméricos.

ksh define un alias para cada *nombre* cuyo *valor* especifiques. Las definiciones previas para cada *nombre* son eliminadas.

Si especificas solamente *nombre*, entonces:

- Sin **–t**, **ksh** visualiza el nombre y valor del alias *nombre*.
- Con **–t**, **ksh** fija el atributo tracked, y fija el valor del alias *nombre* al pathname que se obtiene de realizar una búsqueda de path.

valor puede contener cualquier texto de shell válido. Durante la sustitución de alias, si el último carácter de *valor* es un Espacio o Tabulador, **ksh** también chequea la palabra que sigue al alias para ver si debería hacer una sustitución de alias. Utilice un Espacio o Tabulador final cuando el siguiente argumento se supone que es un nombre de comando.

Encierre *valor* entre comillas simples si quieres que *valor* se expanda solamente cuando **ksh** ejecute una referencia al alias. De otro modo, **ksh** también expande *valor* cuando procesa **alias**.

Ejemplos:

```
x=1
alias foo='print $x' bar="print $x"
x=2
foo
2
bar
1
alias -t date
alias -t
date=/bin/date
```

Valor de retorno:

- Si todos los nombre(s) son alias o si especificas atributos: Verdadero
- Si no: Falso. Valor es el número de nombre(s) que no son alias.

Ejemplos:

```
alias ls='ls -C'
alias nohup='nohup  '
```

export [-p] [nombre[=valor]]...

Los nombres son marcados para una exportación automática al entorno de los comandos ejecutados subsecuentemente.

export es los mismo que **typeset –x**, excepto que si utilizas **export** dentro de una función, **ksh** no crea una variable local.

Si no le suministras ningún argumento a **export**, **ksh** visualiza una lista de variables con el atributo export, y sus valores. Cada *nombre* empieza en una línea separada.

Si especificas **–p**, **ksh** visualiza la lista de variables en un formato reutilizable. **Versión**: La opción **–p** está disponibles solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Valor de retorno: Verdadero

Ejemplos:

```
export PWD HOME     # Exporta las variables PWD y HOME
export PATH=/local/bin:$PATH
                    # Fija y exporta la variable PATH
export              # Lista todas las variables exportadas
```

readonly [-p] [nombre[=valor]]...

A *nombre(s)* se les da el atributo readonly.

- No puedes cambiar *nombre* en una asignación de variable posterior en este entorno de **ksh**.
- Solamente el propio **ksh** puede todavía asignarle un nuevo valor a una variable readonly. Por ejemplo, si a la variable **PWD** se le asigna el atributo **readonly**, no le puedes asignar a la variable un nuevo valor. Pero **ksh** le asignará un nuevo valor cuando cambies tu directorio de trabajo. (ver **cd**).
- No puedes eliminar la definición (unset) de las variables readonly

readonly es lo mismo que **typeset –r**, excepto que si usas **readonly** dentro de una función, **ksh** no crea una variable local.

Si no especificas ningún *nombre(s)*, **ksh** visualiza una lista de tus variables que tiene el atributo readonly, y sus valores.

Si especificas **–p**, **ksh** visualiza una lista de variables en un formato reutilizable. **Versión**: La opción **–p** está solamente disponible en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Valor de retorno: Verdadero

Ejemplo:

```
readonly HOME PWD
```

typeset ±f[tu] [nombre...]

Versión: La versión de **ksh** de 16/Nov/1988 soportaba una opción **–x** que exportaba las definiciones de funciones a scripts ejecutados por **ksh**. La opción **–x** no ha sido soportada posteriormente y no se describe aquí.

Utilice esta forma de **typeset** para visualizar nombres de funciones y valores, y para fijar y eliminar atributos de funciones.

Utilice las opciones o flags:

t Para especificar la opción **xtrace** para la función(es) especificada mediante *nombre*.

u Para especificar que *nombre* se refiere a una función que no ha sido definida todavía.

Para fijar atributos, especifique **–f** y una o más de las opciones de arriba y los *nombre(s)* a las cuales se aplican.

Para eliminar atributos, especifique **+f** y una o más de las opciones de arriba y los *nombre(s)* a las cuales se aplican.

Utilice **typeset** para visualizar los nombres y definiciones de función en la salida estándar. Utilice **–f** para visualizar ambos, nombres y definiciones de función. Utilice **+f** para visualizar los nombres de función solamente. Porque **ksh** almacena las definiciones de función en tu fichero histórico, incluso si especificas **–f**, **ksh** no visualizará una definición de función si no tienes un fichero histórico, o si la opción **nolog** estaba activa cuando se leyó la función. Para visualizar:

- Funciones específicas, especifique *nombre* y ninguna de las opciones de arriba
- Todas las funciones con un atributo determinado, especifique una opción y ningún *nombre*.
- Todas las funciones, no especifique ni *nombre(s)* ni opciones.

Valor de retorno:

- Si todos los *nombre(s)* son funciones o especificas **–u**: Verdadero
- Si no: Falso. Valor es el número de *nombre(s)* que no son funciones.

Ejemplos:

```
typeset –f   # Visualiza todos los nombres de función, y sus valores si se conocen
typeset –fu foobar  # Especifica que foobar sea una función indefinida
```

typeset [±Ahlnprtux] [±ELFRZi[n]] [nombre[=valor]]...

Utilice este comando sobre variables para:

- Fijar atributos. Especifique un – y los *nombre(s)*.
- Eliminar atributos. Especifique un + y los *nombre(s)*.
- Fijar valor(es). Especifique *nombre(s)* y *valor(es)*.

  – Para fijar atributo(s) después de fijar *valor(es)*

  + Para eliminar atributo(s) después de fijar *valor(es)*
- Visualizar en la salida estándar, variables y/o sus atributos. No especifique ningún *nombre(s)*, y especifique:

  – Para visualizar nombres y valores de todas las variables que tienen el atributo(s) que especificas

  + Para visualizar solamente los nombres

  Nada (ni + ni –) para visualizar nombres y atributos de todas las variables.

La opción **–p** visualiza los nombres de variables, atributos y valores en un formato reutilizable.

Si especificas **typeset** dentro de una función definida con la sintaxis de palabra reservada **function**, **ksh** crea una nueva instancia de cada variable *nombre* si *nombre* es un identificador. Es decir, **ksh** crea variables locales, y restaura los valores y atributos de estas variables cuando la función finaliza.

Puedes especificar los siguientes atributos:

–u Mayúsculas

–l Minúsculas

–i Entero. *n* especifica la base aritmética

–n Referencia de nombre

–A Array asociativo

–E Número exponencial. *n* especifica los dígitos significativos

–F Número en punto flotante. *n* especifica los lugares decimales

–L Justificado a la izquierda. *n* especifica el ancho del campo

–LZ Justificado a la izquierda y elimina los ceros iniciales. *n* especifica el ancho de campo

–R Justificado a la derecha. *n* especifica el ancho de campo

–RZ Justificado a la derecha. *n* especifica el ancho de campo y lo rellena con ceros por la izquierda

–Z Relleno con ceros. *n* especifica el ancho de campo. Equivalente a **–RZ**

–r Readonly

–x Export

–H Mapeo de pathname de sistema operativo sistema-a-host de UNIX

–t Etiqueta definida por el usuario

Nota: Algunos de estos atributos, tales como **–u** y **–E**, son mutuamente exclusivos y no pueden ser utilizados juntos.

Versión: Las opciones **–A**, **–E**, **–F**, **–n** y **–p** está disponibles solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Valor de retorno: Verdadero.

Ejemplos:

```
typeset –p          # Visualiza los nombres y atributos de todas las variables
typeset             # Lo mismo que typeset –p
typset –xi          # Visualiza los nombres y valores de todas las variables con
                    # los dos atributos entero y export
typeset +xi         # Igual que arriba, pero visualiza solamente los nombres
typeset a b c       # Define las variables a, b, c sin ningún atributo especial. Si
                    # se ejecuta dentro de una función, esto crea variables locales
typeset –i8 x       # x será una variable entero y se imprimirá en octal (base 8)
typeset –u x        # Cuando se le de a x un valor, todas las letras en minúsculas
                    # serán convertidas a mayúsculas
...
```

unalias [-a] nombre ...

unalias elimina *nombre(s)* de la lista de alias.

Si especificas **–a**, entonces todos los alias son eliminados de la memoria.

Versión: La opción **–a** está disponible solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Valor de retorno:

- Si todos los *nombre(s)* son alias: Verdadero
- Si no: Falso. Valor es el número de *nombre(s)* que no son alias.

Ejemplo:

```
unalias ls nohup    # Elimina las definiciones de alias ls y nohup
```

unset [-fnv] nombre ...

Si especificas **–v** o no especificas **–f** o **–n**, entonces *nombre* se refiere a una variable. En este caso, para cada uno de los *nombre(s)*:

- **ksh** elimina los valor(es) y atributo(s)
- Un nombre de array sin subíndice se refiere a todos los elementos del array
- No puedes eliminar las variables readonly.

Si especificas **–n** y *nombre* es una referencia de nombre a otra variable, se eliminará la definición de *nombre* en lugar de la variable a la que se refiere. Si no, **-n** es equivalente a **–v**.

Si especificas **–f**, entonces *nombre* se refiere a una función. En este caso, **ksh** elimina la definición de la función y elimina *nombre*. La opción **–f** anula al resto de opciones.

Precaución: Si especificas **unset** con las variables predefinidas siguientes, **ksh** elimina su significado especial incluso aunque las definas (set) posteriormente:

_ (temporalmente) **LINENO MAILCHECK OPTARG OPTIND RANDOM SECONDS TMOUT**

Versión: Las opciones **–n** y **–v** están disponibles solamente en versiones de **ksh** posteriores a la versión de 16/Nov/1988.

Valor de retorno:

- Si todos los *nombre(s)* son funciones o variables: Verdadero
- Si no: Falso. Valor es el número de *nombre(s)* que no son funciones o variables.

Ejemplos:

```
unset VISUAL        # Elimina la definición de la variable VISUAL
unset –f foo bar    # Elimina la definición de las funciones foo y bar
```

PARÁMETROS POSICIONALES Y OPCIONES

set [±Cabefhkmnopstuvx-] [±o opción]... [±A nombre] [argumento...]

shift [n]

FLUJO DE CONTROL

Comando Punto (dot command)

break

command

continue

eval

exit

return

trap

ENTRADA/SALIDA

echo

exec

print

printf

read

SISTEMA OPERATIVO – ENTORNO

cd

pwd

times

ulimit

umask

SISTEMA OPERATIVO – CONTROL DE JOBS O TRABAJOS

bg

fg

disown

jobs

kill

wait

MISCELÁNEA

Comando Null (nulo)

builtin

false

getconf

getopts

hist

let

newgrp

sleep

test

Comando de Corchete izquierdo

true

whence

12. OTROS COMANDOS

COMANDOS

cat

chmod

cp

cut

date

ed

find

grep

ln

lp

lpr

ls

mail

mkdir

more

mv

paste

pg

rm

rmdir

sort

stty

tail

tee

tr

tty

uniq

wc

what

who

13. INVOCACIÓN Y ENTORNO

Este capítulo especifica que realiza **ksh** cuando invocas a **ksh**, scripts **ksh**, y funciones **ksh**. También discute qué es lo que constituye un entorno para un proceso **ksh**, y como el entorno **ksh** es compartido o heredado a través de diferentes tipos de invocaciones.

ENTORNO

El entorno de un script o función de **ksh** está definido por:

- Ficheros abiertos
- Permisos de acceso a ficheros y procesos
- Directorio de trabajo
- Valor de la máscara de creación de ficheros
- Recursos que puedes fijar con **ulimit**
- Traps vigentes
- Variables y atributos de la shell
- Aliases
- Funciones
- Ajustes de opciones

INVOCACIÓN DEL SHELL

ksh crea un nuevo proceso cada vez que invocas a **ksh** explícitamente, con o sin argumentos. Puedes también hacer que **ksh** se invoque implícitamente.

Los elementos de más abajo aparecen en el orden que **ksh** los establece en el entorno:

Herencia

Un proceso hijo de **ksh** hereda una copia del entorno de su proceso padre el cuál incluye:

- Todos los ficheros abiertos que no tengan fijados el close-on-exec.
- Derechos de acceso a ficheros y procesos. **ksh** fija el identificador de usuario efectivo del proceso al identificador de usuario real del proceso, y el identificador de grupo efectivo del proceso al identificador de grupo real del proceso a menos que lo hayas invocado con la opción **privileged (-p)** .
- El directorio de trabajo
- Valor de la máscara de creación de ficheros
- Valores de los límites de recursos
- Señales que son ignoradas por el padre. **ksh** no te permite fijar traps para estas señales.
- Señales que no son ignoradas por el proceso padre. **ksh** fija las acciones por defecto para estas señales, pero pueden ser cambiadas.

- Variables de la shell, con la excepción de **IFS**, con el atributo export en el proceso padre, son heredadas por los nuevos proceso **ksh**. Otras variables de la shell no son heredadas. Los atributos de las variables son también heredados.
- Las definiciones de alias y de funciones no son heredadas. Puedes poner definiciones de alias en el fichero de entorno para hacer que los procesos hijos interactivos tengan acceso a ellos. Las definiciones de funciones pueden ser puestas en un directorio definido por **FPATH** para hacerlas accesibles a los procesos hijos.
- Los ajustes de opciones no son heredados directamente. Sin embargo, los ajustes de opciones se pueden especificar en la línea de comando, y algunos ajustes de opciones pueden ser heredados indirectamente a través del valor de las variables exportadas.

Parámetros posicionales

ksh fija los parámetros posicionales a los valores de los argumentos que no son opciones que tu proporcionas.

Si no especificas ningún argumento que no sea opción a **ksh**, **ksh** fija el parámetro **0** al nombre de **ksh**.

Si no, **ksh** fija el parámetro **0** al primer argumento que no es opción, y el resto de los argumentos se convierten en los parámetros **1**, **2**, **3**, etc.

Ejemplo:

```
ksh –x –v prog abc
# Ejecuta ksh con las opciones x y v
# $0 toma el valor prog
# $1 toma el valor abc
```

Opciones de la línea de invocación

Puedes especificar todas las opciones al comando **set**. Puedes también especificar las siguientes opciones en la invocación:

–c cadena — **ksh** lee comandos de la *cadena*.

–i — **ksh** ejecuta un shell interactivo. En este caso, las señales **QUIT** y **TERM** son ignoradas y la señal **INT** es atrapada y hace que el comando actual termine y se muestre un nuevo prompt.

–r — **ksh** ejecuta un shell restringido

–s — **ksh** lee comandos de la entrada estándar. **ksh** automáticamente activa la opción **–s** si no especificas la opción **–c**, o cualquier otro argumento de **ksh** que no sea opción. **ksh** ignora la opción **–s** si especificas **–c**.

–D — Visualiza la lista de cadenas entrecomilladas dobles que están precedidas por un **$** en el script dado, pero no ejecuta el script. **Versión**: La opción **–D** no estaba disponible con la versión de **ksh** de 16/Nov/1988.

ksh ignora un argumento inicial de – y desde que el comando **ksh** sigue la convención de comando estándar, – – finaliza las opciones y por lo tanto es ignorado. Por lo que, puedes usar – o – – para indicar un final de opciones para especificar un primer argumento que comience con –.

ksh inicialmente desactiva todas las opciones que no especificas, excepto las siguientes:

- **ksh** activa la opción **interactive** si no especificas ningún argumento que no se opción a **ksh**, y la entrada estándar es dirigida desde un terminal, y la salida estándar es dirigida a un terminal.
- **ksh** activa la opción **trackall**.
- **ksh** activa la opción **monitor** si la opción **interactive** está activada, y si **ksh** puede determinar que el sistema puede manejar control de trabajos.
- **ksh** fija las opciones de los editores **emacs**, **gmacs** y/o **vi** basadas en los valores de las variables **EDITOR** y **VISUAL**.
- **ksh** activa la opción **bgnice**. Este es el comportamiento por defecto cuando está en modo interactivo.
- **ksh** fija la opción **restricted** si la variable **SHELL** o el nombre de fichero de la shell es **rsh**, **krsh**, o **rksh**.
- **ksh** activa la opción **privileged** si el identificador de usuario efectivo del proceso no es igual al identificador de usuario real del proceso, o si el identificador de grupo efectivo del proceso no es igual al identificador de grupo real del proceso.

Fichero de entorno

Si la opción **privileged (-p)**:

- Está desactiva. Si está en modo interactivo, **ksh** expande el valor de la variable **ENV** para expansión de parámetro. Si la expansión lleva al nombre de un fichero para el que tienes permiso para leer, entonces **ksh** lee y ejecuta cada uno de los comandos de este fichero en el entorno actual. **Versión**: Con la versión de 16/Nov/1988 de **ksh**, el fichero **ENV** era también procesado para shells no interactivos.
- Está activada. Si existe el fichero cuyo nombre es **/etc/suid_profile** y **ksh** tiene permiso para leer este fichero, entonces **ksh** lee y ejecuta cada uno de los comandos de este fichero en el entorno actual.

Para hacer que las definiciones de alias tengan efecto para todas las invocaciones interactivas de **ksh**, debes ponerlas en tu fichero **ENV**.

Fichero histórico

Para shells interactivas, **ksh** abre o crea el fichero histórico tan pronto como:

- Se lee la definición de la primera función mientras que la opción **nolog** esté desactivada
- Después de que el fichero de entorno ha sido leído y ejecutado.

Para shells no interactivas, **ksh** abre y crea el fichero histórico cuando es referenciado por primera vez, por ejemplo, con **read –s** o **print –s**.

SHELLS DE LOGIN O ENTRADA AL SISTEMA

Si el nombre del programa **ksh** comienza con un – (menos), entonces **ksh** es un shell de login. Un shell de login es lo mismo que una invocación de **ksh** normal excepto en que:

- No puedes especificar ningún argumento que no sea opción
- **ksh** lee y ejecuta el fichero **/etc/profile** si existe. Este fichero es creado por el administrador del sistema.
- Si la opción **privileged** está desactivada, entonces **ksh** lee y ejecuta los comandos del fichero definidos por la expansión de **${HOME: –.}/.profile**.

Estos ficheros profile son leídos y ejecutados un comando cada vez, de forma que los alias definidos en estos ficheros afectan a los comandos subsecuentes en el mismo fichero.

SHELLS RESTRINGIDOS

Un shell restringido es un entorno de ejecución que está más controlado que el entorno de la shell **ksh** estándar. Con una shell restringida no puedes:

- Cambiar el directorio de trabajo
- Fijar el valor o atributos de las variables **SHELL**, **ENV** o **PATH**.
- Especificar el pathname de un comando con un **/** en él.
- Redireccionar la salida de un comando con **>**, **>|**, **<>**, o **>>**.
- Añadir comandos built-in

Estas restricciones no se aplican cuando **ksh** procesa el fichero **.profile**, y el fichero definido por la variable **ENV**.

Si un shell restringido ejecuta un shell script, **ksh** lo ejecuta en un entorno no restringido.

SHELL SCRIPTS

Un script puede ser invocado por nombre o como el primer argumento de **ksh**. A excepción de cómo se lista en el siguiente párrafo, la invocación de un script por nombre no produce una invocación por separado de **ksh** en la mayoría de los sistemas. Los scripts invocados por nombre necesitan menos tiempo para empezar.

Los shell scripts se llevan a cabo como una invocación separada de **ksh** cuando:

- El script tiene permisos de ejecución pero no tiene permisos de lectura
- El script tiene los permisos setuid o setgid fijados
- **Dependiente del sistema**: En algunos sistemas, todos los shell scripts se llevan a cabo como una invocación separada de **ksh** para todos lo scripts cuyo primera línea comienza con los caracteres **#!** seguidos por el pathname de **ksh**. Los caracteres después de **#!** definen el interpretador para ejecutar el procesamiento del script.

Cuando **ksh** ejecuta un script sin una invocación de **ksh** separada:

- El fichero histórico se hereda
- Los arrays exportados son heredados

Un script se ejecuta hasta que:

- No le quedan más comandos por ejecutar
- Se ejecuta el comando **exit**.
- Se ejecuta el comando **return** estando fuera de una función.
- Se ejecuta el comando **exec** con un argumento. En este caso el script se remplazará por el programa definido por el primer argumento de **exec**.
- Se detecta uno de los errores listados en la página xxxx mientras se procesa un comando built-in de los que aparecen con el símbolo .
- Se recibe una señal que no está siendo ignorada, para la cual no se ha fijado ningún trap, y que normalmente hace que un proceso termine.
- Se sale de un comando con un valor de retorno Falso mientras que la opción **errexit** está activada y no se ha definido ningún trap **ERR**.

SUBSHELLS

Un subshell es un entorno separado que es un copia del entorno de shell del padre. Los cambios realizados al entorno de subshell no afectan al entorno del padre. Un entorno de subshell no necesita ser un proceso separado.

ksh crea subshells para realizar:

- Los comandos (...)
- Sustitución de comando
- Co-procesos
- Procesos desatendidos o background
- Cada elemento de un pipeline a excepción del último

FUNCIONES DE SHELL

Las funciones de shell definidas con la palabra reservada **function** comparten todo el entorno con el proceso que las llama excepto:

- Asignaciones de variables que son parte de la llamada
- Parámetros posicionales
- Ajustes de opciones
- Variables declaradas dentro de la función con **typeset**

SCRIPTS DE PUNTO

Un script punto es un script que está especificado con el comando **.** (punto). Un script punto se lee por completo y después expandido y ejecutado en el entorno de proceso actual. Por lo que,

ningún alias definido en tales script no afectan a los comandos subsecuentes en el mismo script. Un error de sintaxis en un script punto hace que el script que lo llamó termine.

Los argumentos especificados con el comando **.** (punto) remplazan a los parámetros posicionales mientras se ejecuta el .script. **Versión**: Con la versión de **ksh** de 16/Nov/1988, los parámetros posicionales no eran restaurados después de ejecutar el .script.

FUNCIONES POSIX

Una función definida con la sintaxis ***nombre()***, se expande y ejecuta en el entorno de proceso actual como un script **.** (punto).

Los argumentos especificados con la función sustituyen a los parámetros posicionales mientras se está ejecutando la función. **Versión**: Con la versión de 16/Nov/1988 de **ksh**, las funciones con la sintaxis ***nombre()*** se ejecutaban en el mismo entorno que las funciones definidas con la palabra reservada **function**.

COMANDOS BUILT-IN

Cada built-in se evalúa en el entorno de proceso actual.

El redireccionamiento de la entrada/salida aplicado a built-ins que no sean **exec** no afecta al entorno actual.

La asignación de variables para built-ins denotados por un en el capítulo “Comandos Built-in” xxxxx afectan al entorno actual. Otras asignaciones afectan solamente al built-in específico.

PARTE V: APÉNDICE

14. GLOSARIO

Alias. Un nombre usado como una abreviación para uno o más comandos. Un alias te posibilita remplazar un nombre de comando con cualquier secuencia de caracteres que se desee.

- **Alias prefijado**. Un alias que ya está definido por el propio **ksh** cuando quiera que se invoca.
- **Alias tracked**. No es realmente un alias. Su valor se queda fijado al pathname de un programa la primera vez que se ejecuta el programa. Se usa para reducir el tiempo que **ksh** tarda en localizar un programa en peticiones subsecuentes.

Array. Una variable indexada por un subíndice

Atributo. Características que están opcionalmente asociadas con una variable, como por ejemplo readonly o uppercase.

Búsqueda de path. Las formas de encontrar el pathname de un programa que se corresponde con el nombre de programa.

Cadena de permiso. Una cadena de 10 caracteres que se usa para representar permisos de acceso.

Caracteres de control de terminal. Los caracteres de teclado que son procesados especialmente por el sistema para borrar entrada, para detener y reiniciar la salida, y para enviar señales. Los caracteres de control terminal y sus valores por defecto se listan en la página xxxx.

Comando. Una acción para que sea realizada por **ksh**. Un comando puede ser:

- **Un comando simple**.
    - Asignación de variable
    - Redireccionamiento de la entrada/salida
    - Comando built-in
    - Evaluación aritmética, ((…))
    - Función
    - Programa
- **Un comando compuesto**.
    - Pipeline
    - Comando de iteración
    - Comando condicional

Comando built-in. Un comando procesado por el propio **ksh**. Su código es interno a **ksh**.

Descriptor de fichero. Un pequeño número asociado con un fichero abierto.

Directiva. Un comando procesado por uno de los editores built-in **emacs** o **vi**.

Directorio. Un fichero que contiene nombres de ficheros o nombres de directorios.

- Directorio Bin. Un directorio donde se almacenan los programas.
- Directorio Home. Tu directorio de trabajo cuando te conectas al sistema
- Directorio Root. El directorio de más alto nivel, nombrado **/**
- Subdirectorio. Un directorio que existe dentro de otro directorio

- Directorio de Trabajo. Todo pathname que no comienza con un **/** es definido relativo a este directorio. Cada proceso tiene un directorio de trabajo.

Editor Built-in. Editores de línea de comando que son parte del propio **ksh**. Existen dos, **emacs** e **vi**.

Enlace. A cada entrada de directorio se la llama enlace. Un fichero podría tener varios enlaces a él.

Enlace simbólico. Algunos sistemas tienen un tipo de fichero especial llamado un enlace simbólico. Un enlace simbólico es un fichero especial cuyo contenido es el pathname de un fichero. Cuando se referencia como parte de un pathname, los contenidos del enlace simbólico son usados para localizar el fichero.

Entorno. El estado de un proceso, incluyendo información como sus ficheros abiertos, directorio de trabajo, máscara de creación de ficheros, y variables locales y globales.

- **Entorno hijo**. El entorno de un proceso hijo
- **Entorno actual**. El entorno del proceso actual o en curso
- **Entorno padre**. El entorno del proceso padre.
- **Entorno de subshell**. El entorno de una subshell

Entrada estándar. El fichero asociado con el descriptor de fichero 0. Los programas normalmente leen su entrada de este descriptor.

Entrecomillado. Un mecanismo para hacer que los caracteres especiales tomen su significado literal.

Error estándar. El fichero asociado con el descriptor de fichero 2. Por convención, los programas visualizan todos los mensajes de error en este descriptor.

Expansión de la tilde. **ksh** expande ciertas palabras que comienzan con un **~**.

Export. Para pasar una variable a un proceso hijo

Fichero. Un objeto del que se puede leer y/o al que se puede escribir y/o que puede se ejecutado.

Fichero de entorno. Un script que se lee y ejecuta cuando **ksh** comienza una ejecución interactiva.

Fichero histórico. El fichero en el cuál **ksh** salva cada comando que introduces de forma interactiva. Puedes editar y reintroducir comandos que estén en el fichero histórico.

Filtro. Un comando que lee de su entrada estándar y escribe a su salida estándar.

Flag u opción. Una opción de comando, normalmente indicado por una única letra precedida de un –.

Función. Un comando compuesto que consiste en una lista de comando **ksh**. Una vez que se define una función, es ejecutada cuando se referencia su nombre. Una función comparte gran parte del entorno con el script o función que la invocó.

Here-document o documento empotrado. Líneas de un script que representan la entrada estándar de un comando en el propio script.

Identificador. Una cadena de caracteres que comienzan con una letra o subrayado, y que contiene solamente letras, dígitos o subrayados.

Identificador de grupo. Cada usuario es miembro de uno o más grupos, cada uno de los cuáles está identificado por un número llamado el identificador de grupo.

- **Identificador de grupo efectivo**. Cada proceso tiene un identificador que define sus permisos con respecto al acceso de grupo de ficheros.
- **Identificador de grupo real**. Cada proceso tiene el identificador de grupo de su usuario actual.

Identificador de usuario. Cada usuario está identificado por un entero llamado el identificador de usuario.

- **Identificador de usuario efectivo**. Cada proceso tiene un identificador de usuario el cuál define sus permisos de accesos a ficheros y para enviar señales a otros procesos.
- **Identificador de usuario real**. Cada proceso recuerda el identificador de usuario del usuario real.

Intérpretador. Un programa que lee comandos de un terminal o fichero, y los ejecuta.

Interrupción. Una señal, generada típicamente por el teclado. Hace que el proceso en ejecución actual termine a menos que se haya definido un trap para que maneje la señal.

Locale. Las propiedades de las cadenas y números que dependen de convenciones de lenguaje y culturales tales como orden de comparación, clases de caracteres, y punto decimal.

Máscara de creación de ficheros. Un número que representa qué permisos deberían ser denegados cuando un usuario crea un fichero.

Modo privilegiado. Un opción que se fija cuando **ksh** se ejecuta con su identificador de usuario real que no es igual a su identificador de usuario efectivo, o con su identificador de grupo real distinto de su identificador de grupo efectivo..

Opción. Un ajuste que afecta al comportamiento de **ksh**.

Palabra. Una palabra es cualquier token que no es ninguno de los operadores definidos en la página xxxx, un here-document, o un Newline.

Pathname o nombre de path. Una cadena que se usa para identificar un fichero.

- **Pathname absoluto**. Un pathname que comienza por **/**.
- **Completar el Pathname**. La forma de generar todos los pathnames que encajan con un patrón dado.
- **Expansión del pathname**. La forma de remplazar un patrón por la lista de pathnames que encajan con el patrón.
- **Pathname físico**. Un pathname con todos los enlaces simbólicos resueltos.

Palabra delimitadora. La palabra que define la cadena que finaliza un here-document o documento empotrado.

Palabra reservada. Una palabra que está reservada como parte de la gramática de **ksh**. Las palabras reservadas son reconocidas como palabras reservadas solamente en ciertos contextos.

Parámetro. Una entidad que mantiene un valor en el lenguaje **ksh**. Vea también Variable xxxx.

- **Parámetro nominado**. Un parámetro denotado por un identificador. Una variable shell.
- **Expansión de Parámetros**. Remplazar los parámetros por sus valores.
- **Modificador de parámetro**. Una operación que se aplica cuando se expande el parámetro.
- **Parámetro posicional**. Un parámetro designado por un número.
- **Parámetro Especial**. Un parámetro cuyo nombre es $, #, @, !, –, o *.

Particionamiento de campos. Después de la expansión de parámetros y la sustitución de comandos, **ksh** parte los campos resultantes en argumentos de comando.

Patrón. Una cadena de caracteres que consta de caracteres literales que encajan con ellos mismos solamente, y caracteres de patrón que encajan con uno o más caracteres.

- **Sub-patrones**. Una cadena de caracteres que consiste en una *lista-de-patrón* encerrada entre paréntesis y precedida por uno de los caracteres @, ?, *, +, o !.
- **Lista de patrones**. Uno o más patrones separados por **|** o **&**. Una lista de patrones debe aparecer dentro de un sub-patrón.

Permiso. Los derechos que tienes para leer, escribir y/o ejecutar un fichero, o para leer, escribir, y/o buscar en un directorio.

Pipeline. Uno o más procesos conectados juntos por pipes.

Pipe o tubería. Un conducto por el cuál un stream de caracteres puede pasar desde un proceso a otro. Cada final de un pipe está asociado con un descriptor de fichero. Un proceso se parará y esperará si lee de un pipe vacío o escribe a un pipe lleno.

Proceso. Una única hebra o thread de ejecución que consta de un programa y un entorno de ejecución.

- **Proceso hijo**. Un proceso hijo creado por un proceso.
- **Co-proceso**. Un proceso creado por **ksh** que tiene su entrada y salida conectadas a **ksh** por pipes.
- **Proceso padre**. Un proceso que crea un proceso hijo.
- **Grupo de proceso**. Cada proceso activo es un miembro de un grupo identificado por el identificador de grupo de proceso.
- **Identificador de grupo de proceso**. El identificador de grupo de proceso del primer proceso que inicia un nuevo grupo de proceso.
- **Identificador de proceso**. Cada proceso activo está identificado únicamente con un entero positivo llamado el identificador de proceso.

Profile. Un fichero que contiene comandos de la shell. El fichero se ejecuta cuando te conectas al sistema.

Prompt. Un mensaje que un **ksh** interactivo visualiza cuando esta listo para leer entrada.

Redireccionamiento de la entrada/salida. El proceso de cambiar el fichero asociado con uno o más descriptores de fichero.

Referencia hacia atrás o backreference. Usada dentro de un patrón para referirse a la cadena que encajó con un subpatrón anterior. Se denota con un *dígito*, donde *dígito* es el número de subpatrón.

Salida estándar. El fichero asociado con el descriptor de fichero 1. Los programas a menudo escriben su salida a este descriptor.

Script. Un programa escrito en el lenguaje **ksh**.

- **Script Punto**. Un script que se lee y ejecuta en el entorno actual.

Señal. Un mensaje asíncrono que consta de un número que puede ser enviado desde un proceso a otro. También puede ser enviado desde el sistema operativo a un proceso cuando el usuario pulsa ciertas teclas, o cuando se levanta una condición excepcional.

Shell restringido. Una opción la cuál, cuando está fijada o activa, limita el conjunto de comandos que pueden ser ejecutados.

Subíndice. Una cadena o una expresión aritmética contenida entre llaves después de un nombre de variable. Cada subíndice de una variable puede almacenar puede almacenar un valor separado.

Subshell. Una shell hija que inicialmente contiene una copia del entorno shell padre. Un subshell no tiene que ser un proceso separado.

Superusuario. Un identificador de usuario que no tiene restricciones de permisos de fichero.

Sustitución de comando. Una palabra o parte de una palabra puede ser remplazada por la salida de un comando.

Token. **ksh** parte su entrada en unidades llamadas tokens.

Trap. Una especificación para que una acción sea realizada por **ksh** cuando ocurra una condición dada. Un ejemplo de una condición es la recepción de una señal.

Trabajo. Un sinónimo para un pipeline iniciado por un shell interactivo.

- **Trabajo desatendido o background**. Un trabajo ejecutándose en un grupo de proceso que no está asociado con el terminal.
- **Trabajo foreground o en primer plano**. Un trabajo ejecutándose en un grupo de proceso que está asociado con el terminal.
- **Control de trabajos**. La habilidad de detener trabajos y cambiarlos desde modo desatendido a primer plano y viceversa.

Valor de retorno. Un número de 0 a 255 que es devuelto por cada comando. Un valor de cero representa que el comando ha finalizado correctamente o un valor booleano de Verdadero.

Variable. Un parámetro de la shell denotado por una lista separada por puntos de identificadores.

- **Variable Array**. Una variable que está indexada por un subíndice.
- **Asignación de variable**. Un comando que asigna un valor a una variable.

15. REFERENCIA RÁPIDA

16. PORTABILIDAD

CARACTERÍSTICAS DE KSH QUE NO ESTÁN PRESENTES EN LA SHELL DE BOURNE

Las siguientes características no están en la versión temprana de la Bourne Shell. Por lo tanto, para una portabilidad máxima, evite el uso de estas características **ksh**:

- # para comentarios. (Utilice : en su lugar)
- ! para negación dentro de una clase de caracteres
- : dentro de la expansión de parámetros
- Redireccionamiento con comando built-in
- Funciones shell

test y **echo** no eran built-ins en versiones tempranas de la Bourne Shell y por lo tanto su comportamiento es dependiente del sistema.

CARACTERÍSTICAS DE KSH EN POSIX QUE NO ESTÁN EN SYSTEM V

CARACTERÍSTICAS DE KSH QUE NO ESTÁN EN EL SHELL POSIX

COMPATIBILIDAD DE KSH CON LA SHELL DE SYSTEM V

NUEVAS CARACTERÍSTICAS EN LA VERSIÓN DE KSH DE 28-DICIEMBRE-1993

CARACTERÍSTICAS OBSOLETAS

EXTENSIONES POSIBLES

INDICACIONES PARA LOS USUARIOS DE CSH SE PASEN A KSH

17. CONJUNTO DE CARACTERES

ÍNDICE