El Lenguaje de Programación AWK/GAWK

Una guía de Usuario para AWK

Jesús Alberto Vidal Cortés

http://inicia.es/de/chube

chube@inicia.es

Madrid, Febrero de 2002

1. Introducción

Dentro de las herramientas del sistema UNIX **awk** es equivalente a una navaja del ejercito suizo, que es útil para modificar archivos, buscar y transformar bases de datos, generar informes simples y otras muchas cosas. **Awk** puede usarse para buscar un nombre particular en una carta o para añadir un nuevo campo a una base de datos pequeña. También se puede utilizar para realizar el tipo de funciones que proporcionan muchas de las otras herramientas del sistema UNIX – buscar patrones, como **egrep**, o modificar archivos, como **tr** o **sed** --. Pero puesto que también es un lenguaje de programación, resulta más potente y flexible que cualquiera de ellos.

Awk está especialmente diseñado para trabajar con archivos estructurados y patrones de texto. Dispone de características internas para descomponer líneas de entrada en campos y comparar estos campos con patrones que se especifiquen. Debido a estas posibilidades, resulta particularmente apropiado para trabajar con archivos que contienen información estructurada en campos, como inventarios, listas de correo y otros archivos de bases de datos simples. Este manual mostrará como utilizar **awk** para trabajar con tales tipos de archivos.

Muchos programas útiles **awk** solamente son de una línea de longitud, pero incluso un programa **awk** de una línea puede ser el equivalente de una herramienta regular del sistema UNIX. Por ejemplo, con un programa awk de una línea puede contarse el número de líneas de un archivo (como **wc**), imprimir el primer campo de cada línea (como **cut**), imprimir todas las líneas que contienen la palabra <<comunicación>> (como **grep**), intercambiar la posición de los campos tercero y cuarto de cada línea (**join** y **paste**) o borrar el último campo de cada línea. Sin embargo, **awk** es un lenguaje de programación con estructuras de control, funciones, y variables. Así, si se aprenden órdenes awk adicionales, pueden escribirse programas más complejos.

El nombre de awk se debe a las iniciales de sus diseñadores: Alfred V. **A**ho, Peter J. **W**einberger y Brian W. **K**ernighan. La versión original de awk fue escrita en 1977 en los Laboratorios de AT&T. En 1985 una nueva versión hizo al lenguaje de programación más potente, introduciendo funciones definidas por el usuario, múltiples streams de entrada y evaluación de expresiones regulares. Esta nueva versión estuvo disponible de forma general con el Unix System V Release 3.1.El Release 4 de System V añadió algunas características nuevas y también corrigió algunos de los agujeros que presentaba el lenguaje.

La implementación GNU, **gawk**, fue escrita en 1986 por **Paul Rubin** y **Jay Fenlason**, con consejos de Richard Stallman. John Woods también contribuyó con parte del código. En 1988 y 1999, David Trueman, con ayuda de Arnold Robbins, trabajaron duramente para hacer a gawk compatible con el nuevo awk.

2. Empezando con awk

La función básica de awk es buscar líneas en ficheros (u otras unidades de texto) que contienen ciertos patrones. Cuando en una línea se encuentra un patrón, awk realiza las acciones especificadas para dicho patrón sobre dicha línea. Awk sigue realizando el procesamiento de las líneas de entrada de esta forma hasta que se llega al final del fichero.

Cuando ejecutas awk, especificas un programa awk que le dice a awk que tiene que hacer. El programa consiste en una serie de reglas (podría también contener *definiciones de funciones*, pero esa es una característica avanzada, así que ignorémosla por ahora. Ver la sección 12. Funciones definidas por el Usuario). Cada regla especifica un patrón a buscar, y una acción a realizar cuando se encuentre dicho patrón en el registro de entrada.

Sintácticamente, una regla consiste en un patrón seguido por una acción. La acción se encierra entre llaves para separarlas de los patrones. Las reglas están separadas por caracteres newline. Por lo tanto, un programa awk presentaría la siguiente forma:

```
patrón { acción }
patrón { acción }
...
```

Un ejemplo muy simple

El siguiente comando ejecuta un programa awk simple que busca la cadena de caracteres 'foo' en el fichero de entrada 'Lista-BBS' y si la encuentra la imprime. (Las cadenas de caracteres son normalmente llamadas *cadenas*.)

```
awk '/foo/ { print $0 }' Lista-BBS
```

Cuando se encuentran líneas que contengan 'foo', éstas son impresas, ya que 'print $0' hace que se imprima la línea actual.

Habrás advertido las barras, '/', alrededor de la cadena 'foo' en el programa awk. Las barras indican que 'foo' es un patrón para búsqueda. Este tipo de patrón se llama *expresión regular*, y es cubierta con más detalle posteriormente (Ver la sección Expresiones Regulares como Patrones). Existen comillas simples alrededor del programa awk para que el shell no interprete el programa como caracteres especiales de la shell.

Aquí se muestra lo que este programa imprime:

```
fooey       555-1234     2400/1200/300     B
foot        555-6699     1200/300          B
macfoo      555-6480     1200/300          A
sabafoo     555-2127     1200/300          C
```

En una regla awk, o el patrón o la acción puede ser omitida, pero no ambos. Si el patrón se omite, entonces la acción se realiza para cada línea de entrada. Si se omite la acción, la acción por defecto es imprimir todas las líneas que concuerden con el patrón.

Por lo que, podríamos omitir la acción (la sentencia print y las llaves) en el ejemplo anterior, y el resultado sería el mismo: todas las líneas que concuerden con el patrón 'foo' serán impresas. Por comparación, la omisión de

la sentencia print pero manteniendo las llaves produce una acción vacía que no realiza nada; por lo que no se imprimirá ninguna línea.

Un ejemplo con dos reglas

La utilidad awk lee los ficheros de entrada línea a línea. Para cada línea, awk comprueba todos los patrones de todas las reglas. Si concuerdan varios patrones entonces se ejecutan las distintas acciones de cada patrón, en el orden en que aparecen en el programa awk. Si no concuerda ningún patrón, entonces no se ejecuta ninguna acción.

Después de ejecutar las acciones de las reglas que concuerden con la línea (quizás no concuerde ninguna), awk lee la siguiente línea (aunque existen excepciones, Ver la sección La Sentencia next). Esto continúa hasta que se alcanza el final de fichero.

Por ejemplo, el programa awk:

```
/12/  { print $0 }
/21/  { print $0 }
```

contiene dos reglas. La primera regla tiene la cadena ‘12’ como patrón y realiza la acción ‘print $0’. La segunda regla tiene la cadena ‘21’ como patrón y también realiza la acción ‘print $0’. La accion/es de cada regla se encierra entre un par de llaves.

Este programa imprime cada línea que contiene o la cadena ‘12’ o la cadena ‘21’. Si una línea contiene ambas cadenas, ésta línea es impresa dos veces, una vez por cada regla.

Si ejecutamos este programa sobre nuestros dos ficheros de datos de ejemplo, `Lista-BBS' y `inventario-enviado' (Ver el apéndice A – Ficheros de Datos para los Ejemplos), como se muestra aquí:

```
awk '/12/ { print $0 }
     /21/ { print $0 }' Lista-BBS inventario-enviado
```

obtenemos la siguiente salida:

```
aardvark     555-5553     1200/300          B
alpo-net     555-3412     2400/1200/300     A
barfly       555-7685     1200/300          A
bites        555-1675     2400/1200/300     A
core         555-2912     1200/300          C
fooey        555-1234     2400/1200/300     B
foot         555-6699     1200/300          B
macfoo       555-6480     1200/300          A
sdace        555-3430     2400/1200/300     A
sabafoo      555-2127     1200/300          C
sabafoo      555-2127     1200/300          C
Jan  21  36  64 620
Apr  21  70  74 514
```

Dese cuenta de que la línea contenida en `Lista-BBS' que empieza por `sabafoo' fue impresa dos veces, una vez para cada regla.

Un ejemplo más complejo

Aquí tienes un ejemplo para darte una idea de lo que puede hacer un programa awk típico. Este programa muestra como awk puede ser usado para sumarizar, seleccionar y relocalizar la salida de otra utilidad. En el ejemplo se usan características que no han sido todavía explicadas, así que no te preocupes si no entiendes todos los detalles.

```
ls -l | awk '$5 == "Nov" { sum += $4 }
             END { print sum }'
```

Este comando imprime el número total de bytes de todos los ficheros del directorio actual que fueron modificados por última vez en Noviembre (de cualquier año). (En el C Shell necesitarás teclear un punto y coma y después una barra invertida al final de la primera línea; en el Shell de Bourne o el Shell Bourne-Again puedes teclear el ejemplo tal y como se muestra).

La parte de este ejemplo `` `ls -l` `` es un comando que te da un listado completo de todos los ficheros de un directorio, incluyendo el tamaño del fichero y la fecha. La salida es algo como lo siguiente:

```
-rw-r--r--  1 close        1933 Nov  7 13:05 Makefile
-rw-r--r--  1 close       10809 Nov  7 13:03 gawk.h
-rw-r--r--  1 close         983 Apr 13 12:14 gawk.tab.h
-rw-r--r--  1 close       31869 Jun 15 12:20 gawk.y
-rw-r--r--  1 close       22414 Nov  7 13:03 gawk1.c
-rw-r--r--  1 close       37455 Nov  7 13:03 gawk2.c
-rw-r--r--  1 close       27511 Dec  9 13:07 gawk3.c
-rw-r--r--  1 close        7989 Nov  7 13:03 gawk4.c
```

El primer campo presenta los permisos de lectura-escritura, el segundo campo contiene los enlaces al fichero, y el tercer campo identifica al propietario del fichero. El cuarto campo contiene el tamaño del fichero en bytes. Los campos quinto, sexto y séptimo contienen el mes, el día y la hora, respectivamente, en la cual el fichero fue modificado por última vez. Finalmente, el octavo campo contiene el nombre del fichero.

La expresión $5=="Nov" de tu programa awk es una expresión que chequea si el quinto campo de la salida generada por `'ls -l'` es igual a 'Nov'. Cada vez que una línea presenta la cadena 'Nov' en su quinto campo, se realiza la acción `` `{ sum += $4 }' ``. Esto añade el cuarto campo (el tamaño del fichero) a la variable sum. Como resultado, cuando awk ha finalizado la lectura de líneas de entrada, sum es la suma de los tamaños de los ficheros cuyas líneas contuvieron el patrón.

Después de que la última línea de la salida generada por `ls` haya sido procesada, se ejecuta la regla `END`, y el valor de la variable `SUM` se imprime. En este ejemplo, el valor de sum sería 80600.

Estas técnicas de awk más avanzadas serán cubiertas en secciones posteriores. (Ver la sección 7. Acciones: Overview). Antes de enseñarte la programación de awk más avanzada, debes saber como awk interpreta tu entrada y visualiza tu salida. Manipulando campos y usando sentencia print, puedes producir algunos informes muy útiles y de apariencia espectacular.

Cómo ejecutar los programas awk

Hay varias formas de ejecutar un programa awk. Si el programa es corto, es más fácil incluir el programa en el mismo comando que ejecuta awk, de la siguiente forma:

```
awk 'program' input-file1 input-file2 ...
```

donde *program* consiste en una serie de patrones y acciones, según se describió anteriormente.

Cuando el programa es largo, probablemente preferirías poner el programa en un fichero y ejecutarlo con un comando de la siguiente forma:

```
awk -f program-file input-file1 input-file2 ...
```

Programas de ejecución rápida (One-shot Throw-away)

Una vez que estás familiarizado con awk, escribirás con frecuencia programas simples sobre la marcha para solucionar algo puntual. Puedes escribir el programa como el primer argumento del comando awk, de la siguiente forma:

```
awk 'programa' input-file1 input-file2 ...
```

donde *programa* consiste en una serie de *patrones* y *acciones*, como se describieron anteriormente.

Este formato de comando le dice al shell que ejecute awk y use *programa* para procesar los registros en el fichero(s) de entrada. Aparecen comillas simples alrededor del *programa* de forma que el shell no interprete ningún carácter awk como carácter especial del shell. Esto hace que el shell trate *programa* por completo como un único argumento de awk. Por lo tanto permite que *programa* tenga una extensión de varias líneas.

Este formato es también útil para la ejecución de programas pequeños y medios desde shell scripts, porque evita la necesidad de un fichero separado para el programa awk. Un shell script empotrado en la línea de comando es más fiable ya que no hay otros ficheros implicados para que se produzcan fallos.

Ejecutar awk sin Ficheros de Entrada

También puedes usar awk sin ficheros de entrada. Si tecleas la siguiente línea de comando:

```
awk 'programa'
```

entonces awk aplica el *programa* a la *entrada estándar*, que en la mayoría de los casos es todo lo que tecleas en el terminal.

Esto continuará hasta que indiques el final de fichero mediante la combinación de teclas **Control-d**.

Por ejemplo, si tu ejecutas el siguiente comando:

```
awk '/th/'
```

cualquier cosa que teclees a continuación será cogido como datos para tu programa awk. Si continúas y tecleas los siguientes datos:

```
Kathy
Ben
Tom
```

```
Beth

Seth

Karen

Thomas

Control-d
```

Entonces awk imprime la siguiente salida:

```
Kathy

Beth

Seth
```

como las líneas que contienen el patrón especificado 'th'. Dese cuenta de que no reconoce "Thomas" como línea que cumpla el patrón. El lenguaje awk hace distinciones entre mayúsculas y minúsculas, y busca la concordancia con el patrón exacta. (Sin embargo, puedes evitar esto con la variable `IGNORECASE`. Ver la sección Sensibilidad a Mayúsculas en el Matching.)

Ejecución de Programas Grandes

Algunas veces, tus programas awk pueden ser muy grandes. En este caso, es más conveniente poner el programa en un fichero separado. Para decirle a awk que use ese fichero como programa, se teclea:

```
awk -f fichero-fuente input-file1 input-file2 ...
```

El `'-f'` le dice a la utilidad awk que obtenga el programa awk del fichero *fichero-fuente*. Cualquier nombre de fichero puede ser usado como *fichero-fuente*. Por ejemplo, podrías poner el programa:

```
/th/
```

en el fichero `'th-prog'`. Entonces este comando:

```
awk -f th-prog
```

hace la misma cosa que este otro:

```
awk '/th/'
```

lo cual fue explicado anteriormente (ver sección Ejecutar awk sin Ficheros de Entrada). Dese cuenta que no necesitas normalmente comillas simples alrededor del nombre del fichero que especificas con '-f', porque la mayoría de los nombres de ficheros no contienen ninguno de los caracteres especiales de la shell.

Si quieres identificar tus ficheros con programas awk como tales, puedes añadir la extensión `'.awk'` a los nombres de ficheros. Esto no afecta a la ejecución de un programa awk, pero hace que el nombre del fichero sea más legible y fácil de localizar.

Programas awk Ejecutables

Una vez que hayas aprendido awk, podrías querer escribir scripts de awk contenidos dentro de un Shell Script ejecutable de Unix, usando el mecanismo de script `'#!'`. Puedes hacer esto en sistemas Unix BSD y (algún día) en GNU.

Por ejemplo, podrías crear un fichero texto llamado ‘hola’ que tuviese lo siguiente (donde ‘BEGIN’ es una característica todavía no vista):

```
#! /bin/awk -f
# un programa awk de ejemplo
BEGIN { print "Hola Mundo" }
```

Después de hacer este fichero ejecutable (con el comando chmod), puedes teclear simplemente:

```
hola
```

desde el shell, y el sistema ejecutará el awk del mismo modo que si hubieses tecleado:

```
awk -f hello
```

Los scripts de awk auto contenidos (introducidos dentro de un Shell Script ejecutable de Unix) son útiles cuando quieres escribir un programa que puedan invocar los usuarios sin que sepan que el programa está escrito en awk.

Si tu sistema no soporta el mecanismo ‘#!’, puedes obtener un efecto similar usando un shell script regular. Sería algo del estilo a esto:

: Los dos puntos te aseguran de que este script se ejecuta con el shell de Bourne.

```
awk 'program' "$@"
```

Usando esta técnica, es vital encerrar el *program* entre comillas simples para proteger que el programa sea interpretado por el shell.

El `"$@"' hace que el shell se salte todos los argumentos de la línea de comando al programa awk, sin interpretación. La primera línea, que comienza con un colon, se usa de forma que este shell script funcionará incluso si es invocado por un usuario que use la C-Shell.

Comentarios en Programas awk

Un *comentario* es un texto que es incluido en el programa para que dicho programa sea más entendible por los lectores del mismo; no siendo los comentarios realmente parte del programa. Lo comentarios pueden explicar que es lo que hace el programa, y cómo lo hace. Prácticamente todos los lenguajes de programación tiene alguna forma para introducir comentarios, ya que hay muchos programas realmente difíciles de entender sin su ayuda extra en forma de comentarios.

En el lenguaje awk, un comentario comienza con el signo almohadilla, ‘#’, y dicho comentario continúa hasta el final de la línea. El Lenguaje awk ignora el resto de una línea desde el momento que encuentra un carácter ‘#’. Por ejemplo, podríamos haber puesto lo siguiente en el programa ‘th-prog’.

```
# Este programa encuentra registros que contengan el patrón ‘th’. De esta forma
# es como continuas el comentario en una línea adicional.
/th/
```

Sentencias frente a Líneas en awk

Bastante a menudo, cada línea en un programa awk es una sentencia separada o regla separada, como esta:

```
awk '/12/  { print $0 }

     /21/  { print $0 }' Lista-BBS inventario-enviado
```

Pero algunas veces una sentencia puede ocupar más de una línea, y una línea puede contener varias sentencias. Puedes partir una sentencias en múltiples líneas insertando un carácter newline (salto de línea) detrás de alguno de los siguientes caracteres:

```
,    {    ?    :    ||    &&    do    else
```

Un carácter newline el cualquier otro punto es considerado como el final de una sentencia.

Si te gustaría partir una única sentencia en dos líneas en un determinado punto en el cual un carácter newline terminaría dicha sentencia, puedes continuarla finalizando la primera línea con un carácter de barra invertida, '\'. Esto esta permitido absolutamente en cualquier sitio de la sentencia, incluso en mitad de una cadena o expresión regular. Por ejemplo:

```
awk '/Este programa es demasiado grande, así que continuala \

 en la línea siguiente / { print $1 }'
```

Nosotros no hemos utilizado, por norma, el uso de la continuación de una línea mediante la barra invertida en los programas de ejemplo de este manual. Ya que no hay límite en la longitud de una línea, nunca es estrictamente necesario particionar ésta; es simplemente por cuestión de estética. La continuación mediante barra invertida es muy útil cuando tu programa awk está en un fichero fuente independiente, en lugar de ser tecleado en la línea de comandos.

Precaución: la continuación de línea usando la barra invertida no funciona tal y como se describe aquí bajo la C Shell. La continuación con barra invertida funciona en tus ficheros con programas awk, y también en los programas que se lanzan escribiéndolos directamente en la línea de comandos teniendo en cuenta que estés usando el Bourne shell o Bourne-again Shell. Pero el C shell usado en Unix Berkeley se comporta de forma diferente! Bajo este Shell, debes usar dos barras invertidas en una fila, seguido por un carácter newline.

Cuando las sentencias awk dentro de una regla son cortas, podrías desear poner más de una en una misma línea. Se puede hacer esto separando las sentencias mediante puntos y coma `';'`. Esto se aplica también a las mismas reglas. Por lo que, el programa de más arriba se podría haber escrito:

```
/12/ { print $0 } ; /21/ { print $0 }
```

Nota: el requerimiento de que las reglas en la misma línea deben ser separadas por puntos y comas es un cambio reciente en el Lenguaje awk; fue hecho por consistencia con el tratamiento de las sentencias dentro de una acción.

Cuando usar awk

Te estarás preguntando: ¿qué uso le puedo yo dar a todo esto? Utilizando programas de utilidades adicionales, patrones más avanzados, separadores de campo, sentencias aritméticas, y otros criterios de selección, puedes producir una salida mucho más compleja.

El lenguaje awk es muy útil para producir informes a partir de grandes cantidades de datos raw, tales como información de sumarización a partir de la salida de otros programas de utilidades tales como `ls`. (Ver la sección Un ejemplo más complejo).

Los programas escritos con awk son a menudo mucho más pequeños de lo que serían en otros lenguajes. Esto hace que los programas awk sean más fáciles de componer y usar. A menudo los programas awk pueden ser compuestos rápidamente en tu terminal, usados una y múltiples veces. Debido a que los programas de awk son interpretados, puedes evitar el ciclo de desarrollo de software normal.

Han sido escritos en awk programas complejos, incluido un completo ensamblador para microprocesadores de 8 bits (Ver la sección 19. Glosario, para más información) y un ensamblador microcodificado para una computadora Prolog de propósito especial. Sin embargo, las posibilidades de awk van más allá de tales complejidades.

Si te encuentras a ti mismo escribiendo scripts de awk de más de, digamos, unos pocos de cientos de líneas, podrías considerar usar un lenguaje de programación diferente. Emacs Lisp es una buena elección si necesitas cadenas sofisticadas o capacidades de búsqueda de patrones. El shell también está bien para búsqueda de patrones y cadenas; además, te permite el potente uso de utilidades del sistema. Lenguajes más convencionales, tales como C, C++, y Lisp, te ofrecen mejores facilidades para la programación de sistema y para manejar la complejidad de grandes programas.

3. Leyendo ficheros de Entrada

En el programa awk típico, toda la entrada se lee de la entrada estándar (normalmente el teclado) o de los fichero cuyos nombres se especifican en la línea de comando de awk. Si especificas ficheros de entrada, awk lee los datos del primer fichero hasta que alcanza el final del mismo; después lee el segundo fichero hasta que llega al final, y así sucesivamente. El nombre del fichero de entrada actual puede ser conocido por la variable implícita FILENAME (Ver la sección 13. Variables Implícitas (Built-in)).

La entrada se lee en unidades llamadas *registros*, y éstos son procesados por las reglas uno a uno. Por defecto, cada registro es una línea del fichero de entrada. Cada registro leído es dividido automáticamente en *campos*, para que puedan ser tratado más fácilmente por la regla. En raras ocasiones necesitarás usar el comando getline, el cual puede realizar entrada explícita de cualquier número de ficheros (Ver la sección Entrada explícita con getline)

Cómo se particiona la Entrada en Registros

El lenguaje awk divide sus registros de entrada en campos. Los registros están separados por un carácter llamado el separador de registros. Por defecto, el separador de registros es el carácter newline. Por lo tanto, normalmente, un registro se corresponde con una línea de texto. Algunas veces puedes necesitar un carácter diferente para separar tus registros. Puedes usar un carácter diferente mediante la llamada a la variable empotrada RS (Record Separator).

El valor de RS es una cadena que dice como separar los registros; el valor por defecto es "\n", la cadena formada por un único carácter newline. Esta es la razón por la cual un registro se corresponde, por defecto, con una línea.

RS puede tener cualquier cadena como valor, pero solamente el primer carácter de la cadena se usará como separador de registros. El resto de caracteres de la cadena serán ignorados. RS es excepcional en este sentido; awk usa el valor completo del resto de sus variables implícitas.

Puedes cambiar el valor de RS en un programa awk con el operador asignación, '=' (Ver la sección Expresiones de Asignación). El nuevo carácter separador de registros de entrada debería estar entre comillas para que sea una constante cadena. A menudo, el momento adecuado para hacer esto es el principio de la ejecución, antes de procesar ninguna entrada, de modo que el primer registro sea leído con el separador apropiado. Para hacer esto, use el patrón especial BEGIN (Ver la sección Los Patrones Especiales BEGIN y END). Por ejemplo:

```
awk 'BEGIN { RS = "/" } ; { print $0 }' Lista-BBS
```

cambia el valor de RS a "/", antes de leer ninguna entrada. Esta es una cadena cuyo primer carácter es una barra; como resultado, los registros se separarán por las barras. Después se lee el fichero de entrada, y la segunda regla en el programa awk (la acción sin patrón) imprime cada registro. Debido a que cada sentencia print añade un salto de línea al final de su salida, el efecto de este programa awk es copiar la entrada con cada carácter barra cambiado por un salto de línea. Otra forma de cambiar el separador de registros es en la línea de comandos, usando la característica asignación de variable (Ver la sección 14. Invocación de awk).

```
awk '...' RS="/" source-file
```

Esto fija el valor de RS a '/', antes de procesar *source-file*.

La cadena vacía (una cadena sin caracteres) tiene un significado especial como valor de RS: significa que los registros están separados solamente por líneas en blanco. Ver la sección Registros de múltiples líneas, para más detalles.

La utilidad awk guarda el número de registros que han sido leídos hasta el momento del fichero de entrada actual. Este valor es almacenado en la variable implícita llamada FNR. Esta variable es inicializada a cero cuando se cambia de fichero. Otra variable implícita, NR, es el número total de registros de entrada leídos de todos los ficheros. Comienza en cero pero nunca es automáticamente reseteada a cero.

Si cambias el valor de RS a mitad de la ejecución de un programa awk, el nuevo valor se usa para delimitar los registros siguientes, pero el registro que esta siendo procesado en ese momento (y los registros ya leídos) no se ven afectados.

Examinando campos

Cuando awk lee un registro de entrada, el registro es automáticamente separado o particionado por el intérprete en piezas llamadas *campos*. Por defecto, los campos son separados por espacios en blanco, al igual que las palabras de una frase. Los espacios en awk significan cualquier cadena de uno o más espacios y/o tabuladores; otros caracteres tales como newline, formfeed, etc … que son considerados como espacios en blanco por otros languajes no son considerados como espacios en blanco por awk.

El propósito de los campos es facilitar al usuario la referencia a las partes constituyentes de un registro (tratar las subpartes de un registro). No tienes que usarlos – puedes operar con el registro completo si es lo que deseas – pero los campos son los que hacen a los programas awk más sencillos tan potentes.

Para referirse a un campo en un programa awk, usas un signo de dólar '$', seguido por el número de campo que desees. Por lo tanto, $1 se refiere al primer campo, $2 se refiere al segundo, y así sucesivamente. Por ejemplo, supón que lo siguiente es una línea del fichero de entrada:

```
This seems like a pretty nice example.
```

Aquí el primer campo, o $1, es 'This'; el segundo campo, o $2, es 'seems'; y así sucesivamente. Advierta que el último campo, $7, es 'example.'. Ya que no hay espacios en blanco entre la 'e' y el punto final de la frase '.', el punto se considera como parte del campo séptimo.

No importa cuantos campos existan, el último campo de un registro puede ser representado por $NF. Por lo que, en el ejemplo anterior, $NF sería lo mismo que $7, o lo que es lo mismo 'example.'. Si intentas referirte a un campo más allá del último, tal como $8 cuando el registro tiene solo 7 campos, obtienes la cadena vacía.

NF, sin el dólar delante, es una variable implícita cuyo valor es el número de campos en el registro actual. $0, lo cual parece un intento de acceder al campo cero, es un caso especial: representa el registro de entrada completo. Esto es lo que usarías cuando no estuvieses interesado en campos. Aquí están algunos ejemplos más:

```
awk '$1 ~ /foo/ { print $0 }' Lista-BBS
```

Este ejemplo imprime cada registro del fichero 'Lista-BBS' cuyo primer campo contiene la cadena 'foo'. El operador '~' se le llama *operador de encaje, búsqueda o matching* (Ver la sección Expresiones de Comparación); chequea si una cadena (aquí, el campo $1) encaja con una expresión regular dada. En contraste, el siguiente ejemplo:

```
awk '/foo/ { print $1, $NF }' Lista-BBS
```

busca 'foo' en el *registro completo* e imprime el primer y el último campo de cada registro de entrada que contenga el patrón 'foo'.

Referenciar campos sin usar constantes

El número de un campo no necesita ser una constante. Cualquier expresión del lenguaje awk puede ser usado después de un '$' para referirse a un campo. El valor de la expresión especifica el número de campo. Si el valor es una cadena, en lugar de un número, dicha cadena se convierte a número. Considere este ejemplo:

```
awk '{ print $NR }'
```

Recuerde que NR es el número de registros leídos hasta el momento: 1 para el primer registro, 2 para el segundo, etc. Así que este ejemplo imprimiría el primer campo del primer registro, el segundo campo del segundo registro, y así sucesivamente. Para el registro veinte, se imprimiría el campo número 20; es probable que el registro tenga menos de 20 campos, así que imprimiría una línea en blanco.

Aquí tienes otro ejemplo de la utilización de una expresión como número de campo:

```
awk '{ print $(2*2) }' Lista-BBS
```

El lenguaje awk debe evaluar la expresión (2*2) y usar su valor como el número del campo a imprimir. El signo '*' representa multiplicación, así que la expresión 2*2 toma el valor 4. Los paréntesis son usados para que la multiplicación se realice antes que la operación '$'; son necesarios donde quiera que haya un operación binario en la expresión número de campo. Este ejemplo, entonces, imprime las horas de operación (el cuarto campo) para cada línea del fichero 'Lista-BBS'.

Si el número de campo toma el valor de 0, obtienes el registro entero. Por lo que, $(2-2) tiene el mismo valor que $0. Números de campo negativos no son permitidos.

El número de campos en el registro actual se almacena en la variable implícita NF (Ver la sección 13. Variables Implícitas (Built-in). La expresión $NF no es una característica especial: es la consecuencia directa de evaluar NF y usar su valor como número de campo.

Cambiando los contenidos de un campo

Puedes cambiar los contenidos de un campo dentro de un programa awk; estos cambios que awk percibe como el registro de entrada actual (La entrada real es intocable: **awk nunca modifica el fichero de entrada**). Mire este ejemplo:

```
awk '{ $3 = $2 - 10; print $2, $3 }' inventario-enviado
```

El signo '-' representa la substracción, de modo que este programa reasigna el campo tres, $3, y le da el valor del campo 2 menos el campo décimo, $2 - $10. (Ver la sección Operadores Aritméticos). El campo dos, y el nuevo valor del campo tres son impresos.

Para poder realizar esta operación, el texto del campo dos debe ser susceptible a ser convertido a número, la cadena de caracteres debe ser convertida a número para que se puede realizar la operación aritmética sobre dicho campo. El número resultante de la sustracción se convierte de nuevo a cadena de caracteres cuando se convierte en el campo tercero. Ver la sección Conversiones de Cadenas y Números.

Cuando cambias el valor de un campo (como es percibido por awk), el texto del registro de entrada es recalculado para contener el nuevo campo en la posición en la que estaba el antiguo. Por lo tanto, $0 cambia para reflejar el campo alterado. Por lo que,

```
awk '{ $2 = $2 - 10; print $0 }' inventario-enviado
```

imprime una copia del fichero de entrada, restándole 10 unidades a cada uno de los valores que presenta el campo 2 para todas las líneas.

También puedes asignar contenidos a campos que están fuera del rango. Por ejemplo:

```
awk '{ $6 = ($5 + $4 + $3 + $2) ; print $6 }' inventario-enviado
```

acabamos de crear el campo $6, cuyo valor es la suma de los campos $2, $3, $4, y $5. El signo '+' representa la suma. Para el fichero 'inventario-enviado', $6 representa el número total de objetos embarcados para un mes en particular.

La creación de un nuevo campo cambia la copia interna del registro de entrada actual que tiene awk – el valor de $0. Por lo que, si realiza un 'print $0' después de añadir un campo, el registro impreso incluye el nuevo campo, con el número de separadores de campo apropiado entre el nuevo campo y los campos existentes previamente.

Esta recomputación afecta y se ve afectada por varias características todavía no discutidas, en particular, el *separador de campos de salida*, OFS, que se usa para separar los campos (Ver la sección Separadores de la Salida), y NF (el número de campos; Ver la sección Examinando campos). Por ejemplo, el valor de NF se fija al número del campo mayor que hayas creado.

Anotar, sin embargo, que *referenciar* a un campo fuera de rango ***no*** cambia ni el valor de $0 ni el de NF. El referenciar a un campo fuera de rango produce simplemente una cadena nula. Por ejemplo:

```
if ($(NF+1) != "")
    print "no puede ocurrir"
else
    print "todo es normal"
```

imprimiría `todo es normal', ya que NF+1 está fuera de rango ciertamente (Ver la sección La Sentencia if, para más información sobre las sentencias de awk if-else).

Especificando como están separados los campos

El modo en el que awk divide los registros de entrada en campos es controlada por el *separador de campo*, el cual es un carácter simple o una expresión regular. Awk recorre el registro de entrada en búsqueda de coincidencias del separador de campos; los campos son el texto que se encuentra entre dichos separadores de campos encontrados. Por ejemplo, si el separador de campos es 'oo', entonces la siguiente línea:

```
moo goo gai pan
```

será particionada en tres campos: `m', ` g' y ` gai pan'.

El separador de campos está representado por la variable implícita FS. ¡Que tomen nota los programadores de la Shell! Awk no usa el nombre IFS el cual es usado por la shell. Puedes cambiar el valor de FS en el programa awk con el operador asignación, '=' (Ver la sección Expresiones de Asignación). A menudo el momento adecuado para hacer esto es el principio de la ejecución, antes de que se procese ninguna entrada, de forma que el primer registro se lea con el separador adecuado. Para hacer esto, use el patrón especial BEGIN (Ver la sección Los Patrones Especiales BEGIN y END). Por ejemplo, aquí hemos fijado el valor de la variable FS a la acadena ",":

```
awk 'BEGIN { FS = "," } ; { print $2 }'
```

Dada la siguiente línea,

John Q. Smith, 29 Oak St., Walamazoo, MI 42139

Este programa awk extrae la cadena `29 Oak St.'.

Algunas veces tus datos de entrada contendrán caracteres separadores que no separen los campos de la forma que tu pensabas que debieran estar separados. Por ejemplo, los nombres de personas en el ejemplo que hemos estado usando podría tener un título o sufijo acoplado, tal como `John Q. Smith, LXIX'. Para la entrada que contuviese dicho nombre:

```
John Q. Smith, LXIX, 29 Oak St., Walamazoo, MI 42139
```

El programa de ejemplo anterior extraería `LXIX', en lugar de `29 Oak St.'. Si estabas esperando que el programa escribiese la dirección, serías sorprendido. Así que elige la disposición de tus datos y separadores de campos cuidadosamente para prevenir tales problemas.

Como ya sabes, por defecto, los campos son separados por secuencias de espacios en blanco (espacios y tabuladores), no por espacios simples: dos espacios en una fila no delimitan un campo vacío. El valor por defecto del separador de campos es una cadena " " que contiene un único espacio. Si este valor fuese interpretado de la forma usual, cada carácter espacio separaría campos, de forma que dos espacios en una fila crearían un campos vacío entre ellos. La razón por lo que esto no ocurre es porque un espacio simple como valor de FS es un caso especial: se toma para especificar la manera por defecto de deliminar campos.

Si FS es cualquier otro carácter simple, tal y como ",", cada ocurrencia de dicho carácter separa dos campos. Dos ocurrencias consecutivas delimitan un campo vacío. Si el carácter ocurre al comienzo o al final de la línea, eso también delimita un campo vacío. El carácter espacio es el único carácter que no sigue estas reglas.

Más generalmente, el valor de FS podría ser una cadena que contuviese una expresión regular. Entonces cada coincidencia en el registro de la expresión regular separa campos. Por ejemplo, la asignación:

```
FS = ", \t"
```

Hace que cada área de una línea de entrada que consista en una coma seguida de un espacio y un tabulador sea un separador de campo ('\t' representa el tabulador).

Para un ejemplo menos trivial de una expresión regular, supón que deseas que los espacios simples separen los campos de la misma forma que lo harían unas comas si fuesen usadas. Le podrías asignar a FS el valor `"[ ]"`. Esta expresión regular concuerda con un único espacio y nada más.

El valor de FS puede ser fijado en la línea de comando. Use el argumento '-F' para hacerlo. Por ejemplo:

```
awk -F, 'program' input-files
```

fija FS para que sea el carácter ','. Dese cuenta de que el argumento aparece en mayúsculas '-F'. Esto contrasta con '-f', el cual especifica un fichero que contiene un programa awk. La distinción entre mayúsculas y minúsculas es significativa en las opciones de comando: las opciones '-F' y '-f' no tienen nada que ver la una con la otra. Puedes usar ambas opciones al mismo tiempo para fijar el argumento FS y para decirle a awk que el programa se encuentra en un determinado fichero.

Un caso especial, en modo compatibilidad (Ver la sección 14. Invocación de awk), si el argumento a '-F' es 't', entonces el FS es fijado al carácter tabulador. (Esto es porque si tú tecleas '-F\t', sin las comillas, en el shell, el carácter '\' es eliminado, de forma que awk supone que tú realmente quieres que tus campos estén separados por tabuladores, y no por t's. Use 'FS="T" en la línea de comando si deseas realmente que tus campos aparezcan separados por t's.)

Por ejemplo, utilicemos un fichero de programa awk llamado `'baud.awk'` que contiene el patrón /300/, y la acción 'print $1'. Aquí se presenta el programa:

```
/300/   { print $1 }
```

Fijemos también el valor de FS al carácter '-', y ejecute el programa sobre el fichero 'Lista-BBS' El siguiente comando imprime una lista de los nombres del bulletin boards que operan a 300 baudios y los tres primeros dígitos de sus números de teléfono:

```
awk -F- -f baud.awk Lista-BBS
```

Produce la siguiente salida:

```
aardvark      555
alpo
barfly        555
bites         555
camelot       555
core          555
fooey         555
foot          555
macfoo        555
sdace         555
sabafoo       555
```

Dese cuenta de la segunda línea de la salida. Si chequeas el fichero original, verás que la segunda línea presenta lo siguiente:

```
alpo-net    555-3412    2400/1200/300    A
```

El guíon '-' como parte del nombre del sistema fue utilizado como separador de campo, en lugar del guión que aparecía en el número de teléfono que era lo que se pretendía. Esto te demuestra porqué tienes que ser cuidadoso a la hora de elegir tus separadores de campo y registro.

El siguiente programa busca en el fichero de sistema password, e imprime las entrada de aquellos usuarios que no tiene password:

```
awk -F: '$2 == ""' /etc/passwd
```

Aquí usamos la opción '-F' de la línea de comando para fijar el separador de campo. Advierta que los campos en '/etc/passwd' están separados por dos puntos. El segundo campo representa una password de usuario encriptada, pero si el campo está vacío, dicho usuario no tiene password.

Registros de múltiples líneas

En algunas bases de datos, una sola línea no puede guardar convenientemente la información de un registro. En tales casos, puedes usar registros de líneas múltiples.

El primer paso para hacer esto es elegir el formato de tus datos: cuando los registros no vienen definidos como líneas simples, ¿cómo quieres definirlos? ¿qué debería separar los registros?

Una técnica es usar un carácter inusual o cadena para separar los registros. Por ejemplo, podrías usar el carácter formfeed (escrito '\f' en awk, como en C) para separarlos, haciendo que cada registro fuese una página del fichero. Para hacer esto, simplemente fija la variable RS a "\f" (una cadena que contenga el carácter formfeed) Cualquier otro carácter podría ser usado igualmente, siempre y cuando dicho carácter nunca forme parte de los datos posibles de un registro.

Otra técnica es tener registros separados por líneas en blanco. Como dispensación especial, una cadena nula como valor de RS indica que los registros estarán separados por una o más líneas en blanco. Si le das a la variable RS

el valor cadena nula, un registro siempre acaba en la primera línea en blanco que encuentra. Y el siguiente registro no comienza hasta que no se encuentra la siguiente línea que no sea una línea en blanco – no importa cuantas líneas en blanco aparezcan en una fila, son consideradas como un único separador de registro.

El segundo paso es separar los campos del registro. Una forma para hacer esto es poner cada campo en una línea por separado: para hacer esto, fija la variable FS a la cadena "\n". (Esta expresión regular simple concuerda con un simple carácter newline).

Otra idea es dividir cada línea en campos de la forma normal. Esto ocurre por defecto como resultado de una característica especial: cuando RS se fija a la cadena nula, el carácter newline *siempre* actúa como un separador de campo. Esto es una adición a cuales quiera separaciones de campos resultantes de FS.

Entrada explícita con getline

Hasta ahora hemos estado obteniendo nuestros ficheros de entrada desde el stream de entrada principal de awk – o la entrada estándar (normalmente tu terminal) o los ficheros especificados en la línea de comandos. El Lenguaje awk tiene un comando implícito especial llamado getline que puede ser usado para leer la entrada bajo tu control explícito.

Este comando es bastante complejo y no debería ser usado por principiantes. Se explica aquí porque este es el capítulo de la entrada. Los ejemplos que siguen a la explicación del comando getline incluyen material que no ha sido explicado todavía. Por lo tanto, vuelve a estudiar el comando getline después de haber visto el manual completo y que tengas un buen conocimiento de cómo funciona awk.

El comando getline devuelve un 1 si encuentra un registro, y 0 si se encuentra el final del fichero. Si se produce algún error al obtener un registro, debido por ejemplo a que dicho fichero no pueda ser abierto, entonces getline devolverá un –1.

En los siguientes ejemplos, *comando* representa una cadena que representa un comando del shell.

getline

El comando getline puede ser usado sin argumentos para leer la entrada del fichero de entrada actual. Todo lo que hace en este caso es leer el siguiente registro de entrada y dividirlo en campos. Esto es útil cuando has acabado de procesar el registro actual y no vas a realizar ninguna alteración del mismo y quieres procesar justo en ese momento el siguiente registro. Aquí tienes un ejemplo:

```
awk '{
     if (t = index($0, "/*")) {
          if(t > 1)
               tmp = substr($0, 1, t - 1)
          else
               tmp = ""
          u = index(substr($0, t + 2), "*/")
          while (! u) {
               getline
               t = -1
               u = index($0, "*/")
          }
```

```
            if(u <= length($0) - 2)
                $0 = tmp substr($0, t + u + 3)
            else
                $0 = tmp
        }
        print $0
}'
```

Este programa awk borra todos los comentarios, `/* ... */', de la entrda. Sustituyendo el comando 'print $0' con otras sentencias, podrías realizar un procesamiento más complejo sobre la entrada comentada, tal y como buscar si casa con una expresión regular.

Esta forma del comando getline fija el valor de NF (el número de campos; Ver la sección Examinando campos), NR (el número de registros leídos hasta ahora, Ver la sección Cómo se particiona la Entrada en Registros), FNR (el número de registros leídos del fichero de entrada actual), y el valor de $0.

Nota: el nuevo valor de $0 se usa en el chequeo de los patrones de las reglas subsiguientes. El valor original de $0 que disparó la regla que ejecutó getline se pierde. Por contraste, la sentencia next lee un nuevo registro pero inmediatamente comienza a procesarlo normalmente, comenzando con la primera regla del programa. Ver la sección La Sentencia next.

getline variable

Esta forma de getline lee un registro en la variable *variable*. Esto es útil cuando quieres que tu programa lea el siguiente registro del fichero de entrada actual, pero no quieres someter el registro que leas al procesamiento de la entrada normal.

Por ejemplo, supón que la siguiente línea es un comentario, o una cadena especial, y quieres leerla, pero quieres realizar lo que sea que no dispare ninguna regla. Esta versión de getline te permite leer esa línea y almacenarla en una variable de forma que el bucle principal de leer una línea y chequearla contra todas las reglas nunca llega a conocer dicha línea.

El siguiente ejemplo alterna (swaps) cada dos líneas de entrada. Por ejemplo, dado:

```
wan
tew
free
phore
```

produce la siguiente salida:

```
tew
wan
phore
free
```

Aquí tienes el programa:

```
awk '{
     if ((getline tmp) > 0) {
          print tmp
          print $0
     } else
          print $0
}'
```

La función getline usada de esta forma fija solamente las variables NR y FNR (y por supuesto, *variable*). El registro no es dividido en campos, de forma que los valores de los campos (incluyendo $0) y el valor de la variable NF no cambia.

getline < fichero

Esta forma de la función getline toma su entrada desde el fichero *fichero*. Aquí *fichero* es una expresión que se trata como una cadena que contiene el nombre del fichero. A la expresión '<*fichero*' se le llama redirección ya que direcciona la entrada para que venga desde un lugar diferente.

Esta forma es útil si quieres leer la entrada de un fichero en particular, en lugar de la stream de entrada principal. Por ejemplo, el siguiente programa lee su registro de entrada del fichero 'foo.input' cuando encuentra un primer campo con un valor igual a 10 en el fichero de entrada actual.

```
awk '{
if ($1 == 10) {
     getline < "foo.input"
     print
} else
     print
}'
```

Debido a que el stream de entrada principal no se usa, los valores de NR y FNR no se cambian. Pero el registro leído es partido en campos como si fuera un registro normal, por lo que los valores de $0 y otros campos son cambiados. Lo mismo le ocurre al valor de NF. Esto hace que el registro leído no sea chequeado contra todos los patrones del programa awk, del mismo modo que ocurriría si el registro hubiese sido leído normalmente por el bucle principal de proceso de awk. Sin embargo el nuevo registro es chequeado contra las reglas restantes, del mismo modo que ocurría cuando se usaba getline sin la redirección.

getline variable < fichero

Esta forma de la función getline toma su entrada del fichero *fichero* y la pone en la variable *variable*. Como anteriormente, *fichero* es una expresión cuyo valor es una cadena, la cual especifica el fichero del que se va a leer.

En esta versión de getline, ninguna de las variable implícitas cambia su valor, y el registro no es dividido en campos. La única variable que cambia es *variable*. Por ejemplo, el siguiente programa copia todos los ficheros de entrada a la salida, excepto los registros que dicen `@include *nombre_fichero*'. Tales registros son reemplazados por el contenido del fichero *nombre_fichero*.

```
awk '{
     if (NF == 2 && $1 == "@include") {
```

```
          while ((getline line < $2) > 0)
               print line
          close($2)
     } else
          print
}'
```

Advierta aquí como el nombre del fichero de entrada extra no se construye en el programa; es cogido de los datos, del segundo campo de las líneas '@include'.

La función close se llama para asegurarse que si aparecen dos líneas '@include' idénticas, el fichero especificado entero se incluye dos veces. Ver la sección Cerrado de Ficheros de Entrada y Pipes.

Una deficiencia de este programa es que no procesa las sentencias '@include' anidadas del mismo modo que un preprocesador de macros haría.

comando | getline

Puedes hacer un pipe de la salida a un comando a getline. Un pipe es simplemente una forma de enlazar la salida de un programa con la entrada de otro. En este caso, la cadena *comando* es ejecutada como un comando de la shell y su salida es pipeada dentro de awk para que sea usado como entrada. Esta forma de getline lee un registro del pipe.

Por ejemplo, el siguiente programa copia la entrada a la salida, excepto las líneas que comienzan con '@execute', las cuales son reemplazadas por la salida producida por la ejecución del resto de la línea como un comando de shell:

```
awk '{
     if ($1 == "@execute") {
          tmp = substr($0, 10)
          while ((tmp | getline) > 0)
               print
          close(tmp)
     } else
          print
}'
```

La función close se usa para asegurarse de que si aparecen dos líneas '@execute' idénticas, el comando se ejecute de nuevo para cada línea. Ver la sección Cerrado de Ficheros de Entrada y Pipes.

Dada la siguiente entrada:

```
foo
bar
baz
@execute who
bletch
```

el programa podría producir:

```
foo
```

```
bar
baz
hack     ttyv0   Jul 13 14:22
hack     ttyp0   Jul 13 14:23     (gnu:0)
hack     ttyp1   Jul 13 14:23     (gnu:0)
hack     ttyp2   Jul 13 14:23     (gnu:0)
hack     ttyp3   Jul 13 14:23     (gnu:0)
bletch
```

Dese cuenta de que este programa ejecutó el comando who e imprimió el resultado.

Esta variación de getline divide el registro en campos, fija el valor de NF y recalcula el valor de $0. Los valores de NR y FNR no son cambiados.

comando | getline variable

La salida del comando *comando* se envía a través de un pipe a getline y a la variable *variable*. Por ejemplo, el siguiente programa lee la hora y día actual a la variable tiempo_actual, usando la utilidad llamada date, y después la imprime.

```
awk 'BEGIN {
     "date" | getline tiempo_actual
     close("date")
     print "Report printed on " tiempo_actual
}'
```

En esta versión de *getline*, ninguna de las variable implícitas es cambiada, y el registro no se divide en campos.

Cerrado de Ficheros de Entrada y Pipes

Si se usa el mismo nombre de fichero o el mismo comando shell se usa con getline más de una vez durante la ejecución de un programa awk, el fichero es abierto (o el comando es ejecutado) sólo la primera vez. En ese momento, el primer registro de la entrada es leído de ese fichero o comando. La próxima vez que se use ese mismo nombre de fichero o comando con getline, otro registro se leerá de él, y así sucesivamente.

Esto implica que si quieres comenzar la lectura del mismo fichero desde el principio, o si quieres volver a ejecutar un comando (en lugar de leer más salida del comando), debes realizar unos pasos especiales. Lo que puedes usar es la función *close*, tal y como sigue:

close(fichero) o ***close(comando)***

El argumento *fichero* o *comando* puede ser cualquier expresión. Su valor debe ser exactamente el mismo que la cadena que fue usada para abrir el fichero o comenzar el comando – por ejemplo, si abres un pipe con esto:

"sort -r names" | getline foo

entonces debes cerrar el pipe de esta forma:

close("sort -r names")

Una vez que se ejecuta esta llamada a la función, el siguiente getline de ese fichero o comando reabrirá el fichero o reejecutará el comando.

4. Imprimiendo la Salida

Una de las cosas más corrientes que realizan las acciones es sacar o imprimir parte o toda la entrada. Para salida simple, utilice la sentencia print. Para un formateo más elegante utilice la sentencia printf. Ambas son descritas en este capítulo.

La sentencia print

La sentencia *print* realiza la salida con un formato estandarizado y simple. Tú especificas solamente las cadenas o números que van a ser impresos, en una lista separada por comas. Ellos son impresos separados por espacios en blanco, seguidos por un carácter newline o retorno de carro. La sentencia presenta la siguiente forma:

print *item1*, *item2*, ...

La lista completa de items podría ser opcionalmente encerrada entre paréntesis. Los paréntesis son necesarios si algunos de las expresiones items utiliza un operador relacional; por que si no, podría ser confundido con un redireccionamiento (ver la sección Redireccionando la Salida de print y printf). Los operados relacionales son `==', `!=', `<', `>', `>=', `<=', `~' y `!~' (ver sección Expresiones de Comparación)

Los items impresos pueden ser cadenas constantes o números, campos del registro actual (tal y como $1), variables, o cualquier expresión awk. La sentencia print es completamente general para procesar cuales quiera valores a imprimir. Con una excepción (ver sección Separadores de la Salida), lo que no puedes hacer es especificar como imprimirlos – cuantas columnas usar, si se usará notación exponencial o no, y otras. Para eso, necesitas la sentencia printf (ver la sección Uso de sentencias printf para una impresión más elegante).

La sentencia simple ‘print’ sin ningún item es equivalente a ‘print $0’: imprime el registro actual entero. Para imprimir una línea en blanco, use ‘print “”’, donde “” es la cadena vacía o nula.

Para imprimir una parte de texto fija, utilice una constante cadena tal y como “Hola chico” como item. Si olvidas usar los caracteres comillas dobles, tu texto será interpretado como una expresión awk, y probablemente obtendrás un error. Ten en cuenta que se imprime un espacio para separar cuales quiera dos items.

Muy a menudo, cada sentencia print produce una línea de salida. Pero no está limitado a una línea. Si el valor de un item es una cadena que contiene un newline, el carácter newline se imprime con el resto de la cadena. Un único print podría imprimir cualquier número de líneas.

Ejemplos de sentencias print

Aquí aparece un ejemplo de impresión de una cadena que contiene caracteres de retorno de carro o nueva línea empotrados:

```
awk 'BEGIN { print "línea uno\nlínea dos\nlínea tres" }'
```

produce una salida como esta:

```
línea uno
línea dos
línea tres
```

Aquí tienes un ejemplo que imprime los dos primeros campos de cada registro de entrada, con un espacio entre ellos:

```
awk '{ print $1, $2 }' inventario
```

La salida presentaría la siguiente forma:

```
Jan 13
Feb 15
Mar 15
...
```

Un error común en el uso de la sentencia print es omitir la coma entre dos items. Esto a menudo produce el efecto de imprimir los dos valores sin separar por un espacio en blanco. La razón de esto es que la yustaposición de dos expresiones tipo carácter en awk realiza la concatenación de ellas. Por ejemplo, sin la coma:

```
awk '{ print $1 $2 }' inventario
```

imprime:

```
Jan13
Feb15
Mar15
...
```

Debido a que esta salida no le resultaría informativa a la gente que no conociese el contenido del fichero "*invertario*", una línea de cabecera al principio aclararía mucho dicha salida. Añadamos pues, una cabecera a nuestro listado de meses ($1) y canastas verdes enviadas ($2). Nosotros hacemos esto usando el patrón BEGIN (ver la sección Los Patrones Especiales BEGIN y END) para hacer que la cabecera sea impresa una sola vez:

```
awk 'BEGIN {  print "Meses Canastas"
              print "-----  --------" }
           {  print $1, $2 }' inventario
```

¿Has conseguido averiguar qué ocurre? El programa imprime lo siguiente:

```
Meses Canastas
-----  --------
Jan 13
Feb 15
Mar 15
...
```

¡Las cabeceras y las líneas de detalle no están alineadas! Podemos arreglar esto imprimiendo algunos espacios en blanco entre los dos campos:

```
awk 'BEGIN { print "Meses Canastas"
             print "-----  --------" }
           { print $1, "     ", $2 }' inventario
```

Puedes imaginar que esta forma de alineación de columnas se puede volver realmente complicado cuando tienes que cuadrar muchas columnas. Contar los espacios para dos o tres columnas puede ser simple, pero con más de tres columnas te puedes perder bastante facílmente. Este es el motivo por el que se creó la sentencia printf (Ver la sección Uso de sentencias printf para una impresión más elegante); una de sus especialidades es la de alinear las columnas de datos.

Separadores de la Salida

Como se mencionó anteriormente, una sentencia print contiene una lista de items, separados por comas. En la salida, los items son separados normalmente por simples espacios en blanco. Pero esto no tiene por que ser así; el espacio es solamente el separador por defecto. Puedes especificar cualquier cadena de caracteres para usarla como el *separador de campos de salida* fijando la variable implícita OFS. El valor inicial de esta variable es la cadena " ".

La salida completa de una sentencia print se le llama *registro de salida*. Cada sentencia print imprime un registro de salida y después imprime una cadena llamada el *separador de registros de salida*. La variable implícita ORS determina esta cadena. El valor inicial de la variable es la cadena "\n' o lo que es lo mismo el carácter newline, por lo que, normalmente cada sentencia print crea una línea distinta.

Puedes cambiar como se separan los campos y registros de salida, asignándoles nuevos valores a las variables OFS y/o ORS. La sitio normal para hacer esto es en la regla BEGIN (Ver la sección Los Patrones Especiales BEGIN y END), de modo que tomen sus valores antes de que se procese ninguna entrada. También podrías hacer esto con asignaciones en la línea de comando, antes de los nombres de tus ficheros de entrada.

El siguiente ejemplo imprime el primer y segundo campo de cada registro de entrada separados por un punto y coma, añadiéndole una línea en blanco adicional después de cada registro en la salida:

```
awk 'BEGIN { OFS = ";"; ORS = "\n\n" }
          { print $1, $2 }'  Lista-BBS
```

Si el valor de ORS no contiene un carácter newline, toda tu salida se generará a una única línea, a menos que pongas saltos de línea de alguna otra forma.

Uso de sentencias printf para una impresión más elegante

Si quieres un control más preciso sobre el formato de la salida del que te da la sentencia print, utilice printf. Con printf puedes especificar el ancho a utilizar con cada item, y puedes seleccionar entre varias elecciones de estilo para números (tales como qué *"radix"* usar, si imprimir un exponente, si imprimir un signo, y cuando dígitos imprimir después del punto decimal). Haces esto especificando una cadena, llamada la ***cadena de formato***, la cual controla como y donde imprimir los otros argumentos.

Introducción a la sentencia printf

La sentencia printf presenta la siguiente sintáxis:

```
printf formato, item1, item2, ...
```

La lista completa de items podría ser opcionalmente encerrada entre paréntesis. Los paréntesis son necesarios si cualquiera de las expresiones item utiliza un operador relacional, de otra forma podría ser confundido con una redirección (Ver la sección Redireccionando la Salida de print y printf). Los operadores relacionales son `==', `!=', `<', `>', `>=', `<=', `~' y `!~' (Ver la sección Expresiones de Comparación)

La diferencia entre print y printf es el argumento *formato*. Éste es una expresión cuyo valor se toma como una cadena; su función es especificar como se deben imprimir cada uno de los otros argumentos. Se le llama la *cadena de formato*.

La cadena de formato es esencialmente que la que se usa para la función printf de la librería de C. La mayoría del formato es texto para que sea impreso literalmente. Pero en medio del texto que debe ser impreso literalmente

aparecen *especificadores de formato*, uno por item. Cada especificador de formato especifica el formato de salida del *item* o parámetro correspondiente

La sentencia printf no añade automáticamente un carácter newline a su salida. No imprime ninguna otra cosa que lo especificado en el formato. De forma que si quieres un carácter newline, debes incluir uno en el formato. Las variables de separación de la salida OFS y ORS no tienen efecto sobre las sentencias printf.

Letras para el control de formato

Un especificador de formato comienza con el carácter '%' y acaba con una *letra de control de formato*, y le indica a la sentencia printf como imprimir el item correspondiente (Si realmente quieres imprimir un carácter '%', escribe '%%'). La letra de control de formato especifica el tipo de valor a imprimir. El resto del especificador de formato está compuesto de *modificadores* opcionales los cuales son parámetros tales como el ancho de campo a usar.

Aquí tienes una lista de letras de control de formato:

`c'	Esto imprime un número como un carácter ASCII. Por lo que, `printf "%c", 65' imprimiría la letra 'A'. La salida para un valor cadena es el primer carácter de la cadena.
`d'	Esto imprime un entero decimal.
`i'	Esto también imprime un entero decimal.
`e'	Esto imprime un número en notación científica (exponencial). Por ejemplo, printf "%4.3e", 1950 imprime `1.950e+03', con un total de 4 cifras significativas de las cuales 3 siguen al punto decimal. Los *modificadores* `4.3' son descritos más abajo.
`f'	Esto imprime un número en notación punto flotante.
`g'	Esto imprime en notación científica o en notación punto flotante, la que quiera que sea más corta.
`o'	Esto imprime un entero octal sin signo.
`s'	Esto imprime una cadena.
`x'	Esto imprime un entero hexadecimal sin signo.
`X'	Esto imprime un entero hexadecimal sin signo. Sin embargo, para los valores entre 10 y 15, utiliza las letras desde la 'A' a la 'F' en lugar de esas mismas letras pero en minúsculas.
`%'	Esta no es realmente una letra de control de formato. Pero tiene un significado especial cuando se usa después de un '%': la secuencia '%%' imprime el carácter '%'. No consume ni necesita ningún item o argumento correspondiente.

Modificadores para los formatos de printf

Una especificación de formato también puede incluir modificadores que controlan como son impresos los valores de los items y cuanto espacio ocuparán. Los modificadores vienen entre el signo '%' y la letra de control de formato. Aquí están los posibles modificadores, en el orden en el cual podrían aparecer:

`-'	El signo menos, usado antes del modificador de ancho, especifica que se justifique a la izquierda el argumento dentro de la anchura especificada. Normalmente el argumento se ajusta a la derecha dentro del ancho especificado. Por lo que, `printf "%-4s", "foo"` imprime `foo '.
`*ancho*'	Este es un número que representa el ancho deseado para un campo. La inserción de cualquier número entre el signo '%' y el carácter de control de formato fuerza a que el campo se expanda a este ancho. El modo por defecto para hacer esto es rellenando con espacios en blanco por la izquierda. Por ejemplo, `printf "%4s", "foo"` imprime ` foo'. El valor de *ancho* es un ancho mínimo, no un máximo. Si el valor del item requiere más de *ancho* caracteres, podrá ser tan ancho como necesite. Por lo que, `printf "%4s", "foobar"` imprime `foobar'. Precediendo el *ancho* con un signo menos hace que la salida sea rellena con espacios en blanco por la derecha, en lugar de por la izquierda.
`.*precisión*'	Este es un número que especifica la precisión que se debe usar cuando se imprima. Esto especifica el número de dígitos que quieres imprimir a la derecha del punto decimal. Para una cadena, especifica el número máximo de caracteres que se imprimirán de dicha cadena. La capacidad de *ancho* y *precisión* dinámicos de la sentencia printf de la librería de C (por ejemplo, "%*.*s") todavía no está soportada. Sin embargo, se puede simular facilmente usando la concatenación para construir dinámicamente la cadena de formato.

Ejemplos de Uso de printf

Aquí tienes como usar printf para presentar una tabla alineada:

```
awk '{ printf "%-10s %s\n", $1, $2 }' Lista-BBS
```

imprime los nombres de los bulletin boards ($1) del fichero 'Lista-BBS' como una cadena de 10 caracteres justificados a la izquierda. También imprime los números de teléfono ($2) a continación en la línea. Esto produce una tabla alineada de dos columnas de nombres y números de teléfonos:

```
aardvark   555-5553
alpo-net   555-3412
barfly     555-7685
bites      555-1675
camelot    555-0542
core       555-2912
fooey      555-1234
foot       555-6699
macfoo     555-6480
sdace      555-3430
```

```
sabafoo     555-2127
```

¿Te diste cuenta de que no especificamos que se imprimiesen los números de teléfono como números? Tienen que ser impresos como cadenas debido a que están separados por medio con el guión. Este guión podría ser interpretado como un signo menos si hubiesemos intentado imprimir los números de teléfono como números. Esto nos hubieses producido unos resultados confusos y erróneos.

No hemos especifado un ancho para los números de teléfono porque son lo último que se imprimirá en cada línea. No necesitamos poner espacios después de ellos.

Podríamos haber hecho nuestra tabla más elegante incluso añadiéndole cabeceras encima de cada una de las columnas. Para hacer esto, usa el patrón BEGIN(Ver la sección Los Patrones Especiales BEGIN y END) para hacer que se imprima la cabecera solamente una vez, al principio del programa awk:

```
awk 'BEGIN { print "Nombre    Número"
             print "------    ------" }
     { printf "%-10s %s\n", $1, $2 }' Lista-BBS
```

¿Te diste cuenta de que mezclamos las sentencias print y printf en el ejemplo anterior? Podríamos haber utilizado solamente sentencias printf para obtener el mismo resultado:

```
awk 'BEGIN { printf "%-10s %s\n", "Name", "Number"
             printf "%-10s %s\n", "----", "------" }
     { printf "%-10s %s\n", $1, $2 }' Lista-BBS
```

Poniendo cada cabecera de columna con la misma especificación de formato usada para los elementos, nos hemos asegurado que las cabeceras tendrán la misma alineación que las columnas. El hecho de que la misma especificación de formato se use tres veces, puede ser resumido almacenando dicha especificación de formato en una variable, tal y como sigue:

```
awk 'BEGIN { format = "%-10s %s\n"
             printf format, "Name", "Number"
             printf format, "----", "------" }
     { printf format, $1, $2 }' Lista-BBS
```

Redireccionando la Salida de print y printf

Hasta ahora hemos estado tratando solamente con salida que imprime en la salida estándar, normalmente tu terminal. Se les puede decir tanto a print como a printf que envíen su salida a otros sitios. Esto recibe el nombre de *redirección*.

Una redirección aparece después de la sentencia print o printf. Las redirecciones en awk son escritas del mismo modo que se hacen las redirecciones en los comandos de la shell, excepto que son escritas dentro del programa awk.

Redireccionando la Salida a Ficheros y Pipes

Aquí aparecen las tres formas de la redirección de la salida. Ellas se muestran todas para la sentencia print, pero funcionan de forma idéntica para printf.

print items > fichero-salida

Este tipo de redirección imprime los items en el fichero de salida *fichero-salida*. El nombre del fichero *fichero-salida* puede ser cualquier expresión. Su valor se convierte a cadena y después es usada como nombre de fichero. (Ver la sección 8. Acciones: Expresiones).

Cuando se usa este tipo de redirección, el *fichero-salida* es eliminado antes de que se escriba en él la primera salida. Las escrituras siguientes no borrarán el *fichero-salida*, pero sí añadirán a él. Si *fichero-salida* no existe entonces será creado.

Por ejemplo, aquí aparece como un programa awk puede escribir una lista de nombres BBS a un fichero 'lista-nombres' y una lista de números de teléfonos a un fichero 'lista-teléfonos'. Cada fichero de salida contiene un nombre o número por línea.

```
awk '{ print $2 > "lista-teléfonos"
       print $1 > "lista-nombres" }' Lista-BBS
```

print items >> fichero-salida

Este tipo de redirección imprime los items en el fichero de salida *fichero-salida*. La diferencia entres este y el signo '>' único es que los viejos contenidos (si los tuviese) del fichero *fichero-salida* no son eliminados. En su lugar, la salida que genera awk es añadida a dicho fichero.

print items | comando

También es posible enviar la salida a través de un pipe en lugar de a un fichero. Este tipo de redireccionamiento abre un pipe a *comando* y escribe los valores de *items* a través de este pipe, a otro proceso creado para ejecutar *comando*.

El argumento de redirección *comando* es realmente una expresión awk. Su valor se convierte a cadena, cuyo contenido nos proporcina el comando de shell que debe ser ejecutado.

Por ejemplo, este produce dos ficheros, una lista sin ordenar de nombres BBS y una lista ordenada en orden alfabético inverso:

```
awk '{ print $1 > "names.unsorted"
       print $1 | "sort -r > names.sorted" }' Lista-BBS
```

Aquí la lista desordenada se escribe con una redirección normal mientras que la lista ordenada es escrita mediante un pipe al comando sort de Unix.

Aquí tienes un ejemplo que usa el redireccionamiento para enviar un mensaje a una lista de correos 'bug-system'. Esto podría ser útil cuando se encuentran problemas en un script awk que se ejecuta periódicamente para el mantenimiento del sistema.

```
print "Awk script failed:", $0 | "mail bug-system"
print "at record number", FNR, "of", FILENAME  | "mail bug-system"
close("mail bug-system")
```

Llamamos a la función close aquí porque es una buena idea cerrar la tubería o pipe tan pronto como toda la salida ha pasado a través de ella.

El redireccionamiento de la salida us ando `>', `>>', o `|' le pide al sistema que abra un fichero o pipe solo si el fichero o comando particular que has especificado no ha sido ya escrito por tu programa.

Cerrando los Ficheros de Salida y Pipes

Cuando se abre un fichero o pipe, el nombre de fichero o comando asociado con él es recordado por awk y las siguientes escrituras al mismo fichero o comando son añadidas a las escrituras previas. El fichero o pipe permanece abierto hasta que finaliza el programa awk. Esto es normalmente conveniente.

Algunas veces existe una razón para cerrar un fichero de salida o un pipe antes de que finalice el programa awk. Para hacer esto, utilice la función *close*, tal y como sigue:

```
close(nombre-fichero) o close(comando)
```

El argumento *nombre-fichero* o *comando* puede ser cualquier expresión. Su valor debe concordar exactamente con la cadena usada para abrir el fichero o pipe cuando se empezó a usar – por ejemplo, si abres un pipe con esto:

```
print $1 | "sort -r > names.sorted"
```

entonces debes cerrarlo con esto:

```
close("sort -r > names.sorted")
```

Aquí están algunas razones por las cuales podrías necesitar cerrar un fichero de salida:

- Para escribir un fichero y leer el mismo posteriormente en el mismo programa awk. Cierra el fichero cuando hayas acabado de escribir en él; entonces ya puedes empezarlo a leer con *getline*.
- Para escribir numerosos ficheros, sucesivamente, en el mismo programa awk. Si no cierras los ficheros, eventualmente excederás el límite del sistema en el número de ficheros abiertos por un proceso. Así que cierra cada uno cuando hayas acabado de escribirlo.
- Para hacer que un comando acabe. Cuando redireccionas la salida a través de un pipe, el comando que lee del pipe normalmente sigue intentando leer entrada mientras el pipe esté abierto. A menudo esto significa que el comando no puede realizar su trabajo hasta que el pipe es cerrado. Por ejemplo, si redireccionas la salida al programa mail, el mensaje no se enviará realmente hasta que el pipe se cierre.
- Para ejecutar el mismo programa una segunda vez, con los mismos argumentos. Esto no es la misma cosa que darle más entrada a la primera ejecución.

Por ejemplo, supón que haces un pipe de la salida al programa *mail*. Si sacas varias líneas redirigidas a este pipe sin cerrarlo, crear un mensaje de varias líneas. En constraste, si cierras el pipe después de cada línea de salida, entonces cada línea creara un correo distinto.

Streams de Entrada/Salida Estándard

La ejecución convencional de programa tiene tres streams de entrada y salida disponibles para lectura y escritura. Estos son conocidos como la *entrada estándar, salida estándar y salida del error estándar*. Estos streams son, por defecto, entrada y salida por terminal, pero son a menudo redireccionados con el shell, a través de los operadores `<', `<<', `>', `>>', `>&' y `|'. El error estándar se usa solamente para la escritura de mensajes de error; la razón por la cual tenemos dos streams separadas, salida estándar y error estándar, es para que puedan ser redireccionados

independientemente. En otras implementaciones de awk, la única forma de escribir un mensaje de error al error estándar en un programa awk es la siguiente:

```
print "Serious error detected!\n" | "cat 1>&2"
```

Esto trabaja abriendo un pipeline a un comando del shell el cual puede acceder a la stream del error estándar. Esto está lejos de ser elegante, y también es ineficiente, ya que requiere un proceso separado. Por lo que la gente que escribe programas awk se han negado a menudo a hacer esto. En su lugar, han enviado los mensajes de error al terminal, tal y como sigue:

```
NF != 4 {
   printf("line %d skipped: doesn't have 4 fields\n", FNR) > "/dev/tty"
}
```

Esto tiene el mismo efecto la mayoría de las veces, pero no siempre: aunque el stream del error estandárd es normalmente el terminal, puede ser redireccionado, y cuando eso ocurre, la escritura al terminal no es lo correcto. De hecho, si awk se ejecuta desde un trabajo en background, podría no tener un terminal para poner la salida de dicho error. Entonces, abrir el dispositivo `/dev/tty' fallaría.

Gawk proporciona nombres de ficheros especiales para acceder a los tres streams estándar. Cuando redireccionas la entrada o la salida en gawk, si el nombre de fichero encaja con uno de estos nombres especiales, entonces gawk utiliza directamente el stream con el que se corresponde.

`/dev/stdin'	La entrada estándar (descriptor de fichero 0).
`/dev/stdout'	La salida estándar (descriptor de fichero 1).
`/dev/stderr'	La salida del error estándar (descriptor de fichero 2).
`/dev/fd/*n*'	El fichero asociado con el descriptor *n*. Tal fichero debería haber sido abierto por el programa que inicie la ejecución awk (normalmente el shell). A menos que hagas algo para que no sea así, solamente los descriptores 0, 1 y 2 están disponibles.

Los nombres de ficheros `/dev/stdin', `/dev/stdout', y `/dev/stderr' son alias para `/dev/fd/0', `/dev/fd/1', y `/dev/fd/2', respectivamente, pero los primeros nombres son más indicativos.

La forma más correcta para escribir un mensaje de error en un programa awk es usar `/dev/stderr', algo como esto:

```
NF != 4 {
  printf("line %d skipped: doesn't have 4 fields\n", FNR) > "/dev/stderr"
}
```

El reconocimiento de estos nombres de fichero especiales es deshabilitado si awk está en modo compatibilidad (Ver la sección 14. Invocación de awk)

5. Programas de “Una línea” útiles

Programas de awk útiles son a menudo cortos, de una sola línea o dos. Aquí tienes una colección de programas cortos y útiles para empezar. Algunos de estos programas contienen construcciones que no han sido explicadas todavía. La descripción del programa te dará una idea de qué es lo que hace, pero por favor lee el resto del manual para convertirte en un experto de awk.

```
awk '{ num_campos = num_campos + NF }

END { print num_campos }'
```

Este programa imprime el número total de campos de todas las líneas de entrada.

```
awk 'length($0) > 80'
```

Este programa imprime todas las líneas que tengan más de 80 caracteres. La única regla tiene una expresión relacional como su patrón, y no tiene ninguna acción (así que la acción por defecto, imprimir el registro, es la que se usa)

```
awk 'NF > 0'
```

Este programa imprime todas las líneas que tienen al menos un campo. Esta es una forma fácil de eliminar líneas en blanco de un fichero (o en su lugar, crear un nuevo fichero similar al fichero anterior pero en el cual se han suprimido las líneas en blanco)

```
awk '{ if (NF > 0) print }'
```

Este programa también imprime todas las líneas que tienen al menos un campo. Aquí nosotros aplicamos la regla para buscar la línea y a continuación se decide en la acción donde imprimir.

```
awk 'BEGIN { for (i = 1; i <= 7; i++)

print int(101 * rand()) }'
```

Este programa imprime 7 números aleatorios desde 0 a 100, ambos incluidos.

```
ls -l ficheros | awk '{ x += $4 } ; END { print "total bytes: " x }'
```

Este programa imprime el número total de bytes utilizados por *ficheros*.

```
expand file | awk '{ if (x < length()) x = length() }

END { print "maximum line length is " x }'
```

Este programa imprime la máxima longitud de línea de fichero. El fichero de entrada que va a procesar awk es filtrado previamente con el comando expand que cambia los tabuladores por espacios en blanco, así que las longitudes comparadas son realmente las columnas del margen derecho.

6. Patrones

Los patrones en awk controlan la ejecución de reglas: una regla es ejecutada cuando su patrón concuerda (matches) con el registro de entrada actual. Este capítulo habla sobre como escribir patrones.

Tipos de Patrones

Aquí se presenta un sumario de los tipos de patrones soportados en awk.

/expresión regular/	Una expresión regular como patrón. Encaja cuando el texto del registro de entrada concuerda con la expresión regular. (Ver la sección Expresiones Regulares como Patrones).
expresión	Una expresión solamente. Casa cuando su valor, convertido a número, es distinto de cero (si es un número) o no nulo (si es una cadena). (Ver la sección Expresiones como Patrones).
pat1, pat2	Un par de patrones separados por una coma, especificando un rango de registros. (Ver la sección Especificando Rangos de Registros con Patrones).
BEGIN **END**	Patrones especiales para suministrar a awk información para antes del inicio del procesamiento o al final del procesamiento. (Ver la sección Los Patrones Especiales BEGIN y END).
null	El patrón vacío casa con todos y cada uno de los registros de entrada. (Ver la sección El Patrón Vacío).

El Patrón Vacío

Un patrón vacío se considera que casa con todos los registros de la entrada. Por ejemplo, el programa:

```
awk '{ print $1 }' Lista-BBS
```

imprime solamente el primer campo de cada registro.

Expresiones Regulares como Patrones

Una *expresión regular*, o *expreg*, es una forma de describir una clase de cadenas de texto. Una *expresión regular* encerrada entre barras (‘/’) es un patrón awk que casa con cada registro de entrada cuyo texto pertenece a esa clase.

La expresión regular más simple es una secuencia de letras, números, o ambos. Tales expresiones regulares casa con cualquier cadena que contenga dicha secuencia de letras y números. Por lo que, la expresión regular ‘foo’ casa con cualquier cadena que contenga ‘foo’. Por lo tanto, el patrón /foo/ casa con cualquier registro que contenga la cadena ‘foo’. Otros tipos de expresiones regulares te permiten expecificar clases de cadenas más complicadas.

Cómo usar Expresiones Regulares

Una expresión regular puede ser usada como un patrón encerrandola entre barras. Entonces la expresión regular es comparada contra todo el texto de cada registro. (Normalmente, solo hace falta que case parte del texto para que el casamiento tenga éxito). Por ejemplo, esto imprime el segundo campo de cada registro que contiene 'foo' en cualquier parte del registro:

```
awk '/foo/ { print $2 }' Lista-BBS
```

Las expresiones regulares pueden también ser usadas en expresiones de comparación. Entonces puedes especificar la cadena contra la cual casarla; no necesita que case el registro de entrada actual completamente. Estas expresiones de comparación pueden ser usadas como patrones o en sentencias *if* y *while*.

exp ~ /regexp/

Esto es cierto si la expresión *exp* (tomada como una cadena de caracteres) encaja con *regexp*. Los siguientes ejemplo encajan o seleccionan todos los registros de entrada que contengan la letra 'J' en cualquier posición dentrl del primer campo:

```
awk '$1 ~ /J/' inventario-enviado
```

o sino:

```
awk '{ if ($1 ~ /J/) print }' inventario-enviado
```

exp !~ /regexp/

Esto es cierto si la expresión *exp* (tomada como una cadena de caracteres) no concuerda con *regexp*. El siguiente ejemplo encaja o selecciona todos los registros de entrada cuyo primer campo no contiene la letra 'J' en ninguna posición:

```
awk '$1 !~ /J/' inventario-enviado
```

La parte derecha de un operador `~' o `!~' no necesita ser una expresión regular constante (por ejemplo, una cadena de caracteres entre barras). Podría ser cualquier expresión. La expresión es evaluada, y convertida si es necesario a una cadena; los contenidos de la cadena serán utilizados como expresión regular. Una expresión regular que se comporta de esta forma es llamada ***expresión regular dinámica***. Por ejemplo:

```
identifier_regexp = "[A-Za-z_][A-Za-z_0-9]+"
$0 ~ identifier_regexp
```

fija la variable `identifier_regexp` a una expresión regular que describe nombres de variables de awk, y chequea si el registro de entrada encaja con esta expresión regular.

Operadores de Expresiones Regulares

Puedes combinar expresiones regulares con los siguientes caracteres, llamados *operadores de expresiones regulares*, o *metacaracteres*, para incrementar el poder y versatilidad de las expresiones regulares.

Aquí tienes una tabla de metacaracteres. Todos los caracteres que no aparecen no tienen ningún significado especial en una expresión regular.

^

Esto busca el principio de la cadena o el principio de una línea dentro de la cadena. Por ejemplo:

```
^@capítulo
```

coincide con '@capítulo' al principio de una cadena, y puede ser usada para identificar comienzos de capítulos en ficheros fuentes Texinfo.

$

Esto es similar a '~', pero encaja solo al final de una cadena o el final de una línea dentro de la cadena. Por ejemplo:

```
p$
```

encaja con un registro que acabe en 'p'.

.

Esto encaja con cualquier carácter único, excepto el carácter nueva línea. Por ejemplo:

```
.P
```

encaja con cualquier carácter que vaya seguido por una 'P' en una cadena. Usando la concatenación podemos hacer expresiones regulares como 'U.A', la cual encaja con cualquier secuencia de 3 caracteres que comiencen con una 'U' y acaben con 'A'.

[...]

Esto recibe el nombre de conjunto de caracteres. Encaja con cualquiera de los caracteres encerrados entre los corchetes. Por ejemplo:

```
[MVX]
```

encaja con cuales quiera de los caracteres 'M', 'V', y 'X' en una cadena.

Se pueden especificar rangos de caracteres utilizando un guión entre el carácter de inicio y el carácter final del intervalo de caracteres, y encerrando entre los corchetes. Por ejemplo:

```
[0-9]
```

encaja con cualquier dígito.

Para incluir cualquiera de estos caracteres `\', `]', `-' o `^' en un conjunto de caracteres, pon un '\' antes del carácter en cuestión. Por ejemplo:

```
[d\]]
```

encaja con ']' o 'd'.

Este tratamiento de '\' es compatible con otras implementaciones de awk pero incompatible con la expecificación POSIX propuesta para awk. El borrador actual especifica el uso de la misma sintáxis usada en egrep.

Podríamos cambiar gawk para que encajase con el estándar, una vez que estuviésemos seguros de que no volverá a cambiar. Mientras tanto, la opción '-a' especifica la sintaxis de awk tradicional descrita anteriormente (la cual es también la sintaxis por defecto), mientras que la opción '-e' especifica la sintaxis egrep. (Ver la sección Opciones de la línea de comandos).

En la sintáxis egrep, la barra invertida no tiene sintácticamente significado especial dentro de las llaves. Esto significa que tienen que ser usados trucos especiales para representar los caracteres `]', `-' y `^' como miembros de un conjunto de caracteres.

Para buscar un coincidencia del carácter `-', escríbelo como `---', lo cual es un rango que contiene únicamente el carácter `-'. Podrías también dar `-' como el primer o el último carácter del conjunto. Para buscar una coincidencia del carácter `^', ponlo en cualquier lugar excepto como el primer carácter de un conjunto. Para buscar una coincidencia del carácter `]', haz que sea el primer carácter del conjunto. Por ejemplo:

```
[]d^]
```

encaja con `]', `d' o `^'.

[^ ...]

Esto es el *conjunto de caracteres complementario*. El primer carácter después del '[' debe ser un '^'. Encaja con cuales quiera caracteres *excepto* aquellos que se meten entre los corchetes. Por ejemplo:

```
[^0-9]
```

encaja con cualquier carácter que no sea un dígito.

|

Este es el operador *alternación* y se usa para especificar alternativas. Por ejemplo:

```
^P|[0-9]
```

encaja con cualquier cadena que encaje con `^P' o con `[0-9]'. Esto significa que encaja con cualquier cadena que contenga un dígito o comience por 'P'.

La alternación se aplica a la expresión regular posible más grande a cada lado.

(...)

Los paréntesis se usan para agrupar expresiones regulares del mismo modo que en las aritméticas. Pueden ser usados para concatenar expresiones regulares que contengan el operador alternacion '|'.

*****

Este símbolo significa que la expresión regular precedente se va a repetir tantas veces como sea posible para encontrar una concordancia. Por ejemplo:

```
ph*
```

aplica el símbolo '*' a la 'h' precedente y busca la concordancia de una 'p' seguida por cualquier número de haches. También encajará con una única 'p' que no vaya seguida por ninguna 'h'.

El '*' repite la expresión precedente más pequeña posible. (use paréntesis si deseas repetir una expresión más grande) Encuentra tantas repeticiones como sea posible. Por ejemplo:

```
awk '/\(c[ad][ad]*r x\)/ { print }' sample
```

imprime cada registro de la entrada que contenga cadenas de la siguiente forma `(car x)', `(cdr x)', `(cadr x)', y así sucesivamente.

+

Este símbolo es similar a '*', pero la expresión precedente debe encajar al menos una vez. Esto significa que:

```
wh+y
```

encajará con 'why' y 'whhy' pero no con 'wy', donde sin embaro 'wh*y' encajaría con las tres combinaciones. Esta es una forma más simple de escribir el último ejemplo '*':

```
awk '/\(c[ad]+r x\)/ { print }' sample
```

?

Este símbolo es similar a '*', pero la expresión regular precedente puede encajar una vez o ninguna. Por ejemplo:

```
fe?d
```

encajará con 'fed' o 'fd' y nada más.

****

Esto se usa para suprimir el significado especial de un carácter en un matching. Por ejemplo:

```
\$
```

busca una coincidencia del carácter `$'.

Las secuencias escape usadas para constantes de cadena (Ver la sección Expresiones Constantes) son válidas en expresiones regulares también; ellas son también precedidas por un '\'.

En las expresiones regulares, los operadores '*', '+' y '?' tienen la precedencia mayor, seguidos por la concatenación, y finalmente por '|'. Como en las aritméticas, los paréntesis pueden cambiar el modo en que se agrupen los operadores.

Sensibilidad a Mayúsculas en el Matching

La sensibilidad a mayúsculas es normalmente significativa en expresiones regulares, también cuando casa caracteres ordinarios (por ejemplo, no con metacaracteres), y dentro de conjunto de caracteres. Por lo que una 'w' en una expresión regular encaja solamente con una 'w' minúscula y nunca con una 'W' mayúscula.

La forma más fácil de hace una búsqueda o encaje insensible a mayúsculas es usar el conjunto de caracteres: '[Ww]'. Sin embargo, esto puede ser embarazoso si necesitas usarlo a menudo; y puede hacer que la expresión regular sea más ilegible. Existen otras dos alternativas que podrías preferir.

Una forma de realizar una búsqueda insensible a mayúsculas en un punto en particular del programa es convertir el dato a un único tipo (mayúsculas o minúsculas), usando las funciones empotradas para manejo de cadenas *tolower* o *toupper* (que serán explicadas posteriormente en la Sección Funciones Implícitas (Built-in) para Manipulación de Cadenas). Por ejemplo:

```
tolower($1) ~ /foo/  { ... }
```

convierte el primer campo a minúsculas antes de realizar una búsqueda o comparación contra dicho campo.

Otro método es fijar la variable `IGNORECASE` a un valor distinto de cero (ver sección 13. Variables Implícitas (Built-in). Cuando `IGNORECASE` es distinta de cero, todas las operaciones de expresiones regulares son insensibles a mayúsculas/minúsculas. Cambiando el valor de la variable `IGNORECASE` se puede controlar de forma dinámica la

sensibilidad a mayúsculas/minúsculas de tu programa en tiempo de ejecución. La sensibilidad a mayúsculas/minúsculas está activada por defecto porque la variable IGNORECASE (como la mayoría de las variables) es inicializada a cero.

```
x = "aB"
if (x ~ /ab/) ...   # esta comprobación fallará
IGNORECASE = 1
if (x ~ /ab/) ...   # esta comprobación tendrá éxito
```

Normalmente no puedes usar IGNORECASE para hacer que ciertas reglas sean insensibles a mayúsculas/minúsculas y otras reglas sean sensibles, porque no hay forma de fijar el valor de IGNORECASE solamente para el patrón de una regla en particular. Para hacer esto, debes usar los conjuntos de caracteres o *tolower*. Sin embargo, una cosa que puedes hacer solamente con IGNORECASE es activar o desactivar la sensibilidad a mayúsculas/minúsculas para todas las reglas de una vez.

IGNORECASE puede ser fijado en la línea de comando, o en una regla BEGIN. Fijar la variable IGNORECASE en la línea de comando puede ser una forma de hacer a un programa insensible a las mayúsculas/minúsculas sin tener que editarlo.

El valor de IGNORECASE no tiene ningún efecto si gawk está en modo compatibilidad (Ver la sección 14. Invocación de awk). En modo compatibilidad la diferencia entre mayúsculas y minúsculas es siempre significativa.

Expresiones de Comparación como Patrones

Los *patrones de comparación* chequean relaciones tales como igualdad entre dos cadenas o números. Son un caso especial de patrones de expresiones (ver la Sección Expresiones como Patrones). Ellos son escritos con *operadores relacionales*, los cuales son un superconjunto de los mismos en C. Aquí tienes una tabla de ellos:

x < y	Verdad si x es menor que y.
x <= y	Verdad si x es menor o igual que y.
x > y	Verdad si x es mayor que y.
x >= y	Verdad si x es mayor o igual que y.
x == y	Verdad si x es igual a y.
x != y	Verdad si x no es igual a y.
x ~ y	Verdad si x encaja con la expresión regular descrita por y.
x !~ y	Verdad si x no encaja con la expresión regular descrita por y.

Los operandos de un operador relacional son comparados como números si ambos son números. Si no ellos son convertidos y comparados como cadenas (ver la sección Conversiones de Cadenas y Números). Las cadenas son comparadas comparando el primer carácter de ambas, después el segundo carácter de ellas y así sucesivamente, hasta que encuentre una diferencia. Si las dos cadenas son iguales hasta que se acaba la cadena más corta, la cadena más corta se considera menor que la más larga. Por lo que “10” es menor que “9”.

El operando izquierdo de los operadores `~' y `!~' es una cadena. El operador derecho es o una expresión regular constante encerrada entre barras (*/expregl/*), o cualquier expresión, cuyo valor como cadena se usa como una expresión regular dinámica (Ver la sección Cómo usar Expresiones Regulares)

El siguiente ejemplo imprime el segundo campo de cada registro de entrada cuyo primer campo valga justamente 'foo'.

```
awk '$1 == "foo" { print $2 }' Lista-BBS
```

Constrasta esto con el siguiente encaje de expresión regular, el cual aceptaría cualquier registro con un primer campo que contuviese la cadena 'foo'.

```
awk '$1 ~ "foo" { print $2 }' Lista-BBS
```

o, equivalentemente, este:

```
awk '$1 ~ /foo/ { print $2 }' Lista-BBS
```

Operadores Booleanos como Patrones

Un *patrón booleano* es una expresión la cual combina otros patrones usando los *operadores booleanos* "o" ('||'), "y" ('&&') y "not" ('!'). Donde los patrones booleanos encajan con un registro de entrada dependiendo de si encajan o no los subpatrones.

Por ejemplo, el siguiente comando imprime todos los registros en el fichero de entrada 'Lista-BBS' que contengan tanto '2400' como 'foo'.

```
awk '/2400/ && /foo/' Lista-BBS
```

El siguiente comando imprime todos los registros en el fichero de entrada 'Lista-BBS' que contengan '2400' o 'foo' o ambos.

```
awk '/2400/ || /foo/' Lista-BBS
```

El siguiente comando imprime todos los registros del fichero de entrada 'Lista-BBS' que no contengan la cadena 'foo'.

```
awk '! /foo/' Lista-BBS
```

Dese cuenta de que los patrones booleanos son un caso especial de los patrones de expresión (Ver la sección Expresiones como Patrones); son expresiones que usan los operadores booleanos. Ver la sección Expresiones Booleanas, para una información completa sobre los operadores booleanos.

El subpatrón de un patrón booleano pueden ser expresiones regulares constantes, o cualquier otra expresión gawk. Los patrones de rango no son expresiones, por lo que no pueden aparecer dentro de los patrones booleanos. Del mismo modo, los patrones especiales BEGIN y END, los cuales nunca encajan con ningún registro de entrada, no son expresiones y no pueden aparecer dentro de los patrones booleanos.

Expresiones como Patrones

Cualquier expresión awk es también válida como un patrón en gawk. Entonces el patrón "encaja" si el valor de la expresión es distinto de cero (si es un número) o distinto de nulo (si es una cadena).

La expresión es evaluada de nuevo cada vez que la regla se chequea contra un nuevo registro de entrada. Si la expresión utiliza campos tales como $1, el valor depende directamente del texto del registro de entrada nuevo, de otra

forma, depende solo de lo que haya ocurrido hasta el momento en la ejecución del programa awk, pero eso aún podría ser útil.

Los patrones de comparación son realmente un caso especial de este. Por ejemplo, la expresión $5 == “foo” tiene el valor 1 cuando el valor de $5 es igual a “foo”, y en cualquier otro caso 0; por lo tanto, esta expresión como patrón encaja cuando los dos valores son iguales.

Los patrones booleanos son también casos especiales de los patrones de expresión.

Una expresión regular constante como patrón es también un caso especial de un patrón de expresión. /foo/ encaja con cualquier registro que contenga ‘foo’.

Otras implementaciones de awk son menos generales que gawk: permiten expresiones de comparación, y combinaciones booleanas (opcionalmente con paréntesis), pero no necesariamente otros tipos de expresiones.

Especificando Rangos de Registros con Patrones

Un *rango de patrones* está formado por dos patrones separados por una coma, de la forma *patroninicio, patronfinal*. Encaja con rangos de registros de entrada consecutivos. El primer patrón *patroninicio* controla donde comienza el rango, y el segundo *patrónfinal* controla donde acaba. Por ejemplo,

```
awk '$1 == "on", $1 == "off"'
```

imprime todos los registros entre ‘on’/’off’, ambos incluídos.

En más detalle, un patrón de rango comienza chequeando el *patróninicio* contra los registros de entrada; cuando un registro concuerda con *patróninicio*, se activa el patrón de rango y te sacará todos los registros de entrada hasta que encuentra una línea que encaja con el *patrónfinal*. A continuación vuelve a buscar el *patróninicio* a partir de la siguiente línea a la línea que concordó con el *patrónfinal* y así sucesivamente.

El registro que activa el rango de patrones y el que lo desactiva, ambos concuerdan y son incluidos dentro del patrón. Si no se desean que se operen o tengan en cuenta dichos patrones, puedes escribir una sentencia if en la acción de la regla para discriminarlos.

Es posible que exista un mismo registro que sea el que active y desactive el rango de patrón, si ambas condiciones son satisfechas por dicho registro. Entonces la acción se ejecuta solamente para dicho registro.

Los Patrones Especiales BEGIN y END

BEGIN y END son patrones especiales. Ellos no son usados para encajar con registros de entrada. En su lugar, ellos son usados para suministrar al script awk qué hacer antes de empezar a procesar y después de haber procesado los registros de la entrada. Una regla BEGIN se ejecuta una vez, antes de leer el primer registro de entrada. Y la regla END se ejecuta una vez después de que se hayan leído todos los registros de entrada. Por ejemplo:

```
awk 'BEGIN { print "Análisis de `foo'" }
     /foo/ { ++foobar }
     END   { print "`foo' aparece " foobar " veces." }' Lista-BBS
```

Este programa averigua cuantas veces aparece la cadena ‘foo’ en el fichero de entrada ‘Lista-BBS’. La regla BEGIN imprime un título para el informe. No hay necesidad de usar la regla BEGIN para inicializar el contador foobar a cero, ya que awk lo hace por nosotros automáticamente (ver la sección Variables).

La segunda regla incrementa el valor de la variable foobar cada vez que se lee de la entrada un registro que contiene el patrón 'foo'. La regla `END` imprime el valor de la variable foobar al final de la ejecución.

Los patrones especiales `BEGIN` y `END` no pueden ser usados en rangos o con operadores booleanos. Un programa awk podría tener múltiples reglas `BEGIN` y/o `END`. Ellas son ejecutadas en el mismo orden que aparecen, todas las reglas `BEGIN` al principio y todas las reglas `END` al final.

Las secciones `BEGIN` y `END` múltiples pueden ser útiles para escribir funciones de librería, ya que cada librería puede tener su propias reglas `BEGIN` y `END` para hacer sus propias inicializaciones y/o limpieza. Avertirle que el orden en el cual las librerías son nombradas en la línea de comandos controla el orden en el que son ejecutadas las reglas `BEGIN` y `END`. Por lo tanto tienes que tener cuidado cuando escribas tales reglas en ficheros de librerías para que te dé igual el orden en el que las incluyas en la ejecución. Ver la sección 14. Invocación de awk para más información acerca del uso de funciones de librería.

Si un programa awk tiene solamente una regla `BEGIN`, y ninguna otra regla, entonces el programa se sale una vez que se ha ejecutado la regla `BEGIN`, sin llegar a leer los registros del fichero de entrada. Sin embargo, si existe también una regla `END`, entonces se leerán los registros de la entrada, incluso aunque no haya ninguna otra regla en el programa. Esto es necesario en caso de que la regla `END` chequee el valor de la variable empotrada `NR` (número de registros leídos).

Las reglas `BEGIN` y `END` deben tener acciones; no existen acciones por defecto para estas reglas, ya que no hay ningún registro actual cuando son ejecutadas.

7. Acciones: Overview

Un programa o script awk consiste en una serie de *reglas* y definiciones de funciones. (Las funciones son descritas posteriormente. Ver la sección 12. Funciones definidas por el usuario).

Una *regla* (rule) contiene un *patrón* y una *acción*, y cualquiera de las dos puede ser omitida. El propósito de la acción es decirle a awk qué tiene que hacer una vez que se encuentra el patrón en un registro de entrada. Por lo que el programa completo parecería algo como esto:

```
[patrón] [{ acción }]
[patrón] [{ acción }]
...
function nombre_función (argumentos) { ... }
...
```

Una acción consiste en una o más sentencias awk, encerradas entre llaves (`{' y `}'). Cada sentencia especifica una cosa que se tiene que hacer. Las sentencias son separadas por caracteres de nueva línea o puntos y comas (si hay más de una sentencia en una misma línea)

Las llaves alrededor de una acción deben ser usadas incluso si la acción contiene una sola sentencia, o si no contiene ninguna sentencia. Sin embargo, si omites la acción por completo, omite también las llaves. (La omisión de la acción es equivalente a `{ print $0 }').

Aquí están los tipos de sentencias soportadas en awk:

- Expresiones, las cuales pueden llamar a funciones o asignar valores a variables (ver la sección 8. Acciones: Expresiones). La ejecución de este tipo de sentencias simplemente calcula el valor de la expresión y después lo ignora. Esto es útil cuando la expresión tiene efectos laterales (ver la sección Expresiones de Asignación)
- Sentencias de control, las cuales especifican el flujo de control del programa awk. El lenguaje awk te da construcciones similares a las del Lenguaje C (if, for, while, etc.) y otras pocas especiales (Ver la sección 9. Acciones: Sentencias de Control).
- Sentencias compuestas, las cuales consisten en una o más sentencias encerradas entre llaves. Una sentencia compuesta se usa para poner varias sentencias juntas en el cuerpo de una sentencia if, while, do o for .
- Control de la entrada, usando la función getline (Ver la sección Entrada explícita con getline), y la sentencia next (Ver la sección La Sentencia next).
- Sentencias de Salida, print y printf. (Ver la sección 4. Imprimiendo la salida).
- Sentencias de borrado, para eliminar elementos de un array. (Ver la sección La Sentencia delete).

Los dos próximos capítulos cubren en detalles las sentencias de control y las expresiones respectivamente. Vamos a tratar también los arrays, funciones implícitas (built-in) que son utilizados en las expresiones. Después procederemos a explicar como definir tus propias funciones.

8. Acciones: Expresiones

Las expresiones son los bloques de construcción básicos de las acciones awk. Una expresión se evalua a un valor, el cual puedes imprimir, testear, almacenar en una variable o pasar a una función.

Pero, más allá de eso, una expresión le puede asignar un nuevo valor a una variables o un campo, con un operador de asignación. Una expresión puede servir como una sentencia en si misma. La mayoría de los otros tipos de sentencias contienen una o más expresiones las cuales especifican los datos sobre los que se van a operar. Al igual que en otros lenguajes, las expresiones en awk incluyen variables, referencias a arrays, constantes, y llamadas a función, al igual que combinaciones de estas con varios operadores.

Expresiones Constantes

El tipo de expresión más simple es la constante, la cual tiene siempre el mismo valor. Existen tres tipos de constantes: constantes numéricas, constantes de cadenas, y constantes de expresiones regulares.

Una constante numérica representa un número. Este número puede ser un entero, una fracción decimal, o un número en notación científica (exponencial). Dese cuenta que todos los valores numéricos están representados en awk como doble-precisión en punto flotante. Aquí se presentan algunos ejemplos de constantes numéricas, las cuales tienen todas el mismo valor:

```
105
1.05e+2
1050e-1
```

Una constante cadena consiste en una secuencia de caracteres encerrados entre dobles comillas. Por ejemplo:

"loro"

representa la cadena cuyo contenido es ‘loro’. Las cadenas en gawk pueden ser de cualquier longitud y pueden contener cualquier carácter ASCII de 8-bits incluído el ASCII NUL. Otras implementaciones de awk podrían tener dificultad con algunos códigos de carácter.

Algunos caracteres no pueden ser incluídos literalmente en una constante cadena. Las representas en su lugar con *secuencias de escape*, las cuales son secuencias de caracteres que empiezan con una barra invertida (‘\’). Uno de los usos de una secuencia de escape es incluir un carácter de doble comillado en una constante cadena. Ya que un simple comilla doble representaría el final de la cadena, debes usar ‘\’ para representar un carácter de doble comilla como parte de la cadena. El mismo carácter de barra invertida es otro carácter que no puede ser incluído normalmente, debes escribir ‘\\’ para incluir este carácter en una cadena. Por lo que, la cadena cuyo contenido son los dos caracteres ‘”\’ debería ser escrita ‘”\\”’. Otro uso de la barra invertida es para representar caracteres no imprimibles como el newline. Aquí tienes una tabla de todas las secuencias de escape usadas en awk:

`\\`	Representa una barra invertida literal, `\'.
`\a`	Representa el carácter “alerta”, control-g, código ASCII 7.
`\b`	Representa el retroceso o espacio atrás, control-h, código ASCII 8.
`\f`	Representa un formfeed, control-l, código ASCII 12.
`\n`	Representa un carácter de nueva línea (newline), control-j, código ASCII 10.
`\r`	Representa un retorno de carro, control-m, código ASCII 13.
`\t`	Representa un tabulador horizontal, control-i, código ASCII 9.
`\v`	Representa un tabulador vertical, control-k, código ASCII 11.
`\nnn`	Representa el valor octal *nnn*, donde *nnn* son tres dígitos comprendidos entre 0 y 7. Por ejemplo, el código para el valor del carácter ASCII ESC (escape) es `\033'.
`\xhh...`	Representa el valor hexadecimal hh, donce hh son dígitos hexadecimales (desde ‘0’ a ‘9’ y desde ‘A’ a ‘F’ o desde ‘a’ a ‘f’) Al igual que la misma construcción en el ANSI C, la secuencia escape continúa hasta que se encuentra el primer dígito no hexadecimal. Sin embargo, el uso de más de dos dígitos hexadecimales produce resultados indefinidos.

Una expresión regular constante es una descripción de expresión regular encerrada entre barras, tal y como /^comienzo y fin$/. La mayoría de las expresiones regulares usadas en los programas awk son constantes, pero los operadores ‘~’ y ‘!~’ pueden también encajar con expresiones regulares computadas o “dinámicas” (Ver la sección Cómo usar Expresiones Regulares).

Las expresiones regulares constantes son útiles sólo con los operadores ‘~’ y ‘!~’; no las puedes asignar a variables o imprimirlas. Ellas no son expresiones realmente en el sentido estricto de la palabra.

Variables

Las variables te permiten darle un nombre a los valores y referirte a ellos posteriormente. Ya has visto variables en muchos de los ejemplos. El nombre de una variable debe ser una secuencia de letras, dígitos y subrayados, pero no pueden empezar con un dígito. Se hace distinción entre mayúsculas y minúsculas en los nombres de las variables, la variable a es distinta de la variable A.

Un nombre de variable es una expresión válida en si misma, representa el valor actual de la variable. Las variables reciben valores nuevos con los operadores de asignación y operadores incrementales. Ver la sección Expresiones de Asignación.

Unas pocas variables tienen significados implícitos especiales, tal y como FS, el separador de campo, y NF, el número de campos en el registro de entrada actual. Ver la sección 13. Variables Implícitas (Built-in) para ver una lista de las mismas. Estas variables implícitas pueden ser usadas y asignadas como cuales quiera otras variables, pero sus valores son también usados o cambiados automáticamente por awk. Todos los nombre de variables implícitas están formados por letras mayúsculas.

A las variables en awk se les puede asignar valores numéricos o cadenas de texto. Por defecto, las variables son inicializadas a la cadena nula, la cual es un 0 cuando es convertida a número. Por lo que no es necesario inicializar

cada variable explícita en awk, del mismo modo que necesitarías hacerlo en C o en la mayoría de los lenguajes de programación tradicionales.

Asignación de Variables en la Línea de Comando

Puedes asignarle un valor a cualquier variable awk incluyendo una asignación de variable entre los argumentos en la línea de comando cuando invocas a awk (Ver la sección 14. Invocación de awk). Tal asignación presenta la siguiente forma:

variable=*text*

Con ello, puedes darle un valor a una variable al principio de la ejecución de awk o entre el procesamiento de los ficheros de entrada.

Si precedes la asignación con la opción '-v', tal y como esto:

-v *variable*=*text*

entonces la variable toma su valor al principio, antes incluso de que se ejecuten las reglas BEGIN. La opción '-v' y su asignación debe preceder todos los argumentos de nombres de ficheros.

De otro modo, la asignación a variable se realiza en un momento determinado por su posición entre los argumentos nombre de fichero: después del procesamiento del argumento fichero de entrada precedente. Por ejemplo:

```
awk '{ print $n }' n=4 inventario-enviado n=2 Lista-BBS
```

imprime el valor del campo n para todos los registros de entrada. Antes de que se lea el primer fichero, la línea de comando fija el valor de la variable n a 4. Esto causa que se imprima el campo cuarto de todos los registros del fichero 'inventario-enviado'. Después de acabar con el primer fichero, pero antes de que se comience con el segundo, n toma el valor de 2, de forma que se imprime el segundo campo de todas las líneas del fichero 'Lista-BBS'.

Los argumentos de la línea de comando son puestos a disposición de awk para una examinación explícita en el array llamado ARGV (Ver la sección 13. Variables Implícitas (Built-in).

Operadores Aritméticos

El lenguaje awk utiliza los operadores aritméticos comunes cuando se evalúan expresiones. Todos estos operadores aritméticos siguen las reglas de precedencia normales, y se comportan como se podría esperar de ellos. Este ejemplo divide el campo tercero entre el campo cuarto y el resultado lo suma al campo dos, almacenando el resultado en el campo uno, e imprime el registro de entrada alterado:

```
awk '{ $1 = $2 + $3 / $4; print }' inventario-enviado
```

Los operadores aritméticos en awk son:

x + *y*	Suma.
x - *y*	Resta.
- *x*	Negación.
x * *y*	Multiplicación
x / *y*	División. Ya que todos los números en awk son puntos flotante de doble precisión, el resultado no se redondea a entero: 3 / 4 tiene el valor 0.75.

x % y	Resto. El cociente se redondea a cero o a un entero, se multiplica por y y este resultado se le resta a x. La siguiente relación siempre se cumple: b * int(a / b) + (a % b) == a Otro efecto indeseable de esta definición de resto es que x%y sea negativo si x es negativo. Por lo que, -17 % 8 = -1 En otras implementaciones awk, la falta de signo del resto puede ser dependiente de la máquina.
x ^ y *x ** y*	Exponenciación: x elevado a la potencia de y. 2 ^ 3 tiene el valor 8. La secuencia de caracteres '**' es equivalente al carácter '^'.

Concatenación de Cadenas

Existe una única operación con cadenas: la concatenación. No tiene un operador específico para representarla. En su lugar, la concatenación se realiza escribiendo expresiones unas al lado de las otras, sin operador. Por ejemplo:

```
awk '{ print "Campo número uno: " $1 }' Lista-BBS
```

produce, para el primer registro en `Lista-BBS':

```
Campo número uno: aardvark
```

Sin el espacio en la cadena constante después de los dos puntos ':', la línea hubiese aparecido junta. Por ejemplo:

```
awk '{ print "Campo número uno:" $1 }' Lista-BBS
```

produce, para el primer registro en `Lista-BBS':

```
Campo número uno:aardvark
```

Ya que la concatenación de cadenas no tiene un operador explícito, es necesario a menudo asegurarse que ocurre donde la quieres encerrando los items a ser concatenados en paréntesis. Por ejemplo, el siguiente fragmento de código no concatena fichero y nombre como podrías esperar:

```
file = "fichero"
name = "nombre"
print "algo útil" > file name
```

Es necesario usar lo siguiente:

```
print "algo útil" > (file name)
```

Recomendamos que uses los paréntesis alrededor de la concatenación en al menos todos los contextos más comunes (tal y como en la parte derecha del operador '=').

Expresiones de Comparación

Expresiones de comparación comparan cadenas o números para relaciones tales como la igualdad. Se escriben usando *operadores relacionales*, los cuales son un superconjunto de los de C. Aquí tienes una tabla de ellos:

x < *y*	Verdad si x es menor que y.
x <= *y*	Verdad si x es menor o igual que y.
x > *y*	Verdad si x es mayor que y.
x >= *y*	Verdad si x es mayor o igual que y.
x == *y*	Verdad si x es igual a y.
x != *y*	Verdad si x no es igual a y.
x ~ *y*	Verdad si la cadena x encaja con la expresion regular representada por y.
x !~ *y*	Verdad si la cadena x no encaja con la expresión regular representada por y.
subíndice in *array*	Verdad si el array *array* tiene un elemento con el subíndice *subíndice*.

La expresiones de comparación tienen el valor 1 si son ciertas y 0 sin son falsas.

Los operandos de un operador relacional son comparados como números si ambos son números. Sino, son convertidos y comparados como cadenas (Ver la sección Conversiones de Cadenas y Números).

Las cadenas son comparadas comparando el primer carácter de cada una, después el segundo carácter de cada una, y así sucesivamente. Por lo que, “10” es menor que “9”.

Por ejemplo,

```
$1 == "foo"
```

tiene el valor de 1, o es cierto, si el primer campo del registro de entrada actual es precisamente ‘foo’. En contraposición,

```
$1 ~ /foo/
```

tiene el valor 1 si el primer campo contiene ‘foo’.

La parte derecha de los operadores ‘~’ y ‘!~’ podría ser una expresión regular constante (/…/) o una expresión ordinaria, en cuyo caso el valor de la expresión como una cadena es una expresión regular dinámica (Ver la sección Cómo usar Expresiones Regulares)

En implementaciones de awk muy recientes, una expresión regular constante entre barras por si misma es también una expresión. La expresión regular */expreg/* es una abreviación de esta expresión de comparación:

```
$0 ~ /expreg/
```

En algunos contextos podría ser necesario escribir paréntesis alrededor de la expresión regular para evitar confundir al analizador de sentencias de awk. Por ejemplo, `(/x/ - /y/) > umbral` no está permitido, pero `((/x/) - (/y/)) > umbral` si es aceptado por el analizador de sentencias.

Un sitio especial donde `/foo/` no es una abreviación de `$0 ~ /foo/` es cuando es el operador de la derecha de `~' o `!~'!

Expresiones Booleanas

Una *expresión booleana* es una combinación de expresiones de comparación o expresiones matching, usando los *operadores booleanos* "o" (`||'), "y" (`&&'), y "not" (`!'), usando los paréntesis para controlar el anidamiento. La veracidad de la expresión booleana se calcula combinando la veracidad de las expresiones que la componen.

Las expresiones booleanas pueden ser usadas en cualquier lugar donde puedas usar una expresión de comparación o matching. Pueden ser usadas en sentencias *if* y *while*. Tienen valores numéricos (1 si es cierto, 0 si es falso), cuando el resultado de la expresión booleana se almacena en una variable, o usada en aritmética.

Además, toda expresión booleana es también un patrón booleano válido, por lo que puedes usarla como un patrón para controlar la ejecución de reglas.

Aquí están las descripciones de los tres operadores booleanos, con un ejemplo de cada uno. Podría ser instructivo comparar estos ejemplos con los ejemplos análogos de patrones booleanos (Ver la sección Operadores Booleanos como Patrones) los cuales usan los mismos operadores booleanos en patrones en lugar de expresiones.

booleana1 && booleana2

Verdad si tanto *booleana1* como *booleana2* son ciertas. Por ejemplo, la siguiente sentencia imprime el registro actual si contiene '2400 y 'foo'.

```
if ($0 ~ /2400/ && $0 ~ /foo/) print
```

La subexpresión *booleana2* se evalúa solamente si la expresión *booleana1* es cierta. Esto puede implicar una distinción cuando *booleana2* contiene expresiones que tienen efectos laterales: en el caso de `$0 ~ /foo/ && ($2 == bar++)`,la variable *bar* no se incrementa si no aparece primeramente 'foo' en el registro.

booleana1 || booleana2

Esto es cierto si al menos una de las dos, o *booleana1* o *booleana2*, es cierta. Por ejemplo, el siguiente comando imprime todos los registro del fichero de entrada 'Lista-BBS' que contengan '2400' o 'foo' o ambos.

```
awk '{ if ($0 ~ /2400/ || $0 ~ /foo/) print }' Lista-BBS
```

La subexpresión *booleana2* se evalúa solamente si la expresión *booleana1* es falsa. Esto debe ser tenido en cuenta cuando *booleana2* contiene expresiones que tienen efectos laterales.

!booleana

Verdad si *booleana* es falsa. Por ejemplo, el siguiente programa imprime todos los registros del fichero de entrada 'Lista-BBS' que no contengan la cadena 'foo'.

```
awk '{ if (! ($0 ~ /foo/)) print }' Lista-BBS
```

Expresiones de Asignación

Una *asignación* es una expresión que almacena un valor nuevo en una variable. Por ejemplo, asignemos el valor 1 a la variable *z*.

```
z = 1
```

Después de que esta expresión se haya ejecutado, la variable *z* tiene el valor 1. Cualquier valor anterior que tuviese *z* antes de la asignación se pierde.

Las asignaciones pueden almacenar valores tipo cadena también. Por ejemplo, esto almacenaría el valor "*esta comida es buena*" en la variable *mensaje*.

```
cosa = "comida"
predicado = "buena"
mensaje = "esta " cosa " es " predicado
```

(Este ejemplo además ilustra una concatenación de cadenas).

Al signo '=' se le llama *operador asignación*. Es el operador de asignación más simple porque el valor del operando que se encuenta a la derecha permanece invariable.

La mayoría de los operadores (adición, concatenación, etc.) no tienen ningún efecto para computar un valor. Si ignoras el valor, podrías también no usar el operador. Un operador de asignación es diferente, produce un valor, pero incluso si ignoras el valor, la asignación todavía se realiza. Se llama a esto *efecto lateral*.

El operando a la izquierda de la asignación no tiene porqué ser una variable (Ver la sección Variables); también podría ser un campo (Ver la sección Cambiando los contenidos de un campo) o el elemento de un array (Ver la sección 10. Arrays en awk). Estos son llamados ***valoresi***, lo cual significa que pueden aparecer en la parte izquierda de un operador de asignación. El operando de la derecha podría ser una expresión; produce el número valor que la asignación almacena en la variable, campo o elemento de array especificado.

Es importante señalar que las variables *no* tienen tipos de datos permanentes. El tipo de una variable es simplemente el tipo del valor que almacena la variable en ese momento. En el siguiente fragmento de programa, la variable *foo* tiene un valor numérico primero y un valor cadena después:

```
foo = 1
print foo
foo = "bar"
print foo
```

Cuando la segunda asignación le da a la variable *foo* un valor cadena, el hecho de que esta variable almacenase previamente un valor numérico es olvidado.

La asignación es una expresión, de forma que tiene un valor: el mismo valor que es asignado. Por lo que, `z = 1` como expresión tiene el valor 1. Una consecuencia de esto es que puedes escribir asignaciones múltiples juntas:

```
x = y = z = 0
```

almacena el valor 0 en las tres variables. Hace esto porque el valor de `z = 0`, es cual es 0, se almacena en y, y el valor de `y = z = 0`, que es 0, se almacena en x.

Puedes usar una asignación en cualquier lugar en el que se pueda llamar a una expresión. Por ejemplo, es válido escribir `x != (y = 1)` para darle a y el valor 1 y posteriormente chequear si x es igual a 1. Pero este estilo tiende a crear programas difíciles de leer.

Además de '=', existen otras diversos operadores de asignación que realizan una operación aritmética con el valor antiguo de la variable. Por ejemplo, el operador '+=' calcula un nuevo valor para la variable añadiendo el valor de la derecha al valor antiguo de la variable. Por lo que, la siguiente asignación añade 5 a la variable de 'foo':

```
foo += 5
```

Esto es precisamente equivalente a:

```
foo = foo + 5
```

Usa el que haga que tu programa sea más claro.

Aquí tienes una tabla de operadores de asignación aritméticos. En todos los casos, el operando de la derecha es una expresión cuyo valor se convierte a número.

valori += incremento	Añade *incremento* al valor de *valori* para obtener el nuevo valor a asignar a *valori*.
valori -= decremento	Substrae *decremento* del valor de *valori*.
valori *= coeficiente	Multiplica el valor de *valori* por *coeficiente*.
valori /= coeficiente	Divide el valor de *valori* entre *coeficiente*.
valori %= módulo	Le asigna a *valori* el resto de la división entre *modulo*.
valori ^= potencia ***valori **= potencia***	Eleva a *valori* a la potencia de *potencia*.

Operadores Incrementales

Los *Operadores incrementales* incrementan o decrementa el valor de una variable en 1. Podrías hacer lo mismo con un operador de asignación, de forma que los operadores incrementales no añaden potencia al lenguaje ***awk***; pero añaden una abreviación conveniente para algo muy común.

El operador para añadir 1 se escribe '++'. Puede ser usado para incrementar una variable antes o después de coger su valor.

Para pre-incrementar una variable *v*, escribe ++v. Esto añade 1 al valor de *v* y ese nuevo valor es también el valor de esta expresión. La expresión de asignación v += 1 es completamente equivalente.

La escritura de '++' después de la variable especifica un post-incremento. Esto incrementa la variable de igual manera; la diferencia es que el valor de la expresión de incremento por si misma es el viejo valor de la variable (antes de ser incrementada). Por lo que, si *foo* tiene un valor de 4, entonces la expresión *foo*++ tiene el valor de 4, pero cambia el valor de la variable *foo* a 5.

El post-incremento *foo*++ es aproximadamente equivalente a escribir *(foo += 1) – 1*. No es perfectamente equivalente porque todos los números en ***awk*** son punto flotante: en punto flotante, *foo +1 – 1* no es necesariamente igual a foo. Pero la diferencia es mínima mientras lo uses con números que sean pequeños (menores a un trillón).

Cualquier *valori* puede ser incrementado. Los campos y los elementos de array pueden ser incrementados del mismo modo que las variables.

El operador decremento '--' funciona del mismo modo que el '++' excepto que lo que hace es restar 1 en lugar de añadirlo. Al igual que el operador '++', puede ser usado antes del *valori* para un pre-decremento o después para un post-decremento.

Aquí tienes un sumario de las expresiones de incremento y decremento.

++valori	Esta expresión incrementa *valori* y el nuevo valor se convierte en el valor de la expresión.
valori++	Esta expresión hace que se incremente el valor de *valori*.. El valor de la expresión es el valor *antiguo* de *valori*.
--valori	Como ++*valori*, pero en lugar de adición, substrae. Decrementa el valor de *valori* y devuelve dicho valor como resultado de la expresión.
valori--	Como *valori*++, pero en lugar de añadir, substrae. Decrementa *valori*. El valor de la expresión es el valor *antiguo* de *valori*.

Conversiones de Cadenas y Números

Las cadenas son convertidas a números, y los números a cadenas, si el contexto del programa *awk* lo requiere. Por ejemplo, si el valor de la variable *foo* o la variable *bar* en la expresión *foo* + *bar* es una cadena, dicho valor es convertido a número antes de que se realice la suma. Si aparecen valores numéricos en una concatenación de cadenas, son convertidos a cadenas. Considere esto:

```
two = 2; three = 3
print (two three) + 4
```

Esto imprime eventualmente el valor (numérico) 27. Los valores numéricos de las variables *two* y *three* son convertidos a cadenas y concatenados juntos, y el resultado de la concatenación se convierte a número, resultando el número 23, al cual se le añade el número 4.

Si ,por alguna razón, necesitas forzar la conversión de un número a cadena, concaténa la cadena nula con dicho número. Para forzar que una cadena sea convertida a número, súmale cero a esa cadena.

Las cadenas son convertidas a números mediante su interpretación como números: `"2.5"` se convierte en 2.5, y `"1e3"` se convierte en 1000 (notación científica). Las cadenas que no pueden ser interpretadas como números válidos son convertidas a cero.

La forma exacta que se usa para convertir números en cadenas es controlada por la variable implícita OFMT (Ver la sección 13. Variables Implícitas (Built-in).

La manera exacta en la que los números son convertidos en cadenas está controlada por la variable implícita de awk `OFMT` (Ver la sección 13. Variables Implícitas (Built-in). Los números son convertidos usando una versión especial de la función `sprintf` (Ver la sección 11. Funciones Implícitas (Built-in)) con `OFMT` como especificador de formato.

El valor por defecto de `OFMT` es `"%.6g"`, el cual imprime un valor con al menos seis dígitos significativos. Para algunas aplicaciones querrás cambiar este valor para obtener una precisión mayor. "*Double precision*" en la mayoría de las máquinas modernas te da 16 o 17 dígitos decimales de precisión.

Podrían darse resultados extraños si le das a `OFMT` un valor o cadena especificadora de formato que no le indique correctamente a ***`sprintf`*** como formatear números en punto flotante de una forma correcta. Por ejemplo, si olvidas el '%' en el formato, todos los números serán convertidos a la misma cadena constante.

Expresiones Condicionales

Una *expresión condicional* es un tipo especial de expresión con tres operandos. Te permite usar el valor de una expresión para seleccionar una expresión de entre otras dos posibles.

La expresión condicional presenta el mismo formato que en el Lenguaje C:

`selector ? if-true-exp : if-false-exp`

Existen tres subexpresiones. La primera, *`selector`*, es siempre tratada primero. Si es “cierta” (distinta de cero) entonces se procesa la expresión `if-true-exp` y su valor se convierte en el valor de la expresión completa. Si no, `if-false-exp` es procesada y su valor es el que se convierte en el valor de la expresión completa.

Por ejemplo, esta expresión produce el valor absoluto de x:

```
x > 0 ? x : -x
```

Cada vez que la expresión condicional se procesa, se ejecuta siempre una de las dos expresiónes *`if-true-exp`* o *`if-false-exp`*, la otra es ignorada. Esto es importante cuando las expresiones contienen efectos laterales. Por ejemplo, esta expresión condicional examina el elemento `i` de el array `a` o del array `b`, y a continuación lo incrementa.

```
x == y ? a[i++] : b[i++]
```

Esto garantiza que se incrementa `i` exactamente una vez, porque cada vez se ejecuta o uno u otro incremento y el otro no.

Llamadas a Funciones

Una *función* es un nombre para un cálculo particular. Debido a que tiene un nombre, puedes llamarla por dicho nombre en cualquier punto de un programa. Por ejemplo, la función *sqrt* calcula la raíz cuadrada de un número.

Un conjunto fijo de funciones son implícitas (*built-in*), lo que significa que están disponibles en cualquier programa de awk. La función *sqrt* es una de estas. Ver la sección 11. Funciones Implícitas (Built-in), para una lista de funciones implícitas y sus descripciones. Además de éstas funciones, puedes definir tus propias funciones en el programa para usar en cualquier punto dentro del mismo programa. Ver la sección 12. Funciones definidas por el usuario, para conocer como hacer esto.

La forma de utilizar una función es con una *expresión de llamada a función*, la cuál consiste en el nombre de la función seguido de una lista de *argumentos* entre paréntesis. Los argumentos son expresiones las cuales le proporcionan los valores a la función para que ésta realice sus cálculos u operaciones. Cuando existen más de un argumento, éstos estarán separados por comas. Si la función no necesita argumentos, escribe simplemente ‘()’ detrás del nombre de la función. Aquí tienes algunos ejemplos:

```
sqrt(x**2 + y**2)    # One argument
atan2(y, x)          # Two arguments
rand()               # No arguments
```

¡No pongas ningún espacio en blanco entre el nombre de la función y el paréntesis de apertura!. Una función definida por el usuario es similar al nombre de una variable, y un espacio produciría que la expresión pareciese la concatenación de un variable con una expresión encerrada entre paréntesis (argumentos de la función). Por lo que

sería bueno acostumbrarse a no dejar espacios en blanco entre los nombres de funciones (definidas por el usuario o implícitas) y los paréntesis con los argumentos.

Cada función espera un número concreto de argumentos. Por ejemplo, la función *sqrt* debe ser llamada con un único argumento, el número del que se va a obtener la raíz cuadrada.

`sqrt(`*`argument`*`)`

Algunas de las funciones implícitas te permiten omitir el argumento final. Si haces esto, estas funciones realizan algo razonable por defecto. Ver la sección 11. Funciones Implícitas (Built-in), para detalles sobre esto. Si se omiten argumentos en llamadas a funciones definidas por el usuario, entonces esos argumentos son tratados como variables locales, inicializadas a la cadena nula (Ver la sección 12. Funciones definidas por el usuario).

Al igual que otras muchas expresiones, la llamada a función tiene un valor, el cual es calculado por la función basándose en los argumentos que le pasas. En este ejemplo, el valor de *sqrt(argumento)* es la raíz cuadrada del argumento. Una función también puede tener efectos laterales, tales como asignar el valor de ciertas variables o hacer Entrada/Salida. Aquí está un comando para leer números, un número por línea, e imprime la raíz cuadrada de cada uno:

```
awk '{ print "La raíz cuadrada de ", $1, " es ", sqrt($1) }'
```

Precedencias de Operadores: Cómo se anidan los Operadores

La *precedencia de operadores* determina como se agrupan los mismos, cuando aparecen distintos operadores cerca unos de otros en una expresión. Por ejemplo, '*' tiene mayor precedencia que '+'; por lo que a +b * c significa que hay que multiplicar b * c y el resultado sumárselo a *a*.

Puedes saltarte la precedencia de operadores mediante los paréntesis. De hecho, es conveniente usar paréntesis donde quiera que tengas una combinación de operadores inusual, porque otras personas que lean el programa podrían no recordar cuál es la precedencia en ese caso. Tú mismo también podrías olvidarla; por lo que podrías cometer un fallo. Los paréntesis explícitos te prevendrán de tales errores.

Cuando se usan juntos operadores de igual precedencia, el operador más a la izquierda es el que tiene la precedencia, a excepción de la asignación, y operadores condicionales y exponenciales, los cuales se agrupan en el orden contrario. Por lo que, `a - b + c` se agrupa de la siguiente forma `(a - b) + c`; `a = b = c` se agrupa `a = (b = c)`.

La precedencia de operadores unarios prefijo no importa ya que solo se ven implicados los operadores unarios, porque solo hay una forma de hacerles el parse – el más interno primero. Por lo que, `$++i` significa `$(++i)` y `++$x` significa `++($x)`. Sin embargo, cuando otro operador sigue al operando, entonces la precedencia de los operadores unarios puede presentar problemas. Por lo que, `$x**2` es lo mismo que `($x)**2`, pero `-x**2` significa `-(x**2)`, porque `` `-' `` tiene menor precedencia que `` `**' `` mientras que `` `$' `` tiene una mayor precedencia.

Aquí tienes una lista de operadores de awk, en orden de precedencia creciente:

asignación	`` `=', `+=', `-=', `*=', `/=', `%=', `^=', `**=' `` Estos operadores se agrupan de derecha a izquierda.
condicional	`` `?:' `` Estos operadores se agrupan de derecha a izquierda.
"o" lógico.	`` `||' ``

“y” lógico	`&&'
Pertenencia a array	in
Encaje o patrón	`~', `!~'
Relacional y redireccionamiento	Los operadores relacionales y las redirecciones tienen el mismo nivel de precedencia. Caracteres tales como ‘>’ sirven tanto como relacionales como de redireccinamiento; el contexto hace que se diferencie entre los dos significados. Los operadores relacionales son `<', `<=', `==', `!=', `>=' y `>'. Los operadores de redireccionamiento de la Entrada/Salida son `<', `>', `>>' y `|'. Anotar que los operadores de redireccionamiento de la Entrada/Salida en las sentencias print y printf pertenecen al nivel de sentencia, no a las expresiones. El redireccionamiento no produce una expresión la cual podría ser el operando de otro operador. Como resultado de esto, no tendría sentido usar un operador de redirección cerca de otro operador de precedencia menor, sin paréntesis. Tales combinaciones, por ejemplo `print foo > a ? b : c', produce un error de sintaxis.
Concatenación	No se usa ningún token especial para indicar la concatenación. Los operandos son escritos simplemente uno al lado de otro (con un espacio en medio de ambos).
adición, sustración	`+', `-'
Multiplicación, división y módulo	`*', `/', `%'
Más y menos unario, "not"	`+', `-', `!'
exponenciación	`^', `**' Estos operadores se agrupan de derecha a izquierda
Incremento, decremento	`++', `--'
Campo	`$'

9. Acciones: Sentencias de Control

Las sentencias de control tales como if, while, y así sucesivamente controlan el flujo de la ejecución de los programas awk. La mayoría de las sentencias de control de awk son similares a sus sentencias análogas en Lenguaje C.

Todas las sentencias de control comienzan con las palabras claves tales como if y while, para distinguirlas de las expresiones simples.

Muchas sentencias de control contienen otras sentencias; por ejemplo, la sentencia if contiene otra sentencia la cual podría ser ejecutada o no. Las sentencias contenidas se llaman el *cuerpo*. Si quieres incluir más de una sentencia en el *cuerpo*, agrupalas en una única sentencia compuesta con llaves, separándose entre ellas con puntos y comas y saltos de línea.

La Sentencia if

La sentencia ***if-else*** es una sentencia para la toma de decisiones de awk. Presenta la siguiente forma:

if (*condición*) *cuerpo-then* [else *cuerpo-else*]

Aquí condición es una expresión que controla qué es lo que realizará el resto de la sentencia. Si la condición es verdad, entonces se ejecuta el cuerpo-then; sino se ejecuta el cuerpo-else (asumiendo que esté presente la clásula else). La parte ***else*** de la sentencia es opcional. La condición se considera falsa si su valor es 0 o una cadena nula, sino se considerará verdadera.

Aquí se presenta un ejemplo:

```
if (x % 2 == 0)
    print "x es par"
else
    print "x es impar"
```

En este ejemplo, si la expresión x%2 == 0 es cierta (es decir, el valor de x es divisible entre 2), entonces se ejecuta la primera sentencia print, sino se ejecuta la segunda sentencia print.

Si el *else* aparece en la misma línea que el *cuerpo-then*, y el *cuerpo-then* no es una sentencia compuesta (no aparece entre llaves), entonces un punto y coma debe separar el *cuerpo-then* del *else*. Para ilustrar esto veamos el siguiente ejemplo:

```
awk '{ if (x % 2 == 0) print "x is even"; else
        print "x is odd" }'
```

Si olvidas el ‘;’ awk no será capaz de realizar el parse de la sentencia, y obtendrás un error de sintáxis. De todas formas es una mala práctica escribir la sentencia else de esta forma, ya que puede llevar al lector a confusión.

La Sentencia while

En programación, un *bucle* significa una parte de programa que es (o al menos puede ser) ejecutada dos o más veces seguidas.

La sentencia *while* es la sentencia de bucle más simple de awk. Ejecuta repetidamente una sentencia mientras una condición sea cierta. Presenta la siguiente forma:

```
while (condición)
  cuerpo
```

Aquí *cuerpo* es una sentencia a la que llamamos el *cuerpo* del bucle, y *condición* es una expresión que controla cuantas veces se ejecuta dicho *cuerpo*.

La primera cosa que hace la sentencia *while* es comprobar la *condición*. Si *condición* es cierta, ejecuta la sentencia *cuerpo*. (Verdad, como es usual en awk, significa que el valor de la *condición* es distinto de cero y la cadena nula). Después de que el *cuerpo* se haya ejecutado, se chequea la *condición* de nuevo, y si sigue siendo cierta, el *cuerpo* se ejecuta de nuevo. Este proceso se repite hasta que la *condición* deja de ser cierta. Si la *condición* es falsa inicialmente, el cuerpo del bucle nunca es ejecutado.

Este ejemplo imprime los tres primeros campos de cada registro, uno por línea.

```
awk '{ i = 1
       while (i <= 3) {
           print $i
           i++
       }
}'
```

Aquí el cuerpo del bucle es una sentencia compuesta encerrada entre llaves, que contiene dos sentencias.

El bucle trabaja de la siguiente forma: primero, el valor de i es fijado a 1. Entonces, el *while* chequea si i es menor o igual a tres. Este es el caso cuando *i* es igual a uno, así que el campo iésimo es impreso. Entonces el comando i++ incrementa el valor de i y se repite el bucle. El bucle termina cuando *i* alcanza el valor de 4.

Como puedes comprobar, un carácter de nueva línea no es requerida entre la condición y el cuerpo; pero usándola hace que el programa sea más claro a menos que el cuerpo sea una sentencia compuesta o una muy simple. El carácter newline después de la llave de apertura que inicia la sentencia compuesta no es obligatoria, pero el programa será menos legible sino se pone.

La Sentencia do-while

El bucle *do* es una variación de la sentencia de bucle *while*. El bucle *do* ejecuta el *cuerpo* al menos una vez, después repite el *cuerpo* mientras la *condición* se siga evaluando a cierto. Presenta la siguiente forma:

```
do
  cuerpo
while (condición)
```

Incluso aunque *condición* sea falsa inicialmente, el *cuerpo* se ejecuta al menos una vez (y sólo una vez, a menos que el *cuerpo* de ejecución siga haciendo que la *condición* sea cierta). Contrasta esto con la correspondiente sentencia *while*:

```
while (condición)
  cuerpo
```

Esta sentencia no ejecuta el *cuerpo* ni una vez si la *condición* es falsa antes de empezar.

Aquí está un ejemplo de una sentencia *do*:

```
awk '{ i = 1
       do {
          print $0
          i++
       } while (i <= 10)
}'
```

imprime cada registro de entrada 10 veces. No es un ejemplo muy realista, ya que un *while* ordinario podría haber hecho lo mismo.

La Sentencia for

La sentencia *for* es más conveniente para contar las iteraciones de un bucle. La forma general de la sentencia *for* presenta la siguiente forma:

```
for (inicialización; condición; incremento)
  cuerpo
```

Esta sentencia comienza ejecutando *inicialización*. Después, mientras la *condición* sea cierta, ejecuta repetidamente el *cuerpo* y el *incremento*. Normalmente la *inicialización* fija el valor de la variable a 0 o 1, *incremento* añade 1 a dicho valor, y *condición* compara el nuevo valor contra el número de iteraciones deseado.

Aquí tienes un ejemplo de una sentencia *for*:

```
awk '{ for (i = 1; i <= 3; i++)
          print $i
}'
```

Esto imprime los tres primers campos de cada registros de entrada, un campo por línea.

En la sentencia *for*, el *cuerpo* está formado por sentencias, pero *inicialización*, *condición* e *incremento* son simples expresiones. No puedes fijar el valor de más de una variable en la parte *inicialización* a menos que uses una sentencia de asignación múltiple tal como x = y = z = 0, la cual sólo es posible si todos los valores iniciales son el mismo (pero puedes inicializar variables adicionales escribiendo sus asignaciones como sentencias separadas que precedan al bucle *for*).

Lo mismo se cumple para la parte *incremento*; para incrementar variables adicionales, debes escribir sentencias separadas al final del bucle. La expresión compuesta C, que usa el separador de C coma, sería útil en este contexto, pero no está soportada por awk.

En la mayoría de los casos, *incremento* es una expresión incremental, como en el ejemplo de arriba. Pero esto no es obligatorio; podría ser cualquier expresión. Por ejemplo, esta sentencia imprime todas las potencias de 2 entre 1 y 100:

```
for (i = 1; i <= 100; i *= 2)
  print i
```

Cualquiera de las tres expresiones en los paréntesis que siguen al *for* podría ser omitida si no fuese necesaria. Por lo que, `for (;x > 0;)' es equivalente a `while (x > 0)'. Si la *condición* se omite, se trata como cierta, dando lugar a un bucle infinito efectivo.

En la mayoría de los casos, un bucle for es una abreviación de un bucle *while*, tal y como se muestra aquí:

```
inicialización
while (condición) {
  cuerpo
  incremento
}
```

La única excepción es cuando se utiliza la sentencia *continue* (Ver la sección La Sentencia continue) dentro del bucle; cambiar una sentencia *for* por una sentencia *while* de esta forma puede cambiar el efecto de la sentencia *continue* dentro del bucle.

Existe una versión alternativa del bucle *for*, para hacer una iteración para cada índice de un array:

```
for (i in array)
    hacer algo con array[i]
```

Ver la sección 10. Arrays en awk, para más información de esta versión del bucle *for*.

El Lenguaje awk tiene una sentencia *for* además de la sentencia *while* porque a menudo un bucle *for* es más fácil para escribirla y más natural para pensarla y entenderla. Contar el número de iteraciones es muy común en los bucles. Es más fácil pensar este conteo como parte del bucle en lugar de cómo algo a hacer dentro del bucle. La siguiente sección tiene ejemplos más complicados de bucles *for*.

La Sentencia break

La sentencia break se sale del bucle *for, while* o *do-while* más interno en el que está contenido. El siguiente ejemplo encuentra el divisor más pequeño de cualquier entero, y también identifica números primos:

```
awk '# encuentra el divisor más pequeño de num
     { num = $1
       for (div = 2; div*div <= num; div++)
         if (num % div == 0)
           break
       if (num % div == 0)
         printf "Smallest divisor of %d is %d\n", num, div
       else
         printf "%d is prime\n", num  }'
```

Cuando el resto es cero en la primera sentencia *if*, awk se sale inmediatamente del bucle *for* en el que está contenido. Esto significa que awk procede inmediatamente a la sentencia que sigue al bucle y continúa el procesamiento. (Esto es muy distinto de la sentencia *exit* (ver la sección La Sentencia exit) la cual detiene el programa awk completo.)

Aquí está otro programa equivalente al anterior. Ilusta como la *condición* de un *for* o *while* podría simplemente ser substituida con un break dentro de un *if*:

```
awk '# find smallest divisor of num
     { num = $1
       for (div = 2; ; div++) {
         if (num % div == 0) {
           printf "Smallest divisor of %d is %d\n", num, div
           break
         }
         if (div*div > num) {
           printf "%d is prime\n", num
           break
         }
       }
}'
```

La Sentencia continue

La sentencia *continue*, al igual que break, se usa solamente dentro de un bucle *for, while* o *do-while*. Lo que hace es saltarse todas las sentencias que falten por ejecutarse dentro del cuerpo del bucle y volver a chequear automáticamente la condición del bucle. Contrasta con la sentencia break, en que ésta produce directamente un salto fuera del bucle. Aquí se presenta un ejemplo:

```
# print names that don't contain the string "ignore"
# first, save the text of each line
{ names[NR] = $0 }
# print what we're interested in
END {
   for (x in names) {
       if (names[x] ~ /ignore/)
           continue
       print names[x]
   }
}
```

Si uno de los registros de entrada contiene la cadena ‘ignore’, este ejemplo se salta la sentencia print para ese registro, y vuelve a la primera sentencia del bucle.

Este no es un ejemplo práctico de *continue*, ya que está sería una forma más sencilla de escribir el bucle:

```
for (x in names)
  if (names[x] !~ /ignore/)
    print names[x]
```

La sentencia *continue* en un bucle *for* hace que awk se salte el resto del cuerpo del bucle, y resume la ejecución con la expresión incremento de la sentencia *for*. El siguiente programa ilustra esto de hecho:

```
awk 'BEGIN {
     for (x = 0; x <= 20; x++) {
         if (x == 5)
             continue
         printf ("%d ", x)
     }
     print ""
}'
```

Este programa imprime todos los números de 0 a 20, excepto el 5 para el cual se salta el print. Ya que el incremento no es saltado, x no permanece estancado en 5. Contrasta el bucle *for* de arriba con el bucle *while*:

```
awk 'BEGIN {
     x = 0
     while (x <= 20) {
         if (x == 5)
             continue
         printf ("%d ", x)
         x++
     }
     print ""
}'
```

Este programa se convierte en un bucle infinito una vez que x alcanza el valor de 5.

La Sentencia next

La sentencia *next* fuerza a awk que detenga inmediatamente el procesamiento del registro actual y vaya a leer el siguiente registro. Esto significa que no se ejecutará ninguna regla más para el registro en curso. El resto de las acciones de la regla actual no se ejecutarán.

Contrasta esto con el efecto de la función *getline* (Ver la sección Entrada explícita con getline). Eso también hace que awk lea el siguiente registro inmediatamento, pero no se altera el control de flujo del programa. De forma que el resto de la acción actual se ejecuta con un nuevo registro de entrada.

En el nivel más alto, la ejecución de un programa awk es un bucle que lee un registro de entrada y entonces lo chequea contra el patrón de cada regla. Si piensas en este bucle como una sentencia *for* cuyo cuerpo contiene reglas, entonces la sentencia *next* es análoga a una sentencia *continue*: se salta el resto del cuerpo de este bucle implícito, y ejecuta el imcremento (lo que provoca la lecuta de otro registro).

Por ejemplo, si tu programa awk trabaja sólo sobre registros con cuatro campos, y no quieres que procese una entrada incorrecta, podrías usar esta regla cerca del principio del programa:

```
NF != 4 {
  printf("line %d skipped: doesn't have 4 fields", FNR) > "/dev/stderr"
  next
}
```

de forma que las reglas siguientes no verán el registro incorrecto. El mensaje de error se redirecciona al streem de salida de error estándar, como debiera ser para los mensajes de error. Ver la sección Streams de Entrada/Salida Estándard.

La sentencia *next* no está permitida en una regla ***BEGIN*** o ***END***.

La Sentencia exit

La sentencia *exit* hace que awk detenga la ejecución de la regla actual inmediatamente y detenga el procesamiento de la entrada, cualquier registro que falte es ignorado.

Si una sentencia *exit* se ejecuta en una regla ***BEGIN*** el programa detiene el procesamiento de todo inmediatamente. No se leería ningún registro de entrada. Sin embargo, si existe una regla ***END***, ésta es ejecutada (Ver la sección Los Patrones Especiales BEGIN y END).

Si se usa *exit* como parte de una regla ***END***, hace que el programa se detenga inmediatamente.

Una sentencia *exit* como parte de una regla ordinaria (esto es, no forme parte de una regla ***BEGIN*** y ***END***) detiene la ejecución de cualquier regla automática posterior, pero la regla ***END*** sí es ejecutada si existe. Si no quieres que la regla ***END*** realice su trabajo en estos casos, puedes fijar una variable a distinto de cero antes de la sentencia *exit*, y chequear dicha variable en la regla ***END***.

Si se le suministra un argumento a *exit*, su valor se usa como el código de estado de salida para el proceso awk. Si no se le suministra argumento, *exit* devuelve el estado 0 (éxito). Por ejemplo, digamos que has descubierto una condición de error que no sabes como manejar realmente. Convencionalmente, el programa te reporta esto saliéndose con un estado distinto de cero. Tu programa awk puede hacer esto usando una sentencia *exit* con un argumento distinto de cero. Aquí tienes un ejemplo de esto:

```
BEGIN {
       if (("date" | getline date_now) < 0) {
         print "Can't get system date" > "/dev/stderr"
         exit 4
       }
}
```

10. Arrays en awk

Un array es una tabla de varios valores, llamados elementos. Los elementos de un array se distinguen por sus índices. Los índices pueden ser tanto cadenas como números. Cada array tiene un nombre, que es similar al nombre de una variable, pero no debe estar siendo usado como nombre de variable en un mismo programa awk.

Introducción a los Arrays

El lenguaje awk tiene arrays de una dimensión para almacenar grupos de cadenas o números relacionados.

Todos los arrays en awk deben tener un nombre. Los nombres de los arrays presentan la misma sintáxis que los nombres de variables; cualquier nombre de variable válido será un nombre válido para un array. Pero no se puede usar un mismo nombre para un array y una variable en un mismo programa awk.

Los arrays en awk se asemejan superficialmente a los arrays en otros lenguajes de programación; pero existen diferencias fundamentales. En awk, no es necesario especificar el tamaño de un array antes de empezar a usarlo. Lo que és más, en awk, cualquier número o incluso cadena podría ser usado como un índice de array.

En la mayoría de otros lenguajes, tienes que declarar un array y especificar cuantos elementos o componentes tiene. En dichos lenguajes, la declaración causa la reserva de un bloque de memoria contiguo para tantos elementos como se hayan especificado. Un índice para esos arrays debe ser un entero positivo, por ejemplo, el índice 0 especifica el primer elemento del array, el cual se encuentra realmente almacenado al principio del bloque de memoria reservado. Índice 1 especifica el segundo elemento, el cual está almacenado en la memoria justo a continuación del primer elemento, y así sucesivamente. No es posible añadir más elementos al array, porque la reserva de memoria ya se ha realizado para el número de elementos que se declaró inicialmente que iba a tener el array.

Un array contiguo de cuatro elementos podría presentar una forma similar a la siguiente, conceptualmente, si los valores de los elementos son 8, “foo”,”” y 30:

```
+---------+---------+--------+---------+
|    8    |  "foo"  |   ""   |    30   |    value
+---------+---------+--------+---------+
     0         1         2         3        index
```

Solamente los valores son almacenados; los índices son implícitos por el orden de los valores. 8 es el valor del índice 0, porque 8 aparece en la posición con 0 elementos antes de él.

Los arrays en awk son diferentes: son *asociativos*. Esto significa que cada array es una colección de pares: un índice, y su valor de elemento de array correspondiente:

```
Elemento 4     Valor 30
Elemento 2     Valor "foo"
Elemento 1     Valor 8
Elemento 3     Valor ""
```

Hemos mostrado los pares en un orden aleatorio porque el orden no tiene ningún significado.

Una ventaja de un array asociativo es que puedes añadir nuevas parejas en cualquier momento. Por ejemplo, supón que añadimos a ese array un elemento décimo cuyo valor es “número diez”. El resultado es este:

```
Elemento 10    Valor "número diez"
Elemento 4     Valor 30
Elemento 2     Valor "foo"
Elemento 1     Valor 8
Elemento 3     Valor ""
```

Ahora el array es disperso (algunos índices no aparecen): tiene los elementos 4 y 10, pero no tiene los elementos 5, 6, 7, 8, y 9.

Otra consecuencia de los arrays asociativos es que los índices no tienen por qué ser enteros positivos. Cualquier número, o incluso una cadena, puede ser un índice de un array. Por ejemplo, aquí está un array el cual traduce palabras de Inglés a Francés:

```
Elemento "dog" Valor "chien"
Elemento "cat" Valor "chat"
Elemento "one" Valor "un"
Elemento 1     Valor "un"
```

Aquí nosotros decidimos traducir el número 1 en ambas formas, la forma numérica y alfabética, lo que supone un ejemplo claro de que un array puede tener ambos números y cadenas como índices.

Cuando awk crea un array para ti, por ejemplo con la función implícita *split* (Ver la sección Funciones Implícitas (Built-in) para Manipulación de Cadenas), esos índices del array son enteros consecutivos que comienzan con el 1.

Refiriéndose a un elemento de un Array

La forma principal de usar un array es referirse a uno de sus elementos. Una referencia a un array es una expresión que presenta la siguiente forma:

***array*[*índice*]**

Aquí *array* es el nombre de un array. La expresión *índice* es el índice del elemento del array que tú quieres.

El valor de la referencia al array es el valor actual del elemento del array. Por ejemplo, foo[4,3] es una expresión para el elemento cuyo índice es 4,3 del array foo.

Si referencias un elemento de array en el que no se ha grabado ningún valor, el valor devuelto por esta referencia es " ", la cadena nula. Esto incluye elementos a los cuales no les has asignado un valor, y los elementos que han sido borrados (Ver la sección La Sentencia delete). Esta referencia automáticamente crea dicho elemento de array, con la cadena nula como su valor. (en algunos casos esto es un incoveniente ya que supone un desperdicio de memoria por parte de awk).

Puedes averigurar si existe un elemento en un array para un determinado índice con la expresión:

índice* in *array

Esta expresión chequea si existe o no el índice especificado, sin el efecto lateral de crear dicho elemento si no está presente. La expresión tendrá el valor de 1 (verdadero) si array[índice] existe, y 0 (falso) si no existe.

Por ejemplo, para chequear si el array *frequencies* contiene el índice "2", podrías escribir esta sentencia:

```
if ("2" in frequencies) print "Subscript \"2\" is present."
```

Señalar que esto no es un chequeo de si el array *frequencies* contiene o no un elemento cuyo valor es “2”. (No hay forma de hacer esto, excepto escaneando todos los elementos). También, esta sentencia ***no crea*** *frequencies[“2”]*, mientras que la siguiente sentencia (incorrecta) alternativa si lo haría:

```
if (frequencies["2"] != "") print "Subscript \"2\" is present."
```

Asignación de Elementos de Array

Los elementos de un array son *valoresi*: les pueden ser asignados valores del mismo modo que a las variables *awk*.

array*[*subíndice*] = *valor

Aquí *array* es el nombre de tu array. La expresión *subíndice* es el índice del elemento del array al que le quieres asignar un valor. La expresión *valor* es el valor que le estás asignando al elemento del array.

Un ejemplo básico de un Array

El siguiente programa toma una lista de líneas, cada una comenzando con un número de línea, y las imprime en orden del número de línea. Los números de línea no están en orden. Este programa ordena las líneas creando un array usando los números como subíndices. Después imprime las líneas ordenadas por sus números. Es un programa muy simple, y se confunde si encuenta números repetidos, huecos o líneas que no comiencen con un número.

```
{
  if ($1 > max)
    max = $1
  arr[$1] = $0
}

END {
  for (x = 1; x <= max; x++)
    print arr[x]
}
```

La primera regla guarda el número de línea más alto leído hasta el momento; también guarda cada línea en el array *arr*, en un índice que es el número de línea.

La segunda regla se ejecuta después de que se hayan leído toda la entrada, para imprimir todas las líneas. Cuando este programa se ejecuta con la siguiente entrada:

```
5  I am the Five man
2  Who are you?  The new number two!
4  . . . And four on the floor
1  Who is number one?
3  I three you.
```

su salida es la siguiente:

```
1  Who is number one?
2  Who are you?  The new number two!
```

```
3  I three you.
4  . . . And four on the floor
5  I am the Five man
```

Si se repite un número de línea, la última línea con dicho número repetido es la que permanece.

Los huecos en números de línea pueden ser manejados con una fácil mejora a la regla END del programa:

```
END {
  for (x = 1; x <= max; x++)
    if (x in arr)
      print arr[x]
}
```

Recorrido de todos los elementos de un Array

En programas que utilizan arrays, a menudo necesitas un bucle que se ejecute para cada elemento de un array. En otros lenguajes, donde los arrays son contíguos y los índices están limitados a enteros positivos, esto es fácil: el índice mayor es una unidad menor que el tamaño del array, y puedes encontrar todos los índices válidos recorriendo desde 0 hasta dicho valor. Esta técnica no servirá en awk, ya que cualquier número o cadena podría ser un índice de array. Así que awk tiene un tipo especial de sentencia *for* para el recorrido de arrays:

```
for (variable in array)
  cuerpo
```

Esto hace que se ejecute *cuerpo* una vez para cada valor diferente que su programa haya usado previamente como un índice en array, con la variable *variable* que recibe el valor de dicho índice.

Aquí tienes un programa que utiliza esta forma de la sentencia *for*. La primera regla escanea los registros de entrada y anota que palabras aparecen (al menos una vez) en la entrada, almacenando un 1 en el array, usando la palabra en cuestión como índice. La segunda regla escanea los elementos del array *used* para encontrar todas las palabras distintas que aparecen en la entrada. Imprime todas las palabras que son de más de 10 caracteres y también imprime el número de veces que aparecen dichas palabras en el fichero de entrada. Ver la sección 11. Funciones Implícitas (Built-in), para más información de la función implícita *length*.

```
# Record a 1 for each word that is used at least once.
{
  for (i = 1; i <= NF; i++)
    used[$i] = 1
}
# Find number of distinct words more than 10 characters long.
END {
  num_long_words = 0
  for (x in used)
    if (length(x) > 10) {
      ++num_long_words
      print x
  }
  print num_long_words, "words longer than 10 characters"
}
```

Ver la sección 17. Programas Ejemplo, para verun ejemplo más detallado de este tipo.

El orden en el cual esta sentencia accede a los elementos del array es determinado por la disposición interna de los elementos del array dentro de awk y no puede ser controlada ni cambiada. Esto puede llevar a problemas si se añaden nuevos elementos al *array* mediante sentencias en *cuerpo*, no se puede predecir si el bucle *for* recorrerá o no estos nuevos elementos. De forma similar, cambiando *variable* dentro del bucle podría producir resultados extraños.

La Sentencia delete

Puedes eliminar un elemento individual de un array utilizando la sentencia delete:

delete *array*[*index*]

Cuando un elemento de array es eliminado, es como sí nunca lo hubieses referenciado y nunca le hubieses dado un valor. Cualquier valor que tuviese el elemento del array eliminado nunca podrá ser obtenido.

Aquí está un ejemplo de la eliminación de elementos en un array:

```
for (i in frequencies)
  delete frequencies[i]
```

el ejemplo elimina todos los elementos del array`frequencies`.

Si eliminas un elemento, una sentencia for realizada a continuación para escanear el array no te devolverá dicho elemento y el operador in para chequear la existencia de un elemento te devolverá un 0:

```
delete foo[4]
if (4 in foo)
  print "Esto nunca será impreso"
```

Arrays Multi-Dimensionales

Un array multidimensional es un array en el cual un elemento es identificado por una secuencia de índices, no por un único índice. Por ejemplo, un array bidimensional requiere dos índices. La forma normal (en la mayoría de los lenguajes incluyendo *awk*) para referirse a un elemento de un array bidimensional llamado *grid* es mediante `grid[x,y]`.

Los arrays multidimensionales son soportados en awk mediante la concatenación de índices en una cadena. Lo que ocurre es que awk convierte los índices en cadenas (Ver la sección Conversiones de Cadenas y Números) y los concatena juntos, con un separador entre ellos. Esto crea una única cadena que describe el valor de los índices separados. La cadena combinada es usada como un índice único (normal) en un array unidimensional normal. El separador usado es el valor de la variable implícita SUBSEP.

Por ejemplo, supón que evaluamos la expresión `foo[5,12]="valor"` donde el valor de SUBSEP es "@". Los números 5 y 12 están concatenados con una coma entre ellos, produciendo `"5@12"`; por lo que, el elemento del array `foo["5@12"]` es fijado a `"valor"`.

Una vez que el valor del elemento es almacenado, awk no tiene forma de saber si se almacenó como un índice único o como una secuencia de índices. Las dos expresiones `foo[5,12]` y `foo[5 SUBSEP 12]` siempre tienen el mismo valor. El valor por defecto de SUBSEP es actualmente la cadena `"\034"`, que contiene un carácter no imprimible que es poco probable que aparezca en un programa awk o en los datos de entrada.

La falta de utilidad de elegir un carácter poco probable viene del hecho de que los valores de índices que contienen una cadena que concuerde con SUBSEP llevan a cadenas combinadas que son ambiguas. Supón que SUBSEP fuese "@"; entonces foo["a@b", "c"] y foo["a", "b@c"] serían indistinguibles porque ambas serían realmente almacenadas como foo["a@b@c"]. Debido a que SUBSEP es "\034", tales confusiones pueden ocurrir realmente solo cuando un índice contiene el carácter con código ASCII 034, lo cual es muy raro.

Puedes chequear si una secuencia de índice en particular existe en un array "multidimensional" con el mismo operador in utilizado para arrays de una sola dimensión. En lugar de un índice simple como el operando izquierdo, escribe la secuencia completa de índices separados por comas, en paréntesis:

(subscript1, subscript2, ...) in array

El siguiente ejemplo trata su entrada como un array bidimensional de campos; rota este array 90 grados en el sentido de las agujas del reloj e imprime el resultado. Asume que todas las líneas tienen el mismo número de elementos.

```
awk '{
     if (max_nf < NF)
          max_nf = NF
     max_nr = NR
     for (x = 1; x <= NF; x++)
          vector[x, NR] = $x
}

END {
     for (x = 1; x <= max_nf; x++) {
          for (y = max_nr; y >= 1; --y)
               printf("%s ", vector[x, y])
          printf("\n")
     }
}'
```

Cuando se le pasa la siguiente entrada:

```
1 2 3 4 5 6
2 3 4 5 6 1
3 4 5 6 1 2
4 5 6 1 2 3
```

produce la siguiente salida:

```
4 3 2 1
5 4 3 2
6 5 4 3
1 6 5 4
2 1 6 5
3 2 1 6
```

Recorrido de Arrays Multi-Dimensionales

No existe ninguna sentencia especial `for` que escanee un array multidimensional; no puede existir ninguna, porque en realidad no hay arrays o elementos multidimensionales; existe solo una forma de acceso a un array multidimensional.

Sin embargo, si tu programa tiene un array que es siempre accedido como multidimensional, puedes obtener el efecto de escanearlo combinando la sentencia de escaneo `for` (Ver la sección Recorrido de todos los elementos de un Array) con la función implícita `split` (Ver la sección Funciones Implícitas (Built-in) para Manipulación de Cadenas). Funciona de la siguiente manera:

```
for (indice_combinado in array) {
  split(índice_combinado, array_índices, SUBSEP)
  ...
}
```

Esto encuentra cada concatenación, índice combinado del array, y lo divide en índices individuales separándolos por las posiciones donde aparezca el valor de `SUBSEP`. Los índices separados se convierten en los elementos del array `array_índices`.

Por lo que, supón que has almacenado previamente en `array[1,"foo"]`; entonces un elemento con índice `"1\034foo"` existe en el array. (Recuerda que el valor por defecto de `SUBSEP` contiene el carácter con código `034`). Antes o después, la sentencia `for` encontrará ese índice y realizará una iteración en la cual `índice_combinado` tomará el valor `"1\034foo"`. Entonces la función `split` se llamará con los siguientes parámetros:

```
split("1\034foo", array_índices, "\034")
```

El resultado de esto es fijar el valor `1` para `array_índices[1]` y el valor `"foo"` para `array_índices[2]`. Por lo que, la secuencia original de índices separados ha sido recuperada.

11. Funciones Implícitas (Built-in)

Las funciones implícitas (Built-in) son funciones que están siempre disponibles para ser llamadas por tu programa awk. Este capítulo define todas las funciones implícitas de awk; algunas de ellas son mencionadas en otras secciones, pero ellas son recogidas aquí para tu conveniencia. (También se pueden crear funciones definidas por uno mismo. Ver sección Funciones definidas por el usuario)

Llamada a funciones implícitas (Built-in)

Para llamar a una función implícita, escribe el nombre de la función seguida por sus argumentos en paréntesis. Por ejemplo, atan2(y+z,1) es una llamada a la función atan2, con dos argumentos.

Los espacios en blanco entre el nombre de la función y los paréntesis son ignorados, pero se recomienda evitar poner espacios entre el nombre de la función y el paréntesis. Las funciones definidas por el usuario no pueden ser llamadas poniendo espacios en blanco entre el nombre de la función y el paréntesis, y te será más útil tomar el criterio de no poner espacios en blanco para cualquier tipo de llamada a función: **no dejar espacios en blanco detrás de un nombre de función**.

Cada función implícita acepta un cierto número de argumentos. En la mayoría de los casos, cualquier argumento extra que se le pase a la función implícita es ignorado. Los valores por defecto para argumentos omitidos varían de una función a otra y son descritos en cada una de las funciones.

Cuando se llama una función, las expresiones que crean los parámetros actuales de la función son evaluadas completamente antes de realizarse la llamada a la función. Por ejemplo, en el fragmento de código:

```
i = 4
j = sqrt(i++)
```

la variable i es fijada antes de que se produzca la llamada a la función sqrt, con un valor de 4 como parámetro actual.

Funciones Implícitas (Built-in) Numéricas

Aquí está una lista completa de funciones implícitas que trabajan con números:

int(*x*)	Esto te da la parte entera de x, truncado hacia 0. Esto produce el entero más cercado a x, localizado entre x y 0. Por ejemplo, `int(3)` es 3, `int(3.9)` es 3, `int(-3.9)` es -3, y `int(-3)` es -3 también.
sqrt(*x*)	Esto te da la raíz cuadrada positiva de x. Devuelve un error si x es un número negativo. Por lo tanto sqrt(4) es 2.
exp(x)	Esto te da el exponencial de x, o reporta un error si x está fuera de rango. Los rangos del valor x puede tener dependencias de la representación de los números flotante de tu máquina.
log(x)	Esto te da el logaritmo natural de x, si x es positivo; sino, devuelve un error.
sin(x)	Esta función te devuelve el seno de x, con x en radianes.
cos(x)	Te devuelve el coseno de x, con x en radianes.

atan2(y, x)	Esto te da el arcotangente de y/x, con el cociente entendido en radianes.

rand()	Esta función te da un número aleatorio. Los valores de rand son distribuídos unifórmemente entre 0 y 1.
	
	A menudo quieres enteros aleatorios en su lugar. Aquí está una función definida por el usuario que puedes usar para obtener un entero aleatorio no negativo menor que n:
	
	`function randint(n) {`
	`  return int(n * rand())`
	`}`
	La multiplicación produce un número real aleatorio mayor que 0 y menor que n. Nosotros entonces lo convertimos a entero (usando int) entre 0 y n-1.
	
	Aquí tienes un ejemplo donde un función similar se usa para producir números enteros entre 1 y n:
	
	`awk '`
	`# Function to roll a simulated die.`
	`function roll(n) { return 1 + int(rand() * n) }`
	`# Roll 3 six-sided dice and print total number of points.`
	`{`
	`  printf("%d points\n", roll(6)+roll(6)+roll(6))`
	`}'`
	Nota: `rand` comienza a generar números desde el mismo punto, o semilla, cada vez que tu ejecutas `awk`. Esto significa que un programa producirá los mismos resultados cada vez que los ejecutas. Los números son aleatorios dentro de una ejecución de `awk`, pero predecibles de una ejecución a otra. Esto es conveniente para la depuración, pero si quieres que un programa haga cosas diferentes cada vez que sea usado, debes cambiar la semilla a un valor que sea distinto cada vez que se ejecute. Para hacer esto, use srand.

srand(x)	La función srand fija el punto de inicio, semilla, para la generación de números aleatorios al valor de x. Cada valor de semilla lleva a una secuencia particular de números “aleatorios”. Por lo que, si fijas la semilla al mismo valor una segunda vez, obtendrás la misma secuencia de números “aleatorios” de nuevo. Si omites el argumento x, como en srand(), entonces la fecha y hora del día actuales son usados como semilla. Esta es la forma de obtener números aleatorios que sean realmente impredecibles. El valor de retorno de srand es la semilla previa. Esto hace más fácil el seguimiento de las semillas para el uso en la reproducción consistente de secuencias de números aleatorios.
time()	La función time (no presente en todas las versiones de `awk`) devuelve la hora actual en segundos desde el 1 de Enero de 1970.
Ctime(then)	La función ctime (no en todas las versiones de `awk`) toma un argumento numérico en segundos y devuelve una cadena que representa la fecha correspondiente, adecuada para impresión o un procesamiento posterior.

Funciones Implícitas (Built-in) para Manipulación de Cadenas

Las funciones en esta sección examinan el texto de una o más cadenas.

index(cadena, cad_buscar)

Esto busca la primera ocurrencia de la cadena *cad_buscar* en la cadena *cadena*, y devuelve la posición donde se produce esa ocurrencia dentro de la cadena *cadena*. Por ejemplo:

```
awk 'BEGIN { print index("peanut", "an") }'
```

devuelve un `3'. Si no se encuentra cad_buscar, index devuelve un 0.

length(cadena)

Esto te da el número de caracteres de la cadena *cadena*. Si *cadena* es un número, la longitud de la cadena de dígitos que representa dicho número es devuelta. Por ejemplo, length("abcde") es 5. En contraste, length(15*35) devuelve 3, que es la longitud de la cadena que se corresponde con el número resultante (525).

Si no se le suministran argumentos, length devuelve la longitud de $0 (la línea completa).

match(cadena, expreg)

La función match busca en la cadena, *cadena*, la más grande, y más a la izquierda subcadena que concuerde con la expresión regular, *expreg*. Devuelve la posición de carácter, o *índice*, de donce dicha subcadena empieza (1, si empieza al principio de *cadena*). Si no se encuentra una subcadena que concuerde con la expresión regular, *expreg*, devuelve un 0.

La función match fija la variable implícita RSTART al valor de índice. También fija la variable implícita RLENGTH a la longitud de la subcadena que concuerda con la expresión regular. Si no se encuentra ninguna coincidencia, RSTART es fijado a 0, y RLENGTH a –1.

Por ejemplo:

```
awk '{
       if ($1 == "FIND")
         regex = $2
       else {
         where = match($0, regex)
         if (where)
            print "La expresión regular ", regex, "encontrada en la posición ", where,
"en ", $0
       }
}'
```

Este programa busca las líneas que concuerdan con la expresión regular almacenada en la variable regex. Esta expresión regular puede ser cambiada. Si la primera palabra de la línea es 'FIND', regex se cambia a la segunda palabra de esa línea. Por lo tanto, dado:

```
FIND fo*bar
My program was a foobar
But none of it would doobar
```

```
FIND Melvin
JF+KM
This line is property of The Reality Engineering Co.
This file created by Melvin.
```

awk imprime:

```
La expresión regular fo*bar encontrada en posicion 18 en My program was a foobar
La expresión regular Melvin encontrada en posición 26 en This file created by Melvin.
```

split(cadena, array, fieldsep)

Esto divide *cadena* en trozos separados por *fieldsep*, y almacena los trozos en *array*. El primer trozo se almacena en array[1], el segundo trozo en array[2], y así hasta el final. El valor cadena del tercer argumento, *fieldsep*, se utiliza como una expresión regular para buscar los sitios por los que particionar *cadena*. Si se omite el *fieldsep*, se utiliza el valor de FS. *Split* devuelve el número de elementos creados.

La función split, entonces, parte cadenas en trozos en una forma similar a la forma en la que las líneas de entrada son divididas en campos. Por ejemplo:

```
split("auto-da-fe", a, "-")
```

parte la cadena 'auto-da-fe' en tres campos usando '-' como separador. Fija los valores del array *a* tal y como sigue:

```
a[1] = "auto"
a[2] = "da"
a[3] = "fe"
```

el valor devuelto por esta llamada a la función split es 3.

sprintf(formato, expresion1, expresion2, ...)

Esta función devuelve (sin imprimirla) la cadena que la función printf habría impreso utilizando los mismos argumentos (ver la sección Uso de sentencias printf para una impresión más elegante). Por ejemplo:

```
sprintf("pi = %.2f (aprox.)", 22/7)
```

devuelve la cadena `"pi = 3.14 (aprox.)"`.

sub(expreg, replacement, cad_destino)

La función sub altera el valor de *cad_destino*. Busca este valor, el cuál debería ser una cadena, en la subcadena más a la izquierda que concuerde con la expresión regular, *expreg*, extendiendo esta concordancia tan lejos como fuese posible. Entonces la cadena completa se cambia reemplazando el texto encajado por *replacement*. La cadena modificada se convierte en el nuevo valor de *cad_destino*.

Esta función es peculiar porque *target* no es simplemente usada para calcular un valor: debe ser una variable, campo o referencia de array, de forma que **sub** pueda almacenar un valor modificado en la misma. Si se omite este argumento, entonces por defecto se usa y modifica $0. Por ejemplo:

```
str = "water, water, everywhere"
sub(/at/, "ith", str)
```

fija `str` a `"wither, water, everywhere"`, reemplazando la ocurrencia más a la izquierda y más larga de `'at'` por `'ith'`.

La función sub devuelve el número de substituciones hechas (o 1 o 0)

Si el carácter especial '&' aparece en *replacement*, el mantiene la subcadena precisa que concordó con regexp. (Si la expresión regular expreg puede encajar con más de una cadena, entonces esta subcadena precisa podría variar.) For example:

```
awk '{ sub(/candidate/, "& and his wife"); print }'
```

cambia la primera ocurrencia de 'candidate' a 'candidate and his wife' en cada línea de entrada.

El efecto de este carácter especial puede ser desactivado poniendo una barra invertida antes de él en la cadena. Como es usual, para insertar una barra invertida en la cadena, debes escribir dos barras invertidas. Por lo tanto, escribe '\\&' en una cadena constante para incluir un literal '&' en el replacement. Por ejemplo, aquí tienes como reemplazar el primer '|' de cada línea por un '&':

```
awk '{ sub(/\|/, "\\&"); print }'
```

Nota: como se mencionó anteriormente, el tercer argumento de sub debe ser un *valori*. Algunas versiones de awk permiten que el tercer argumento sea una expresión la cual no sea un *valori*. En tales casos, sub buscaría de todas formas el patrón y devolvería un 0 o un 1, pero el resultado de la substitución (si existiese) no sería tenido en cuenta porque no hay un lugar para colocarlo. Tales versiones de awk aceptas expresiones tales como esta:

```
sub(/USA/, "United States", "the USA and Canada")
```

Pero esa es considerada erronea en gawk.

gsub(regexp, replacement, target)

Esta es similar a la función sub, excepto que gsub reemplaza todas las concordancias que encuentre con la expresión regular, *regexp* (reemplaza varias concordancias en una misma línea o registro). El 'g' en gsub significa "global", lo cual significa reemplazar en cualquier sitio. Por ejemplo:

```
awk '{ gsub(/Britain/, "United Kingdom"); print }'
```

reemplaza todas las ocurrencias de la cadena 'Britain' por 'United Kingdom' para todos los registros de entrada. La función gsub devuelve el número de substituciones realizadas. Si la variable a ser buscada y alterada, *target*, se omite, entonces se utiliza el registro de entrada completo, $0.

Como en sub, los caracteres `&' y `\' son especiales, y el tercer argumento debe ser un *valori*.

substr(string, start, length)

Esta función devuelve una subcadena de longitud *length* de *string*, empezando el el carácter número *start*. El primer carácter de una cadena es el carácter número 1. Por ejemplo, substr("washington", 5, 3) devuelve "ing".

Si no se especifica *length*, esta función devuelve el sufijo completo de *cadena* que comienza en el carácter número *start*. Por ejemplo, substr("washington", 5) devuelve "ington".

tolower(string)

Esto devuelve una copia de *string*, con cada carácter en mayúsculas de la cadena reemplazado por su correspondiente carácter en minúscula. Los caracteres no alfabéticos no son cambiados. Por ejemplo, tolower("MiXeD cAsE 123") devuelve "mixed case 123".

toupper(string)

Este devuelve una copia de *string*, con cada carácter en minúscula de la cadena reemplazado por su correspondiente carácter en mayúscula. Los caracteres no alfabéticos no son cambiados. Por ejemplo, `toupper("MiXeD cAsE 123")` devuelve `"MIXED CASE 123"`.

Funciones Implícitas (Built-in) para Entrada/Salida

close(filename)

Cierra el fichero filename, para entrada o salida. El argumento podría ser de igual forma un comando de shell que fuese usado para redireccionar a o desde un pipe; entonces el pipe es cerrado.

Ver la sección Cerrado de Ficheros de Entrada y Pipes para saber más sobre el cierre de ficheros y pipes de entrada. Ver la sección Cerrando los Ficheros de Salida y Pipes con respecto al cierre de ficheros y pipes de Salida.

system(comando_sistema)

La función system permite al usuario ejecutar comandos del sistema operativo y después volver al programa awk. La función system ejecuta el comando dado por la cadena *comando_sistema*. Devuelve, como su valor, el status devuelto por el comando que fue ejecutado.

Por ejemplo, si el siguiente fragmento de código es puesto en tu programa awk:

```
END {
     system("mail -s 'awk run done' operator < /dev/null")
}
```

le será enviado un correo al operador de sistema cuando el programa awk acabe el procesamiento de la entrada y comience su procesamiento de final de entrada.

Dese cuenta de que dicho resultado podría ser también obtenido redireccionando print o printf a un pipe. Sin embargo, si tu programa awk es interactivo, system es útil para seguir programas grandes autocontenidos. Tales como un shell o un editor.

Algunos sistemas operativos no pueden implementar la función system. System provoca un error fatal si no es soportada.

12. Funciones definidas por el Usuario

Programas awk complicados pueden ser simplicados a menudo definiendo tus propias funciones. Las funciones definidas por el usuario pueden ser llamadas del mismo modo que las funciones implícitas. (Ver la sección Llamada a funciones implícitas (Built-in), pero tú eres el encargado de definirlas y decirle a awk qué tiene que hacer.

Sintaxis de las Definiciones de Funciones

La definición de funciones puede aparecer en cualquier parte entre las reglas de un programa awk. Po lo que la forma general de un programa awk se extiende para incluir secuencias de reglas y definiciones de funciones definidas por el usuario.

La definición de una función sería algo como esto:

```
function nombre_función (lista_de_parámetros) {
     cuerpo-de-la-función
}
```

La palabra clave `function` puede ser abreviada a `func`.

- **`Nombre_función`** es el nombre de la función que se va a definir. Un nombre de función válido tiene la misma forma que un nombre de variable válido: una secuencia de letras, dígitos y subrayados, siempre que no comience con un dígito.
- **`Lista_de_parámetros`** es una lista de los argumentos de las funciones y nombres de variables locales, separadas por comas. Cuando se llama la función, los nombres de los argumentos se usan para guardar los valores de los argumentos pasados en la llamada a la función. Las variables locales son inicializadas a la cadena nula.
- El **`cuerpo-de-la-función`** consiste en sentencias de awk. Es la parte más importante de la definición, porque dice lo que la función debería hacer realmente. Los nombres de argumentos existen para darle al cuerpo una forma de referirse a los argumentos; variables locales, para darle al cuerpo sitios donde guardar valores temporales.

Los nombres de argumentos no se distinguen sintácticamente de los nombres de variables locales; en su lugar, el número de argumentos suministrados cuando se llama a la función determina cuantas variables argumento hay. Por lo que, si se dan tres valores de argumento, los tres primeros nombres en *lista-de-parámetros* son argumentos, y el resto son variables locales.

Por lo que si el número de argumentos no es el mismo en todas las llamadas que tenga la función, algunos de los nombres en *lista-de-parámetros* podrían ser argumentos en algunas ocasiones y variables locales en otras ocasiones. Otra forma de enfocar esto es que los argumentos omitidos tienen el valor por defecto de cadenas nulas.

Normalmente cuando escribes una función sabes cuantos nombres tratas de usar como argumentos y cuantos tratas de usar como variables locales. Por convención, deberías escribir un espacio extra entre los argumentos y las variables locales, de forma que otras personas puedan serguir como se supone que se usa la función.

Durante la ejecución del cuerpo de la función, los valores de los argumentos y variables locales esconden cualquier variable del mismo nombre usada en el resto del programa. Las variables que tienen el mismo nombre que las

variables locales de la función no son accesibles en la definición de la función, porque no existe ninguna forma de nombrarlas debido a que sus nombres son usados para variables locales. El resto de las variables usadas en el programa awk pueden ser referenciadas o fijadas de forma normal en la definición de la función.

Los argumentos y variables locales permanecen solo mientras se esté ejecutando el cuerpo de la función. Una vez que el cuerpo de la función acaba, las variables del programa principal que se llaman igual que las variables locales de la función se pueden usar de nuevo.

El cuerpo de la función puede contener expresiones con llamadas a funciones. Pueden incluso llamarse a si misma directamente o a través de un función intermedia, a estas funciones se les dice que son *recursivas*.

No existe la necesidad en awk de poner la definición de la función antes de usarla. Esto se debe a que awk lee el programa completo antes de comenzar a ejecutarlo.

Ejemplo de Definición de Función

Aquí se muestra un ejemplo de función definida por el usuario, llamada *miprint*, la cual toma como parámetro un número y lo imprime en un formato especifico.

```
function miprint (num)
{
     printf "%6.3g\n", num
}
```

Para ilustrar una llamada a la función `miprint`, aquí se presenta una regla awk que usar nuestra función *miprint*:

```
$3 > 0     { miprint($3) }
```

Este programa imprime, en nuestro formato especial, todos los terceros campos que contengan un número positivo en nuetra entrada. Por lo tanto, cuando se le da:

```
 1.2   3.4   5.6   7.8
 9.10 11.12 13.14 15.16
17.18 19.20 21.22 23.24
```

este programa, usando nuestra función para formatear el resultado, imprime:

```
   5.6
  13.1
  21.2
```

Aquí está un ejemplo mejor de una función recursiva. Imprime una cadena al revés:

```
function rev (str, len) {
    if (len == 0) {
        printf "\n"
        return
    }
    printf "%c", substr(str, len, 1)
    rev(str, len - 1)
}
```

Llamada a Funciones definidas por el Usuario

La llamada a una función hace que se ejecute la función y realice su trabajo. Una llamada a función es una expresión, y su valor es el valor devuelto por la función.

Una llamada a función consiste en el nombre de la función seguido por los argumentos entre paréntesis. Lo que escribes en la llamada para los argumentos son expresiones `awk`; cada vez que se ejecute la llamada, estas expresiones son evaluadas, y sus valores son los argumentos actuales. Por ejemplo, aquí aparece una llamada a *foo* con tres argumentos:

```
foo(x y, "lose", 4 * z)
```

Nota: los espacios en blanco (espacios y tabuladores) no están permitidos entre el nombre de la función y el paréntesis de apertura de la lista de argumentos. Si escribes un espacio en blanco por error, `awk` podría pensar que deseas concatenar una variable con la expresión entre paréntesis. Sin embargo, te avisa de que usaste un nombre de función en lugar de un nombre de variable, y devuelve un error.

Cuando una función es *llamada*, se le proporciona una copia de los valores de sus argumentos. A esto se le llama *llamada por valor*. El llamador podría usar una variable como la expresión para el argumento, pero la función llamada no sabe esto: todo lo que sabe es el valor que tiene el argumento. Por ejemplo, si escribes este código:

```
foo = "bar"
z = myfunc(foo)
```

entonces no deberías pensar en el argumento de `myfunc` como si fuese "la variable `foo`". En su lugar, piensa en el argumento como la cadena, `"bar"`.

Si la función `myfunc` altera los valores de sus variables locales, esto no tiene efecto en cualquier otra variable. En particular, si `myfunc` hace esto:

```
function myfunc (win) {
  print win
  win = "zzz"
  print win
}
```

para cambiar su primer variable argumento `win`, esto *no cambia* el valor de `foo` de la llamada. La función de `foo` en la llamada a `myfunc` acabó cuando su valor, `"bar"`, se procesó. Si `win` existe también fuera de `myfunc`, el cuerpo de la función no puede alterar este valor externo, porque está escondido durante la ejecución de `myfunc` y no puede ser visto o alterado desde allí.

Sin embargo, cuando los parámetros de las funciones son arrays, ellos ***no pueden ser copiados***. En su lugar, el mismo array está disponible para la manipulación directa por parte de la función. A esto se le llama normalmente ***llamada por referencia***. Los cambios realizados a un parámetro array dentro del cuerpo de la función *son visibles* fuera de dicha función. *Esto podría ser peligroso si no te fijas en lo que estás haciendo*. Por ejemplo:

```
function changeit (array, ind, nvalue) {
     array[ind] = nvalue
}

BEGIN {
```

```
        a[1] = 1 ; a[2] = 2 ; a[3] = 3
        changeit(a, 2, "two")
        printf "a[1] = %s, a[2] = %s, a[3] = %s\n", a[1], a[2], a[3]
    }
```

imprime `a[1] = 1, a[2] = two, a[3] = 3', porque la llamada a changeit almacena "two" en el segundo elemento de a.

La Sentencia return

El cuerpo de las funciones definidas por el usuario puede contener una sentencia *return*. Esta sentencia devuelve el control al resto del programa awk. Puede ser también usada para devolver un valor para que sea usado en el resto del programa. Esta sentencia presenta la siguiente forma:

return *expresión*

La parte *expresión* es opcional. Si se omite, entonces el valor devuelto es indefinido y, por lo tanto, impredecible. Una sentencia *return* sin valor de expresión se asume por defecto al final de todas las definiciones de función. Así que si el control llega al final de la definición de función, entonces la función devuelve un valor impredecible.

Aquí está un ejemplo de una función definida por usuario que devuelve el valor del número mayor contenido entre los elementos de un array:

```
function maxelt (vec,   i, ret) {
    for (i in vec) {
        if (ret == "" || vec[i] > ret)
            ret = vec[i]
    }
    return ret
}
```

Llamas a maxelt con un argumento, un nombre de array. Las variables locales i y ret no intentan ser argumentos; mientras no exista nada que te detenga de pasar dos o más argumentos a maxelt, los resultados serían extraños. El espacio extra antes de i en la lista de parámetros de la función es para indicar que i y ret no se suponen que sean argumentos. Esta es una convención que deberías seguir cuando definas funciones.

Aquí está un programa que usa nuestra función maxelt. Carga un array, llama maxelt, y entonces reporta el número máximo de ese array:

```
awk '
function maxelt (vec,   i, ret) {
    for (i in vec) {
        if (ret == "" || vec[i] > ret)
            ret = vec[i]
    }
    return ret
}

# Load all fields of each record into nums.
```

```
{
          for(i = 1; i <= NF; i++)
               nums[NR, i] = $i
}

END {
     print maxelt(nums)
}'
```

Dada la siguiente entrada:

```
 1 5 23 8 16
44 3 5 2 8 26
256 291 1396 2962 100
-6 467 998 1101
99385 11 0 225
```

nuestro programa nos informa (predeciblemente) que:

```
99385
```

es el número mayor en nuestro array.

13. Variables Implícitas (Built-in)

La mayorías de las variables awk están disponibles para que las uses para tus propios propósitos; nunca cambian excepto cuando se le asigna por programa un nuevo valor, y nunca tienen efectos sobre el programa salvo cuando tu programa las examina.

Unas pocas variables tienen un significado especial implícito (built-in). Algunas de ellas son examinadas por awk automáticamente, de forma que te ofrecen la posibilidad de decirle a awk como tiene que realizar ciertas cosas. Otras reciben un valor de forma automática por parte de awk, de forma que te pueden ofrecer información de cómo está funcionando internamente tu programa awk.

Este capítulo documenta todas las variable implícitas de gawk. La mayoría de ellas son también descritas en los capítulos donde sus areas de actividad son descritas.

Variables Implícitas (Built-in) que controlan el comportamiento de awk

Esta es una lista de variables las cuales tú puedes cambiar para controlar como realiza ciertas cosas awk.

FS

FS es el separador de campos de la entrada (ver la sección Especificando como están separados los campos). El valor es un único carácter o una expresión regular multi-carácter que busca las separaciones entre campos en un registro de entrada.

El valor por defecto es " ", una cadena consistente en un único espacio en blanco. Como excepción especial, este valor significa realmente que cualquier secuencia de espacios y tabuladores forman un único separador. Hace también que los espacios y tabuladores al principio de línea sean ignorados.

Puedes fijar el valor de FS en la línea de comando usando la opción '-F':

```
awk -F, 'programa' ficheros_de_entrada
```

IGNORECASE

Si IGNORECASE es distinta de cero, entonces todas las búsquedas de expresiones regulares se realizan de una forma insensible a las mayúsculas/minúsculas. En particular, las expresiones regulares comparadas con '~' y '!~' y las funciones gsub, index, match, split y sub ignoran las diferencias entre mayúsculas y minúsculas cuando realizan sus operaciones de expresiones regulares particulares. **Nota:** ya que la partición en campos con el valor de la variable FS es también una operación de expresión regular, también se ve afectada por la desactivación/activación de distinción entre mayúsculas/minúsculas. Ver la sección Sensibilidad a Mayúsculas en el Matching.

Si awk está en modo compatibilidad (ver la sección 14. Invocación de awk), entonces IGNORECASE no tiene ningún significado especial, y todas las operaciones de expresiones regulares son sensibles a la distinción entre Mayúsculas/Minúsculas.

OFMT

Esta cadena es usada por awk para controlar la conversión de números a cadenas (ver la sección Conversiones de Cadenas y Números). Funciona siendo pasada, en efecto, como el primer argumento a la función sprintf. Su valor por defecto es “%.6g”.

OFS

Es el separador de los campos de salida (ver la sección Separadores de la Salida). Es sacada entre cada dos campos de la salida por una sentencia print. Su valor por defecto es “ “, una cadena de un solo espacio.

ORS

Este es el separador de los registros de salida. Es colocada al final de cada sentencia print. Su valor por defecto es una cadena que contiene un único carácter de nueva línea, el cual podría ser también escrito como “\n” (ver la sección Separadores de la Salida)

RS

Este es el separador de registros de awk. Su valor por defecto es una cadena que contiene un único carácter newline, lo que significa que cada registro de entrada consiste en una única línea de texto. (Ver la sección Cómo se particiona la Entrada en Registros)

SUBSEP

SUBSEP es un separador de subíndice. Tiene por defecto el valor “\34”, y es usado para separar las partes del nombre de un array multidimensional. Por lo que si accedes a foo[12,3], internamente accede realmente a foo[“12\0343”] (Ver la sección Arrays Multi-Dimensionales).

Variables Implícitas (Built-in) que te proporcionan información

Esta es una lista de variables que son fijadas automáticamente por awk en ciertas ocasiones, así que proporcionan información a tu programa.

ARGC, ARGV

Los argumentos de la línea de comandos disponibles para awk son almacenados en un array llamado ARGV. ARGC es el número de argumentos de la línea de comando presentes. ARGV está indexado desde 0 a ARGC-1. Ver la sección 14. Invocación de awk. Por ejemplo:

```
awk '{ print ARGV[$1] }' inventario-enviado Lista-BBS
```

En este ejemplo, ARGV[0] contiene “awk”, ARGV[1] contiene “inventario-enviado”, y ARGV[2] contiene “Lista-BBS”. El valor de ARGC es 3, uno más que el índice del último elemento en ARGV ya que el primer elemento es el 0.

Dese cuenta de que el programa awk no se entra en ARGV. Las otras opciones de la línea de comando especiales, con sus argumentos, tampoco son entradas. Pero la asignación de variables en la línea de comandos son tratadas como argumentos, y se podrían ver por lo tanto en el array ARGV.

Tu programa puede alterar ARGC y los elementos de ARGV. Cada vez que awk alcanza el final de un fichero de entrada, utiliza el siguiente elemento de ARGV como el nombre del siguiente fichero de entrada. Almacenando un cadena distinta en ARGV, se puede cambiar los ficheros que van a ser leidos por tu programa. Puedes usar “-“ para

representar la entrada estándar. Almacenando elementos adicionales e incrementando ARGC puedes causar que se procesen ficheros adicionales.

Si decrementas el valor de ARGC, eso elimina ficheros de entrada del final de la lista de ficheros de entrada a procesar. Grabando el antiguo valor de ARGC en algún lugar, tu programa puede tratar los argumentos eliminados como otra cosa en lugar de cómo nombres de ficheros.

Para eliminar un fichero de mitad de la lista, almacena la cadena nula ("") en ARGV en lugar del nombre de fichero que quieres que no se procese. Como una característica especial, awk ignora los nombres de ficheros que han sido reemplazados por una cadena nula.

ENVIRON

Este es un array que contiene los valores del entorno. Los índices del array son los nombres de las variables de entorno; los valores son los valores de las variables de entorno particulares. Por ejemplo, ENVIRON["HOME"] podría ser '/home/dwadmin'. El cambio de los valores de este array no afecta al valor real de las variables que tengan en el sistema operativo.

Algunos sistemas operativos podrían no tener variables de entorno. En tales sistemas, el array ENVIRON está vacío.

FILENAME

Este es el nombre del fichero que awk está leyendo en la actualidad. Si awk está leyendo de la entrada estándar (en otras palabras, no se han especificado ficheros en la línea de comando), FILENAME toma el valor de "-". FILENAME se cambia cada vez que se lee desde un fichero nuevo (Ver la sección 3. Leyendo ficheros de entrada)

FNR

FNR es el número de registro del registro actual en el fichero actual. FNR se incrementa automáticamente cada vez que se lee un nuevo registro. (Ver la sección Entrada explícita con getline). Es reinicializado a 0 cada vez que se comienza un nuevo fichero de entrada.

NF

NF es el número de campos del registro de entrada actual. NF recibe un nuevo valor cada vez que se lee un nuevo registro, cuando se crea un nuevo campo, o cuando $0 cambia (Ver la sección Examinando campos).

NR

Este es el número de registros de entrada que han sido procesados desde el comienzo de la ejecución del programa. (Ver la sección Cómo se particiona la Entrada en Registros). NR se incrementa en una unidad cada vez que se lee un nuevo registro.

RLENGTH

RLENGTH es la longitud de la subcadena encontrada por la función *match* (Ver la sección Funciones Implícitas (Built-in) para Manipulación de Cadenas). RLENGTH toma un valor cada vez que se llama a la función *match*. Su valor es la longitud de la cadena encontrada, o –1 si no se encontró ninguna cadena.

RSTART

RSTART es el índice de comienzo de la subcadena encontrada por la función *match* (Ver la sección Funciones Implícitas (Built-in) para Manipulación de Cadenas). RSTART toma un valor cuando se llama a la función *match*. Su valor es la posición de la cadena donde la subcadena encontrada comienza, o 0 si no se encontró.

14. Invocación de awk

Hay dos forma de ejecutar awk: con un programa explícito, o con uno o más ficheros de programas. Aquí aparecen ambas formas; las partes encerradas entre '[...]' son opcionales.

```
awk [-Ffs] [-v var=valor] [-V] [-C] [-c] [-a] [-e] [--] 'program' fichero...
```

```
awk [-Ffs] -f fichero-fuente [-f fichero-fuente ...] [-v var=valor] [-V] [-C] [-c] [-a]
[-e] [--] fichero...
```

Opciones de la línea de comandos

Las opciones comienzan con un signo menos, y consisten en un único carácter. Las opciones y sus significados son los siguientes:

-Ffs	Fija la variable FS al valor fs (ver la sección Especificando como están separados los campos).
-f *source-file*	Indica que el program awk se encuentra en el *fichero-fuente* en lugar de en el primer argumento que no es opción.
-v *var=valor*	Le asigna a la variable *var* el valor *valor* antes de que comience la ejecución del programa. Dichos valores de variables están disponibles dentro de la regla BEGIN (mirar más abajo para una explicación más extensa). La opción '-v' solo puede fijar el valor de una variable, pero se pueden especificar tantas variables como se deseen mediante el uso repetido de esta opción: `-v foo=1 -v bar=2'.
-a	Especifica el uso de la sintáxis de awk tradicional para las expresiones regulares. Esto significa que '\' puede ser usado para quote cualquier expresión regular dentro de corchetes, tal y como sería fuera de ellos. Este modo es actualmente el modo por defecto. Ver la sección Operadores de Expresiones Regulares.
-e	Especifica el uso de la sintáxis de egrep para las expresiones regulares. Esto significa que '\' no sirve como un carácter de quoting dentro de los corchetes; técnicas ideosincráticas son necesarias para incluir varios caracteres especiales dentro de ellas. Este modo podría convertirse en el modo por defecto en el futuro. Ver la sección Operadores de Expresiones Regulares.
-c	Especifica *modo compatibilidad*, en el cuál las extensiones GNU en gawk son deshabilitadas, de forma que gawk se comporta como el awk de Unix. Ver la sección Compatibilidad hacia atrás y Depuración.
-V	Muestra información de la versión para nuestra copia particular de gawk. De esta forma puedes determinar si tu copia de gawk está actualizada con respecto a la que está distribuyendo actualmente la Fundación de Software Gratuito. Esta opción podría desaparecer en una versión futura de gawk.
-C	Muestra la versión corta de la Licencia Pública General. Esta opción podría desaparecer en futuras versiones de gawk.

--	Señala el final de las opciones de la línea de comando. Los siguientes argumentos no son tratados como opciones incluso aunque empezasen con un guión '-'. Esta interpretación de '--' sigue las convenciones de parsing de argumentos POSIX. Esto es útil si tienes nombres de ficheros que comiencen con un signo '-', o en shell scripts, si tienes nombres de ficheros que serán especificados por el usuario y que podrían empezar con '-'.

Cualquier otra opción se avisa que es inválida con un mensaje de advertencia, pero sería ignorada de todas formas.

En modo compatibilidad, como caso especial, si el valor de *fs* suministrado a la opción '-F' es 't', entonces FS toma el carácter tabulador ("\t"). También, las opciones '-C' y '-V' no son reconocidas.

Si no se usa la opción '-f', entonces el primer argumento de la línea de comando que no sea opción se espera que sea el programa.

La opción '-f' podría ser usada más de una vez en la línea de comando. Entonces awk lee su programa fuente de todos los ficheros nombrados, tal y como si fuesen concatenados juntos en un único fichero grande. Esto es útil para crear librerías de funciones awk. Funciones útiles pueden ser escritas una vez, y entonces recuperadas desde un lugar estándar, en lugar de tener que ser incluídas en cada programa individual. Puedes aún teclear un programa en el terminal y usar funciones de librería, especificando '-f /dev/tty'. Awk leerá un fichero desde el terminal para usarlo como parte del programa awk. Después de teclear tu programa, teclee **Control-d** (el carácter final de fichero) para finalizarlo.

Otros argumentos de la línea de comandos

Cualesquiera argumentos adicionales en la línea de comando son tratados normalmente como ficheros de entrada a ser procesados en el orden especificado. Sin embargo, un argumento tiene la forma *variable*=*valor*, lo que significa asignar el valor *valor* a la variable *variable*.

Todos estos argumentos están disponibles para tu programa awk en el array ARGV (Ver la sección 13. Variables Implícitas (Built-in). Las opciones de la línea de comando y el texto del programa (si está presente) son omitidos del array ARGV. Todos los otros argumentos, incluído la asignación de variables son incluídos en el array ARGV.

La distinción entre los argumentos nombres de ficheros y argumentos de asignación de variables se hace cuando awk va a abrir el siguiente fichero de entrada. En ese punto de la ejecución, chequea el "nombre de fichero" para ver si es realmente un asignación de variable; si es así, awk fija el valor de la variable en lugar de leer el fichero.

Por lo tanto, las variables reciben realmente su valor, los valores especificados, después de que todos los ficheros especificados previamente hayan sido leídos. En particular, los valores de variables asignadas en esta fashion *no están disponibles* en la regla BEGIN (Ver la sección Los Patrones Especiales BEGIN y END), ya que tales reglas se ejecutan antes de que awk comienze el escaneo de la lista de argumentos.

En algunas implementaciones previas de awk, cuando la asignación de variable ocurría antes de cualquiera de los nombres de ficheros, la asignación sucedería *antes* de que se ejecute la regla BEGIN. Algunas aplicaciones pueden ser dependientes de esta "*característica*".

La característica de asignación de variable es más útil para asignar valores a variables tales como RS, OFS y ORS, las cuales controlan los formatos de entrada y salida, antes del escaneo de los ficheros de datos. También es útil para controlar el estado si se necesitan múltiples pasadas sobre un fichero de datos. Por ejemplo:

```
awk 'pass == 1  { pass 1 stuff }
     pass == 2  { pass 2 stuff }' pass=1 datafile pass=2 datafile
```

La variable de entorno AWKPATH

La sección anterior describió como los ficheros de programas awk pueden ser especificados desde la línea de comando con la opción '-f'. En algunas implementaciones de awk, debes suministrar el path y nombre de fichero preciso para cada fichero de programa, a menos que el fichero se encuentre en el directorio actual.

Pero en gawk si el nombre de fichero suministrado en la opción '-f' no contiene el carácter '/', entonces gawk busca en una lista de directorios (llamados el path o camino de búsqueda), uno por uno, buscando un fichero con el nombre especificado.

El camino de búsqueda es realmente una cadena que contiene directorios separados por dos puntos. Gawk obtiene su camino de búsqueda de la variable de entorno AWKPATH. Si la variable no existe, gawk utiliza el path por defecto, el cual es: `.:/usr/lib/awk:/usr/local/lib/awk'.

La característica de camino de búsquea es particularmente útil para la construcciones de útiles librerías de funciones de awk. Los ficheros de librería pueden ser colocados en un directorio estándar que esté en el camino por defecto, y después especificarlo en la línea de comando con un nombre de fichero corto. Si no, habría que teclear el nombre de fichero completo para cada fichero.

La búsqueda de path no es realizada por gawk en modo compatibilidad. Ver la sección de 14. Invocación de awk.

Nota: Si quieres que se encuentren ficheros del directorio actual, debes incluir el directorio actual en el path, o escribiendo '.' o escribiendo el path completo del directorio. (Una entrada nula se indica comenzando o empezando el path con dos puntos ':', o colocando dos caracteres de dos puntos consecutivos ('::')). Si el directorio actual no está incluído en el path, entonces los ficheros no pueden ser encontrados en el directorio actual. Este mecanismo de búsqueda de path es identico al de la shell.

15. La evolución del Lenguaje awk

Este manual describe la implementación GNU de awk, la cuál está implementada después de la versión de System V Release 4. Muchos usuarios de awk están solamente familiarizados con la implementación original de awk en la Versión 7 Unix, la cuál es también la base de la versión en el Unix de Berkeley. Este capítulo describe en detalle la evolución del lenguaje awk.

Cambios Mayores entre V7 y S5R3.1

El lenguaje awk ha evolucionado considerablemente entre las release de la versión 7 de Unix (1978) y la nueva versión ampliamente extendida en el System V Release 3.1 (1987). Esta sección resume los cambios, con referencias cruzadas a las secciones correspondientes.

- El requisito del ';' para separar reglas en una misma línea (Ver la sección Sentencias frente a Líneas en awk).
- Las funciones definidas por el usuario, y la sentencia `return` (Ver la sección 12. Funciones definidas por el Usuario)
- La sentencia `delete` (Ver la sección La Sentencia delete).
- La sentencia `do-while` (Ver la sección La Sentencia do-while).
- Las funciones implícitas `atan2, cos, sin, rand y srand` (Ver la sección Funciones Implícitas (Built-in) Numéricas).
- Las funciones implícitas `gsub, sub y match` (Ver la sección Funciones Implícitas (Built-in) para Manipulación de Cadenas).
- Las funciones implícitas `close y system` (Ver la sección Funciones Implícitas (Built-in) para Entrada/Salida).
- Las variables implícitas `ARGC, ARGV, FNR, RLENGTH, RSTART y SUBSEP` (Ver la sección 13. Variables Implícitas (Built-in).
- Las expresiones condicionales que usan los operadores '?' y ':' (Ver la sección Expresiones Condicionales).
- El operador exponencial '^' (Ver la sección Operadores Aritméticos). y su forma como operador de asignación '^=' (Ver la sección Expresiones de Asignación).
- Precedencia de operadores compatible con C (Ver la sección Precedencias de Operadores: Cómo se anidan los Operadores).
- Expresiones regulares como valor de FS (Ver la sección Especificando como están separados los campos) o como tercer argumento de la función `split`. (Ver la sección Funciones Implícitas (Built-in) para Manipulación de Cadenas).
- Expresiones regulares dinámicas como operandos de los operadores `'~'` y `'!~'` (Ver la sección Cómo usar Expresiones Regulares).
- Secuencias de Escape (Ver la sección Expresiones Constantes) en las expresiones regulares.
- Las secuencias de escape `'\b', '\f' y '\r'` (Ver la sección Expresiones Constantes).
- La redirección de la entrada de la función `getline` (Ver la sección Entrada explícita con getline).

- Reglas BEGIN y END múltiples (Ver la sección Los Patrones Especiales BEGIN y END).
- Simulación de arrays multidimensionales (Ver la sección Arrays Multi-Dimensionales).

Cambios Menores entre S5R3.1 y S5R4

La versión del awk de Unix de System V Release 4 añadió estas características:

- La variable ENVIRON (Ver la sección 13. Variables Implícitas (Built-in).
- Múltiples opciones '-f' en la línea de comando (Ver la sección 14. Invocación de awk).
- La opción '-v' para asignar variables antes de que comience la ejecución del programa. (Ver la sección 14. Invocación de awk).
- La opción '--' para terminar las opciones de la línea de comando.
- Las secuencias de escape `'\a'`, `'\v'` y `'\x'`. (Ver la sección Expresiones Constantes).
- Un valor de retorno definido para la función implícita `srand` (Ver la sección Funciones Implícitas (Built-in) Numéricas).
- Las funciones implícitas `toupper` y `tolower` para pasar de mayúsculas a minúsculas y a la inversa (Ver la sección Funciones Implícitas (Built-in) para Manipulación de Cadenas).
- Una especificación más clara para la letra de control de formato `'%c'` en la función `printf` (Ver la sección Uso de sentencias printf para una impresión más elegante).
- El uso de expresiones regulares constantes tales como `/foo/` como expresiones, donde son equivalentes al uso del operador de encaje, como en `$0 ~ /foo/`.

Extensiones en gawk que no están en S5R4

La implementación GNU, gawk, añade las siguientes caracteres:

- La variables de entorno `AWKPATH` para especificar un path de búsqueda para la opción de la línea de comando '-f' (Ver la sección 14. Invocación de awk).
- Las opciones de la línea de comando '-C' y '-V' (Ver la sección 14. Invocación de awk).
- La variable `IGNORECASE` y sus efectos (Ver la sección Sensibilidad a Mayúsculas en el Matching).
- La interpretación de los nombres de ficheros `` `/dev/stdin' ``, `` `/dev/stdout' ``, `` `/dev/stderr' ``, y `` `/dev/fd/n' ``. (Ver la sección Streams de Entrada/Salida Estándard).
- La opción '-c' para desactivar estas extensiones (Ver la sección 14. Invocación de awk).
- Las opciones '-a' y '-e' para especificar la sintaxis de las expresiones regulares que gawk aceptará (Ver la sección 14. Invocación de awk).

16. Sumario de gawk

Este apéndice proporciona un sumario detallado de la línea de comando *gawk y el lenguaje awk*. Está diseñado para que sirva como "*referencia rápida*".

Sumario de Opciones de la Línea de comandos

La línea de comando consiste en las opciones de gawk, el texto del programa awk (si no se suministra a través de la opción '-f')

La línea de comando está formada por opciones del propio gawk, el texto del programa awk (si no se suministra a través de la opción `-f'), y los valores que se van a hacer disponibles en las variables awk predefinidas ARGC y ARGV:

```
awk [-Ffs] [-v var=val] [-V] [-C] [-c] [-a] [-e] [--] 'program' file ...
awk [-Ffs] -f source-file [-f source-file ...] [-v var=val] [-V] [-C] [-c] [-a] [-e] [--]
file ...
```

Las opciones que acepta gawk son:

-F*fs*

Utilice fs para especificar el separador de campos de entrada (el valor de la variable predefinida FS).

-f *program-file*

Lee el código fuente del programa awk del fichero *program-file*, en lugar de desde el primer argumento de la línea de comando.

-v *var=val*

Asigna a la variable *var* el valor *val* antes de comenzar la ejecución del programa.

-a

Especifica el uso de la sintasis tradicional de awk para la expresiones regulares. Esto significa que `\' puede ser utilizado para anular los operadores de expresiones regulares dentro de los corchetes, de la misma forma que pueden hacerlo fuera de ellos.

-e

Especifica el uso de la sintasis de egrep para la expresiones regulares. Esto significa que `\' no sirve como carácter de anulación dentro de los corchetes.

-c

Especifica modo compatibilidad, en el cual se desactivan las extensiones de gawk.

-V

Imprime la información de la versión para la copia en particular de gawk en la salida de error. Esta opción podría desaparecer en una futura versión de gawk.

-C

Imprime la versión corta de la Licencia Pública General en la salida de error. Esta opción podría desaparecer en una futura versión de gawk.

--

Señala el final de las opciones. Esto es útil para permitir que argumentos posteriores al propio programa awk comiencen con un `-'. Esto es principalmente para consistencia con la convención de parsing de argumentos POSIX.

Cualquier otra opción es marcada como inválida, pero son ignoradas de todas formas. Ver la sección 14. Invocación de awk, para más detalles.

Sumario del Lenguaje

Un programa awk consiste en una secuencia de sentencias acción-patrón y definiciones de funciones opcionales.

```
pattern    { action statements }
function name(parameter list)     { action statements }
```

gawk primero lee el código fuente del programa de los ficheros de programa si se especifican, o del primer argumento que no sea opción de la línea de comando. La opción `-f' podría ser usada múltiples veces en la línea de comando. gawk lee el texto del programa de todos los ficheros de programa especificados en la línea de comando, concatenándolos de forma efectiva en el orden en el que son especificados. Esto es útil para construir librerías de funciones awk, sin tener que incluirlas en cada programa awk nuevo que las utilice. Para utilizar una función de librería en un fichero desde un programa tecleado en la línea de comando, especifica `-f /dev/tty'; después teclea tu programa, y finalízalo con un **C-d**. Ver la sección 14. Invocación de awk.

La variable de entorno AWKPATH especifica un path de búsqueda para ser usado en la búsqueda de ficheros fuentes especificados con la opción `-f'. Si la variable AWKPATH no tiene valor, gawk utiliza el path por defecto, `.:/usr/lib/awk:/usr/local/lib/awk'. Si un nombre de fichero dado a la opción `-f' contiene un carácter `/', no se realiza busqueda en el path. Ver la sección La variable de entorno AWKPATH, para una descripción completa de la variable de entorno AWKPATH.

gawk compila el programa a una forma interna, y entonces procede a leer cada fichero nombrado en el array ARGV. Si no se nombran ficheros en la línea de comando, gawk lee la entrada estándar.

Si un "fichero" nombrado en la línea de comando tiene la forma `*var=val*', es tratado como una asignación de variable: a la variable *var* se le asigna el valor *val*.

Para cada línea en la entrada, gawk la comprueba para ver si encaja con cualquier *patrón* en el programa awk. Para cada patrón con el que encaja la línea, se ejecuta la acción asociada.

Variables y Campos

Las variables de awk son dinámicas; se definen cuando son usadas por primera vez. Sus valores son o números de punto flotante o cadenas. awk también tiene arrays de una dimensión; los arrays multidimensionales podrían ser simulados. Existen varias variables predefinidas que awk fija según se va ejecutando un programa; éstas se resumen a continuación.

Campos

Según gawk van leyendo las líneas de entrada, las divide en *campos*, usando el valor de la variable FS como el separador de campos. Si FS es un único carácter, los campos son separados por ese carácter. De otro modo, se espera que FS sea una expresión regular completa. En el caso especial de que FS es un único carácter en blanco, los campos son separados por medio de espacios en blanco y/o tabuladores. Señalar que el valor de IGNORECASE (Ver la sección Sensibilidad a Mayúsculas en el Matching) también afecta a cómo se parten los campos cuando FS es una expresión regular.

Cada campo en la línea de entrada podría ser referenciado por su posición, $1, $2, y así sucesivamente. $0 es la forma de referirse a la línea completa. Se le puede también asignar un valor a un determinado campo. Los números de campo no es necesario que sean constantes:

```
n = 5
print $n
```

imprime el quinto campo de la línea de entrada. La variable NF es fijada al número total de campos de la línea de entrada.

La referencia a un campo no existente (por ejemplo, campos superiores a $NF) devuelve la cadena nula. Sin embargo, la asignación a un campo no existente (e.g., $(NF+2) = 5) incrementa el valor de NF, creando cualquier campo intermedio con la cadena nula como su valor, y hace que el valor de $0 sea modificado, con los campos separados por el valor de OFS.

Ver la sección 3. Leyendo ficheros de entrada, para una descripción completa de la forma en la que awk define y utiliza los campos.

Variables Implícitas (Built-in)

Las variables implícitas de awk son:

ARGC

El número de argumentos de la línea de comandos (no incluyendo las opciones el propio programa awk).

ARGV

El array de argumentos de la línea de comandos. El array es indexado desde 0 a ARGC - 1. El cambio dinámico del contenido de ARGV puede controlar los ficheros utilizados para datos.

ENVIRON

Un array que contiene el valor de las variables de entorno. El array es indexado por el nombre de la variable, siendo cada elemento el valor de esa variable. Por lo que, la variable de entorno HOME se obtendría mediante ENVIRON["HOME"]. Su valor podría ser `/u/close'.

El cambio de este array no afecta al entorno visto por los programas a los que gawk llama a través de la redirección o la función `system`. (Esto podría cambiar en una versión futura de gawk.)

Algunos sistemas operativos no tienen variables de entorno. El array `ENVIRON` está vacío cuando se está ejecutando en estos sistemas.

FILENAME

El nombre del fichero de entrada actual. Si no se especifican ficheros en la línea de comando, el valor de `FILENAME` es `-'.

FNR

El número de registro de entrada en el fichero de entrada actual.

FS

El separador de campos de entrada, un blanco por defecto.

IGNORECASE

El indicador de sensibilidad a mayúsculas-minúsculas para las operaciones de expresiones regulares. Si `IGNORECASE` tiene un valor distinto de cero, el encaje de patrones en las reglas, el particionamiento de campos con `FS`, el encaje de expresiones regulares con `~' y `!~', y las funciones predefinidas `gsub`, `index`, `match`, `split` y `sub` todas ignoran la diferencia entre mayúsculas-minúsculas cuando realizan operaciones de expresiones regulares.

NF

El número de campos en el registro de entrada actual.

NR

El número total de registros de entrada leídos hasta el momento.

OFMT

El formato de salida para números, por defecto "%.6g".

OFS

El separador de campos de la salida, un blanco por defecto.

ORS

El separador de registros de la salida, un salto de línea (*newline*) por defecto.

RS

El separador de registros de entrada, por defecto un salto de línea o newline. `RS` es diferente, ya que solamente el primer carácter de su valor cadena se usa como separador de registros. Si se le da a `RS` la cadena nula, entonces los registros son separados por líneas en blanco. Si se le da a `RS` la cadena nula, entonces el carácter newline o salto de línea actúa siempre como separador de campos, en lugar de cualquier valor que `FS` pudiese tener.

RSTART

El índice del primer carácter que encaja con `match`; 0 si no existe ninguna concordancia o encaje.

RLENGTH

La longitud de la cadena que encaja con `match`; -1 si no existe ninguna concordancia o encaje.

SUBSEP

La cadena utilizada para separar subíndices múltiples en los elementos de un array, por defecto `"\034"`.

Ver la sección 13. Variables Implícitas (Built-in).

Arrays

Los arrays son subindexados con una expresión entre corchetes (`` `[' y `]' ``). La expresión podría ser o un número o una cadena. Debido a que los arrays son asociativos, los índices que son cadenas son más representativos y no son convertidos a números.

Si utilizas múltiples expresiones separadas por comas dentro de los corchetes, entonces el subíndice del array es una cadena que consta de la concatenación de valores de subíndices individuales, convertidos a cadenas, separados por el separador de subíndices (el valor de SUBSEP).

El operador especial `in` podría ser útil en una sentencia `if` o `while` para ver si un array tiene un índice formado por un determinado valor.

```
if (val in array)
        print array[val]
```

si el array tiene subíndices mútiples, utilice `(i, j, ...) in array` para chequear la existencia de un determinado elemento.

La construcción `in` podría ser también usada en un bucle `for` para hacer una iteración para todos los elementos de un array. Ver la sección Recorrido de todos los elementos de un Array.

Un elemento podría ser eliminado de un array utilizando la sentencia `delete`.

Ver la sección 10. Arrays en awk, para información más detallada.

Tipos de Datos

El valor de una expresión awk es siempre o un número o una cadena.

Ciertos contexto (tales como operadores aritméticos) requieren valores numéricos. Ellos convierten cadenas a números interpretando el texto de la cadena como un numérico. Si la cadena no tiene el formato de un numérico, la convierte a 0.

Ciertos contextos (tales como la concatenación) requiere valores cadena de texto. En estos contextos, los números se convierten a cadenas realizando una impresión efectiva de ellos.

Para forzar la conversión de un valor cadena de texto a numérico, simplemente súmale 0 a dicha cadena de texto. Si el valor inicial es ya un número, esta operación nunca altera su valor.

Para forzar la conversión de un valor numérico a cadena de texto, concaténale una cadena nula.

El lenguaje awk define las comparaciones como si fuesen numéricas si fuese posible, si no son posible o uno o los dos operandos se convierten a cadenas y se realiza una comparación de cadenas.

Las variables no inicializadas tienen el valor cadena "" (la cadena nula o vacía). En los contexto donde un número es requerido, esto es equivalente a 0.

Ver la sección Variables, para más información sobre los nombres de variables y su inicialización.; Ver la sección Conversiones de Cadenas y Números, para más información de cómo los valores de las variables son interpretados.

Patrones y Acciones

Un programa awk es principalmente compuesto de reglas, cada una consistente en un patrón seguido de una acción. La acción se presenta encerrada entre `{' y `}'. Se podría omitir tanto el patrón como la acción correspondiente, pero por supuesto que no se podrían omitir ambos. Si se omite el patrón, la acción se ejecuta para todas las líneas individuales de la entrada. La omisión de una acción es equivalente a esta acción,

```
{ print }
```

la cuál imprime la línea completa.

Los comentarios comienzan con el carácter `#', y continúa hasta el final de la línea. Las líneas en blanco podrían ser usadas para separar las sentencias. Normalmente, una sentencia finaliza con un carácter newline, sin embargo, este no es el caso para las líneas que finalizan con un `,', `{', `?', `:', `&&', o `||'. Las líneas que finalizan en do o else también hacen que la sentencias automáticamente continúen en la siguiente línea.

En otros casos, una línea podría ser continuada finalizandola con un `\', en cuyo caso el newline es ignorado.

Se podrían poner múltiples sentencias en una misma línea separándolas con un `;'. Esto se aplica a ambas partes, las sentencias dentro de la parte de acción de una regla (el caso normal), y a las propias sentencias de la regla.

Ver la sección Comentarios en Programas awk, para información sobre la convención para comentarios en awk; Ver la sección Sentencias frente a Líneas en awk, para una descripción del mecanismo para continuación de línea en awk.

Patrones

Los patrones awk podrían ser uno de los siguientes:

```
/expresión regular /
expresión relacional
patrón && patrón
patrón || patrón
patrón ? patrón : patrón
(patrón)
! patrón
patrón1, patrón2
BEGIN
END
```

BEGIN y END son dos tipos especiales de patrones que no son chequeados contra la entrada. Las partes de acción de todas las reglas BEGIN son fusionadas como si todas las sentencias hubiesen sido escritas en una única regla BEGIN. Son ejecutadas antes de ser leída ninguna entrada. De forma similar, todas las reglas END son fusionadas,

y ejecutadas cuando se ha acabado de leer toda la entrada (o cuando se ejecuta una sentencia `exit`). Los patrones `BEGIN` y `END` no pueden ser combinados con otros patrones en expresiones de patrón. Las reglas `BEGIN` y `END` no pueden tener las partes de acción omitidas.

Para los patrones `` `/expresión-regular/' ``, la sentencia asociada se ejecuta para cada línea de entrada que encaja con la expresión regular. Las expresiones regulares son las mismas que las utilizadas en `egrep`, y son resumidas más abajo.

Una expresión regular podría utilizar cualquiera de los operadores definidos más abajo en la sección de acciones. Estas generalmente chequean si ciertos campos encajan con ciertas expresiones regulares.

Los operadores `` `&&' ``, `` `||' ``, y `` `!' `` son el “y” lógico, “o” lógico y “no” lógico respectivamente como en C. Realizan también la evaluación corta de la expresión, también como en C, y se utilizan para combinar expresiones de patrón más primitivas. Como en la mayoría de los lenguajes, los paréntesis podrían ser usados para cambiar el orden de evaluación.

El operador `` `?:' `` es igual que su equivalente en C. Si es igual al primer patrón, entonces el segundo patrón es comparado contra el registro de entrada; si no, el registro de entrada se compara con el tercer patrón. Solamente se produce la comparación con uno de los dos patrones, el segundo o el tercero.

El `` `patrón1, patrón2' `` forma un patrón que es llamado un patrón de rango. Encaja con todas las líneas de entrada empezando en una línea que encaje con el patrón1, y continuando hasta una línea que encaje con el patrón2, ambas inclusive. Un patrón de rango no puede ser utilizado como un operando con ninguno de los operadores patrón.

Ver la sección 6. Patrones, para una descripción completa de la parte de patrón de las reglas awk.

Expresiones Regulares

Las expresiones regulares son un género extendido de `egrep`. Estas compuestas de caracteres como los siguientes:

c

encaja con el carácter *c* (asumiendo que *c* es un carácter sin significado especial en las expresiones regulares).

c

encaja con el carácter literal *c*.

.

encaja con cualquier carácter excepto con el newline.

^

encaja con el principio de una línea o una cadena.

$

encaja con el final de una línea o una cadena.

[*abc...*]

encaja con cualquiera de los carácteres *abc*... (clase de carácter).

[^*abc...*]

encaja con cualquier carácter excepto *abc*... y newline (la inversa de una clase de carácter).

`r1|r2`

encaja con *r1* o *r2* (alternativa).

`r1r2`

encaja con *r1*, y después *r2* (concatenación).

`r+`

encaja con una o más *r*'s.

`r*`

encaja con cero o más *r*'s.

`r?`

encaja con cero o una *r*.

`(r)`

encaja con *r* (agrupación).

Ver la sección Expresiones Regulares como Patrones, para una explicación más detallada de las expresiones regulares.

Las secuencias de escape permitidas en las constantes cadena son también válidas en las expresiones regulares. (Ver la sección Expresiones Constantes).

Acciones

Las sentencias de acción están encerradas entre llaves `{' y `}'. Las sentencias de la acción constan de las sentencias de asignación, condicionales, y bucles usuales en la mayoría de los lenguajes. Los operadores, sentencias de control, y sentencias de entrada/salida disponibles son similares a sus homólogas en C.

Operadores

Los operadores en awk, en orden de precedencia creciente son

`= += -= *= /= %= ^=`

Asignación. Ambas, la asignación absoluta (*variable=valor*) como la asignación con operador (las otras formas) son soportadas

`?:`

Una expresión condicional, como en C. Esta tiene la forma *`expr1 ? expr2 : expr3`*. Si *expr1* es verdadera, el valor de la expresión completa es *expr2*; si no el valor de la expresión completa es *expr3*. Solamente una de las dos (*expr2* y *expr3*) es evaluada.

`||`

“o” lógico.

&&

“y” lógico.

~ !~

Concordancia con expresión regular, no concordancia con expresión regular.

< <= > >= != ==

Los operadores relacionales usuales.

blank

Concatenación de cadenas.

+ -

Suma y resta.

*** / %**

Multiplicación, división y módulo.

+ - !

Operadores unarios suma y resta y la negación lógica.

^

Exponenciación (`**' podría también ser usado, y `**=' para el operador asignación).

++ --

Incremento y decremento, ambos prefijo y sufijo.

$

Referencia a un campo.

Ver la sección 8. Acciones: Expresiones, para una descripción completa de todos los operadores listados más arriba.

Ver la sección Examinando campos, para una descripción del operador de referencia de campos.

Sentencias de Control

Las sentencias de control son las siguientes:

```
if (condición) sentencia [ else sentencia ]
while (condición) sentencia
do sentencia while (condición)
for (expr1; expr2; expr3) sentencia
for (variable in array) sentencia
break
continue
```

`delete` *`array`*`[`*`índice`*`]`
`exit [` *`expresión`* `]`
`{` *`sentencias`* `}`

Ver la sección 9. Acciones: Sentencias de Control, para una descripción completa de todas las sentencias de control listadas arriba.

Sentencias de Entrada/Salida

Las sentencias de entrada/salida son las siguientes:

`getline`

Fija el valor de `$0` a partir del siguiente registro de entrada; fija además los valores de las variables `NF`, `NR`, `FNR`.

`getline <`*`fichero`*

Fija el valor de `$0` a partir del siguiente registro de *fichero*; fija el valor de `NF`.

`getline` ***`var`***

Fija el valor de *var* a partir del siguiente registro de entrada; fija además los valores de las variables `NF`, `FNR`.

`getline` ***`var`*** **`<`*`fichero`***

Fija el valor de *var* a partir del siguiente registro de *fichero*.

`next`

Detiene el procesamiento del registro actual. El siguiente registro de entrada se lee y el procesamiento comienza en el primer patrón en el programa `awk`. Si se alcanza el final de los datos de entrada, la regla/s `END` si existiese/n son ejecutadas.

`print`

Imprime el registro actual.

`print` ***`lista-expresiones`***

Imprime expresiones.

`print` ***`lista-expresiones`*** **`>`** ***`fichero`***

Imprime expresiones sobre *fichero*.

`printf` ***`format, lista-expresiones`***

Formatea e imprime la lista de expresiones.

`printf` ***`format, lista-expresiones`*** **`> fichero`**

Formatea e imprime sobre *fichero*.

También se permiten otras redirecciones de la entrada/salida. Para `print` y `printf`, `` `>> `` *`file`*`'` añade la salida a *fichero*, mientras `` `| `` *`command`*`'` escribe en un pipe. De una forma similar `` ``*`command`*` | getline' `` hace un pipe de la entrada en `getline`. `getline` devuelve un 0 al final del fichero, y –1 en caso de error.

Ver la sección Entrada explícita con getline, para una descripción completa de la sentencia `getline`. Ver la sección 4. Imprimiendo la salida, para una descripción completa de `print` y `printf`. Finalmente, Ver la sección La Sentencia next, para una descripción de como funciona la sentencia `next`.

Sumario printf

La sentencia de `awk printf` y la función `sprintf` acepta los siguientes formatos de especificación de conversión:

%c

Un carácter ASCII. Si el argumento usado para `%c' es numérico, es tratado como un carácter e impreso. Si no, se asume que el argumento es una cadena, y solamente se imprime el primer carácter de esa cadena.

%d

Un número decinal (la parte entera).

%i

También un entero decimal.

%e

Un número en punto flotante de la forma `[-]d.dddddd[+-]dd'.

%f

Un número punto flotante de la forma [-]ddd.dddddd.

%g

Utilice la conversión `%e' o `%f', cualquiera que sea más corta, con los ceros no significativos suprimidos.

%o

Un número octal sin signo (de nuevo, un entero).

%s

Una cadena de caracteres.

%x

Un número hexadecimal sin signo (un entero).

%X

Como `%x', excepto el uso de `A' hasta `F' en lugar de `a' hasta `f' para el decimal 10 hasta el 15.

%%

Un simple carácter `%'; no se convierte ningún argumento.

Existen parámetros adicionales y opcionales que podrían ir entre el `%' y la letra de control:

-

La expresión debería ser justificada a la izquierda dentro de su campo.

ancho

El campo debería ser relleno hasta este *ancho*. Si ancho tiene un cero inicial, entonces el campo es rellano con ceros, si no es relleno con espacios en blanco.

.precisión

Un número que indica el ancho máximo de las cadenas o los dígitos a la derecha del punto decimal.

Ver la sección Uso de sentencias printf para una impresión más elegante, para ejemplos y para una descripción más detallada.

Nombres de ficheros especiales

Cuando se realiza la redirección de I/O desde `print` o `printf` a un fichero, o a través de `getline` desde un fichero, gawk reconoce ciertos nombres de ficheros especiales internamente. Estos nombres de ficheros permiten el acceso a descriptores de ficheros abiertos heredados del proceso padre de gawk (normalmente la shell). Los nombres de fichero son:

`/dev/stdin'

La entrada estándar.

`/dev/stdout'

La salida estándar.

`/dev/stderr'

La salida de error estándar.

`/dev/fd/*n*'

El fichero determinado por el descriptor de fichero abierto *n*.

Estos nombres de ficheros podrían también ser usados en la línea de comando para nombrar ficheros de datos.

Ver la sección Streams de Entrada/Salida Estándard, para una descripción más detallada que proporciona el motivo de esta característica.

Funciones Numéricas

awk presenta las siguientes funciones aritméticas predefinidas:

atan2(*y*, *x*)

devuelve el arcotangente de *y/x* en radianes.

cos(*expr*)

devuelve el coseno en radianes.

exp(*expr*)

la función exponencial.

int(*expr*)

trunca a entero.

log(*expr*)

la función logaritmo natural.

rand()

devuelve un número aleatorio entre 0 y 1.

sin(*expr*)

devuelve el seno en radianes.

sqrt(*expr*)

la función raíz cuadrada.

srand(*expr*)

utiliza *expr* como una nueva semilla para el generador de números aleatorios. Si no se le proporciona *expr*, se usa el momento del día. El valor devuelto es la semilla anterior del generador de números aleatorios.

Funciones de Cadenas

awk tiene las siguientes funciones de cadena predefinidas:

gsub(*r*, *s*, *t*)

para cada subcadena que encaja con la expresión regular *r* en la cadena *t*, la sustituye por la cadena *s*, y devuelve el número de sustituciones. Si no se le suministra el valor de *t* utiliza la línea de entrada completa (`$0`).

index(*s*, *t*)

devuelve el índice de la cadena *t* dentro de la cadena *s*, o `$0` si *t* no está presente.

length(*s*)

devuelve la longitud de la cadena *s*.

match(*s*, *r*)

devuelve la posición dentro de la cadena *s* donde ocurre la expresión regular *r*, o 0 si *r* no está presente, y fija los valores de las variables `RSTART` y `RLENGTH`.

split(*s*, *a*, *r*)

parte la cadena *s* en el array *a* en función del separador determinado por la expresión regular *r*, y devuelve el número de campos. Si se omite *r*, en su lugar se utiliza FS.

sprintf(*fmt, lista-expresiones*)

imprime *lista-expresiones* de acuerdo al formato especificado por *fmt*, y devuelve la cadena resultante.

sub(*r, s, t*)

actúa igual que gsub, pero solamente la primera subcadena que encaje es sustituida.

substr(*s, i, n*)

devuelve la subcadena de *n* caracteres de longitud de *s* comenzando en la posición *i*. Si se omite *n*, devuelve el resto de la cadena *s* a partir de la posición *i*.

tolower(*str*)

devuelve una copia de *str*, con todos los caracteres en mayúsculas de *str* convertidos a su correspondiente carácter en minúsculas. Los caracteres que no son alfabéticos permanecen invariables.

toupper(*str*)

devuelve una copia de *str*, con todos los caracteres en minúsculas de *str* convertidos a su correspondiente carácter en mayúsculas. Los caracteres que no son alfabéticos permanecen invariables.

system(*linea-comando*)

Ejecuta el comando *línea-comando*, y devuelve el estado de salida.

Ver la sección 11. Funciones Implícitas (Built-in), para una descripción de todas las funciones built-in de awk.

Constantes de Cadenas

Las cadenas constantes en awk son secuencias de caracteres encerradas entre dobles comillas ("). Dentro de las cadenas, se reconocen ciertas *secuencias de escape*, como en C. Estas son:

****

Una barra invertida literal.

\a

El carácter “alerta”, normalmente el carácter ASCII BEL.

\b

Backspace o retroceso.

\f

Formfeed.

\n

Nueva línea.

\r

Retorno de Carro.

\t

Tabulador Horizontal.

\v

Tabulador Vertical.

\xhex digits

El carácter representado por la cadena de dígitos hexadecimales que siguen a la `\x'. Como en ANSI C, todos los dígitos hexadecimales siguientes son considerados como parte de la secuencia de escape. E.g., "\x1B" es una cadena que contiene el carácter ASCII ESC (escape).

\ddd

El carácter representado por la secuencia de 1-, 2- o 3- dígitos de dígitos octales.dígitos. Por lo que, "\033" es también una cadena que contiene el carácter ASCII ESC (escape).

\c

El carácter literal *c*.

Las secuencias de escape podrían ser también utilizadas dentro de las expresiones regulares constantes (e.g., la expresión regular /[\t\f\n\r\v]/ encaja con caracteres de espacio en blanco).

Ver la sección Expresiones Constantes.

Funciones

Las funciones en awk se definen de la siguiente manera:

`function` *`name`*`(`*`parameter list`*`) { `*`statements`*` }`

Los parámetros reales suministrados en la llamada a la función son utilizados para instanciar los parámetros formales declarados en la función.

Los arrays son pasados por referencia, otras variables son pasadas por valor.

Si hubiesen menos parámetros en la llamada a la función que nombres existiesen en la *lista de parámetros*, los nombres extras reciben la cadena nula como valor. Los nombres extras tienen el efecto de variables locales.

El paréntesis de apertura en una llamada a función debe seguir inmediatamente al nombre de la función, sin ningún espacio en blanco en medio. Esto es para evitar una ambiguidad sintáctica con el operador concatenación.

La palabra `func` podría ser usada en lugar de `function`.

Ver la sección 12. Funciones definidas por el Usuario, para una descripción más completa.

17. Programas Ejemplo

Caso Práctico 1

El siguiente ejemplo es un programa awk completo, el cual imprime el número de ocurrencias de cada palabra en su entrada. Ilustra la naturaleza asociativa de los arrays de awk usando cadenas como subíndices. También demuestra la construcción `for x in array'. Finalmente, muestra como se puede usar awk en conjunción con otros programas de utilidades para realizar una tarea útil de relativa complejidad con un esfuerzo mínimo. Algunas explicaciones aparecen a continuación del listado del programa.

```
awk '
# Imprime lista de frecuencias de palabras
{
    for (i = 1; i <= NF; i++)
        freq[$i]++
}

END {
    for (word in freq)
        printf "%s\t%d\n", word, freq[word]
}'
```

La primera cosa a resaltar de este programa es que tiene dos reglas. La primera regla, debido a que tiene un patrón vacío, se ejecuta para cada línea de la entrada. Utiliza el mecanismo de acceso a los campos de awk (Ver la sección Examinando campos) para extraer las palabras individuales de la línea, y la variable implícita NF (Ver la sección 13. Variables Implícitas (Built-In)) para conocer cuantos campos están disponibles.

Para cada palabra de entrada, un elementos del array freq se incrementa para reflejar que dicha palabra ha aparecido otra vez en el fichero de entrada.

La segunda regla, debido a que tiene el patrón END, no se ejecuta hasta que se ha acabado de leer la entrada. Imprime el contenido del array freq que ha sido construido dentro de la primera acción.

Advierta que este programa presenta distintos problemas al aplicarlos sobre ficheros de texto reales, ya que no tiene en cuenta los signos de puntuación.

El lenguaje awk considera los caracteres en mayúsculas y minúsculas como distintos. Por lo tanto, 'foo' y 'Foo' no son tratados por este programa como la misma palabra. Esto es indeseable ya que en el texto normal, las palabras comienzan por mayúsculas si son comienzo de sentencias, y un analizador de frecuencias no debería ser sensible a eso.

La salida no aparece en un orden útil. Podrías estar interesado en saber qué palabras aparecen más frecuentemente, o sacar las palabras y sus respectivas frecuencias en orden alfabético.

La forma de resolver estos problemas es usar otras utilidades de sistema para procesar la entrada y salida del script de awk. Supón que el script mostrado más arriba se salva en el fichero 'frequency.awk'. Entonces el comando de shell:

```
tr A-Z a-z < file1 | tr -cd 'a-z\012' \
  | awk -f frequency.awk \
  | sort +1 -nr
```

produce un tabla de las palabras que aparecen en el fichero '`file1`' en orden de frecuencia de aparición decreciente.

El primer comando `tr` en este pipeline transforma todas las letras en mayúsculas del fichero '`fichero1`' a minúsculas. El segundo comando `tr` elimina todos los caracteres de la entrada excepto los caracteres en minúsculas y los caracteres newline. El segundo argumento del segundo `tr` esta entre comillas para proteger a la barra invertida de que sea interpretada por la shell. El programa `awk` lee los datos de entradas con las transformaciones realizadas previamente por la ejecución de los comandos `tr` y produce una tabla de frecuencia de palabras, la cual no está ordenada.

La salida del script `awk` es ahora ordenada por el comando `sort` e impresa en el terminal. Las opciones suministradas a `sort` en este ejemplo especifican la ordenación por el segundo campo de cada línea de entrada (saltándose un campo), que las claves de ordenación sean tratadas como cantidades numéricas (de otra forma `'15'` vendría antes de `'5'`), y que la ordenación se debería hacer en orden descendente (inverso).

Caso Práctico 2

Supón que, en un sistema DataWarehouse, tienes que tratar un fichero de texto gigante (de varios cientos de Megabytes) y te gustaría conocer los valores que te envían para varios de sus campos que hacen referencia a la clave primaria de otras tablas.

El siguiente ejemplo es un programa de Shell Script (Korn Shell de Unix) en el que se utiliza awk para realizar lo que sería una agrupacion en SQL, para saber los valores que presenta un determinado campo, el número de registros que tiene cada valor y su porcentaje con respecto al total de registros.

Las consideraciones para este ejemplo son las siguientes:

- Se va a tratar un fichero de texto plano en el cual cada línea representa un registro. Este fichero plano se llamara "**tarjetas.txt**"
- Los distintos campos de cada registro del fichero estarán separados por el carácter "~" (alt+126). Destrás del último campo no irá el carácter "~"
- Cada línea del fichero presentará los siguientes campos: código de la tarjeta, matrícula, fecha de alta de la tarjeta, fecha de baja de la tarjeta, codigo de bloqueo, marca de la tarjeta, cuenta del cliente
- Se trata de hacer un estudio de los distintos códigos de bloqueo y las distintas marcas de tarjeta existentes.

Una muestra de los registros que presenta este fichero sería la siguiente:

```
7083211086652975~SIN MATRIC~1999-07-15~9999-12-31~z~008~1086658
704314580149969~M-9870-ID~1998-04-06~9999-12-31~z~005~145806
7078831752339974~M-4921-OV~1997-06-26~1999-04-26~z~001~175239
7078830306279991~M-4730-OH~1994-02-08~1996-03-12~z~001~30633
7078830977159967~553745C~1996-07-03~1998-08-05~z~001~97721
```

Aquí tienes el programa:

```
#!/bin/ksh
#
#############################################################################
# El formato del fichero o paquete a procesar sera el siguiente:
# campo1~campo2~campo3~campo4~campo5
# !!! DETRAS DEL ULTIMO CAMPO NO IRA EL SEPARADOR !!!
#
# Para quitarle los gusanillos del final de linea si los tuviera habria que
# ejecutar lo siguiente:
#
# sed "1,$ s/\~$//g" fichero_a_tratar > fichero_tratado
#
#############################################################################
clear
nawk -F~ ' {
num_total_reg++;
for (i=1;i<=NF;i++) {
# Posibilidad 1: Agrupar y obtener los distintos valores para
# todos y cada uno de los campos
# v_array[i,$i]++;
# Posibilidad 2 (la del ejemplo): Que solo nos saque la agrupacion de unos
# determinados campos. Codigo de bloqueo de la tarjeta
# (campo 5) y marca de la tarjeta (campo 6)
if (i==5 ¦¦ i==6) v_array[i,$i]++;.}
}
END {
for (i in v_array) {
split(i,array_sep,SUBSEP);
print
array_sep[1],"~",array_sep[2],"~",v_array[i],"~",(v_array[i]/num_total_reg)*100,"~";
}
} ' tarjetas.txt > /tmp/kk.txt
sort -t~ +0n +2nr /tmp/kk.txt > /tmp/kk2.txt
if [ $? -ne 0 ]
then
echo "Error al ordenar el fichero /tmp/kk.txt"
exit 9
fi
rm /tmp/kk.txt
nawk -F~ '
BEGIN {
num_campo = 0;
num_reg_total = 0;
valor_anterior = "??????";
print " Nombre Campo Valor Num. Regs % ";
```

```
print "----------------- ----------------------------- ---------- ---------";
}
{
# Vamos a controlar cuando se cambia de campo para hacer un salto de linea
if (valor_anterior!=$1) {
num_campo++;
valor_anterior=$1;
print "";
}
# Calculo del numero de registro totales del fichero tratado
if (num_campo==1) { num_reg_total += $3; }
# Asociamos el numero de campo con su DESCRIPCION
if ($1==5) $1="cod_bloqueo_trj";
if ($1==6) $1="marca_tarjeta";
# Formateamos el porcentaje para alinearlo por la derecha
porcentaje = sprintf("%3.4f%%",$4);
printf "%18s %30s %10s %9s\n",$1,$2,$3,porcentaje;
}
END {
print "";
print " Numero total de registros procesados: ",num_reg_total;
} ' /tmp/kk2.txt > /tmp/out.txt
cat /tmp/out.txt
rm /tmp/kk2.txt
rm /tmp/out.txt
```

La salida que generaría la ejecución de este Shell Script es la siguiente:

```
Nombre Campo                   Valor                Num. Regs    %
------------------ ------------------------------ ---------- ---------
cod_bloqueo_trj                                 z        8209   96.2481%
cod_bloqueo_trj                                41         138    1.6180%
cod_bloqueo_trj                                35          76    0.8911%
cod_bloqueo_trj                                17          51    0.5980%
cod_bloqueo_trj                                43          31    0.3635%
cod_bloqueo_trj                                37          24    0.2814%

marca_tarjeta                                 008        5924   69.4571%
marca_tarjeta                                 001        2115   24.7977%
marca_tarjeta                                 007         215    2.5208%
marca_tarjeta                                 002         150    1.7587%
marca_tarjeta                                 005         116    1.3601%
marca_tarjeta                                 004           9    0.1055%
```

Numero total de registros procesados: 8529

18. Notas de Implementación

Este apéndice contiene información de interés principalmente para implementadores y mantenedores de gawk. Todo lo que aparece aquí se aplica especificamente a gawk, y no a otras implementaciones.

Compatibilidad hacia atrás y Depuración

Ver la sección Extensiones en gawk que no están en S5R4, para ver un sumario de las extensiones GNU al lenguaje y programa awk. Todas estas características pueden ser desactivadas o compilando gawk con ‘-DSTRICT’ (no se recomienda), o invocando gawk con la opción ‘-c’.

Si se compila gawk para depuración con ‘-DDEBUG’ , entonces existen dos opciones más disponibles desde la línea de comando:

`-d'	Imprime información de depuración durante la ejecución.
`-D'	Imprime la información de la pila del analizador de sentencias según se va analizando el programa.

Ambas opciones están pensadas solamente para desarrolladores de gawk serios, y no para usuarios ocasionales. Ellas no han sido compiladas probablemente con tu versión de gawk, ya que conllevan una ejecución lenta.

El código para reconocer nombres de ficheros especiales tal y como ‘/dev/stdin’ puede ser deshabilitado en tiempo de compilación con `-DNO_DEV_FD', o con `-DSTRICT'.

Futuras Extensiones Posibles

Esta sección lista exhaustivamente las extensiones que indican las direcciones que estamos considerando actualmente para gawk.

Un printf compatible con el ANSI C

Las funciones printf y sprintf podrían ser mejoradas para hacerlas totalmente compatibles con la especificación para la familia de funciones printf de ANSI C.

RS como una regexp

El significado de RS podría ser generalizado a través de las líneas de FS.

Control del entorno de subproceso

Los cambios hechos en gawk al array ENVIRON podría ser propagado a subprocesos ejecutados por gawk.

Bases de Datos

Podría ser posible mapear un fichero NDBM/GDBM a un array awk.

Campos de un único carácter

La cadena nula, “”, como separador de campo, causará la partición en campos y la función split para separar caracteres individuales. Por lo que, `split(a, "abcd", "")` produciría `a[1] == "a"`, `a[2] == "b"`, y así sucesivamente.

Registros y campos de longitud fija

Se podría proporcionar un mecanismo para permitir la especificación de registros y campos de longitud fija.

Sintáxis de Expreg

La sintaxis de `egrep` para expresiones regulares, que se especifica ahora mismo con la opción `‘-e’`, podría llegar a ser el comportamiento por defecto, ya que el estandárd POSIX podría especificar esto.

19. Glosario

Acción

Una serie de sentencias de awk asociadas a una regla. Si el patrón de la regla encaja con el registro de entrada, el lenguaje awk ejecuta la acción de la regla. Las acciones son siempre encerradas entre llaves. Ver la sección 7. Acciones: Overview.

Asignación

Una expresión de awk que cambia el valor de algún objeto de datos o variable de awk. Un objeto al que le puedes asignar un valor se conoce como valori. Ver la sección Expresiones de Asignación.

C

El lenguaje de programación de sistema en el cual está escrita la mayor parte del software GNU. El lenguaje de programación awk tiene una sintasis similar a la de C, y este manual hace referencia a ello en los distintos apartados donde tiene lugar dicha similitud entre lenguajes.

Cadena

Un dato consistente en una secuencia de caracteres, como por ejemplo “Soy una cadena”. Las cadenas constantes se escriben con comillas dobles en el lenguaje awk, y podrían contener *secuencias de* escape. Ver la sección Expresiones Constantes.

Campo

Cuando awk lee un registro de entrada, parte el registro en piezas separadas por un espacio en blanco (o por una expresión regular separadora que puedes cambiar fijando el valor de la variable implícita FS). Tales piezas son llamadas campos. Ver la sección Cómo se particiona la Entrada en Registros.

Clave (Keyword)

En el lenguaje awk, una clave es una palabra que tiene un significado especial. Las claves están reservadas y no pueden ser utilizadas como nombres de variables.

Las claves de awk son: if, else, while, do...while, for, for...in, break, continue, delete, next, function, func, y exit.

Concatenación

La concatenación de dos cadenas significa pegarlas juntas, una detrás de la otra, resultando una nueva cadena. Por ejemplo, la cadena ‘loco’ concatenada con la cadena ‘motora’ tiene como resultado la cadena concatenada ‘locomotora’. Ver la sección Concatenación de Cadenas.

Editor de Stream

Un programa que lee registros de un stream de entrada y los procesa uno o más a la vez. Esto contrasta con los programas batch, los cuales podrían leer el fichero de entrada completamente antes de empezar a hacer nada, y con programas interactivos, los cuales requieren una entrada de un usuario.

Efecto Lateral

Un efecto lateral ocurre cuando una expresión tiene un efecto que va más alla de producir simplemente un valor. Las expresiones de asignación, expresión de incremento y llamadas a funciones tienen efectos laterales. Ver la sección Expresiones de Asignación.

Espacio en blanco

Una secuencia de caracteres tabulador o espacios en blanco que aparecen dentro de un registro de entrada o cadena.

Ensamblador awk Entretenido

Henry Spencer de la Universidad de Toronto escribió un ensamblador retargetable completamente mediante scripts de awk. Sus miles de líneas incluyen descripciones de máquinas para varias microcomputadoras de 9-bits. Se distribuye con gawk y es un buen ejemplo de un programa que hubiese estado mejor escrito en otro lenguaje.

Expresión Condicional

Una expresión que usa el operador ternario `?:', tal como *expr1* ? *expr2* : *expr3*. Se evalua la expresión *expr1*; si el resultado es cierto, el valor de la expresión completa es el valor de *expr2* sino el valor de la expresión completa es el valor de *expr3*. En cualquier caso, solo una de las dos exprsiones *expr2* o *expr3* es evaluada. Ver la sección Expresiones Condicionales.

Expresión de Comparación

Una relación que es cierta o es falsa, tal y como (a < b). Las expresiones de comparación son usadas en sentencias if y while, y en patrones para seleccionar que registros de entrada se procesan. Ver la sección Expresiones de Comparación.

ExpReg

Abreviatura para *expresión regular*. Una expreg es un patrón que denota un conjunto de cadenas, posiblemente un conjunto infinito. Por ejemplo, la expreg `R.*xp' encaja con cualquier cadena que empiece con la letra R y acabe con las letras ‘xp’. En *awk*, las expreg son usadas en patrones y expresiones condicionales. Expreg podrían contener secuencias escape. Ver la sección Expresiones Regulares como Patrones.

Expresión Regular

Ver “ExpReg”.

Expresión Regular Constante

Una expresión regular constante es una expresión regular escrita entre barras, tal como `/foo/'. Esta expresión regular se fija cuando escribes tu programa awk, y no puede ser cambiada durante su ejecución. Ver la sección Cómo usar Expresiones Regulares.

Expresiones Regulares Dinámicas

Una expresión regular dinámica es una expresión regular escrita como una expresión ordinaria. Podría ser una cadena constante, tal como “loco”, pero podría ser también una expresión cuyo valor podría variar. Ver la sección Cómo usar Expresiones Regulares.

Fichero Especial

Un nombre de fichero interpretado internamente por gawk, en lugar de ser manejado directamente por el sistema operativo subyacente. Por ejemplo, '/dev/stdin'. Ver la sección Streams de Entrada/Salida Estándard.

Formato

Las cadenas de formato se usan para controlar la apariencia de la salida en la sentencia printf. La conversión de número a cadenas también es controlada por la cadena de formato contenida en la variable implícita OFMT. Ver la sección Letras para el control de formato; Ver también la sección Separadores de la Salida.

Función

Un grupo de sentencias especializadas usadas a menudo para encapsular tareas generales o específicas de un programa. awk tiene un número de funciones implícitas, y también te permite definir tus propias funciones. Ver la sección 11. Funciones Implícitas (Built-In); Ver también la sección 12. Funciones definidas por el Usuario.

Función Implícita (Built-in)

El lenguaje awk proporciona funciones implícitas (built-in) que realizan distintas operaciones sobre números y cadenas. Ejemplos son sqrt (para obtener la raíz cuadrada de un número) y substr (para obtener una subcadena de un cadena). Ver la sección 11. Funciones Implícitas (Built-In).

gawk

La implementación de GNU de awk.

Lenguaje awk

El lenguaje en el cual están escritos los programas awk.

Llaves (Curly Braces)

Estos son los caracteres `{' y `}'. Las llaves se usan en awk para delimitar acciones, sentencias compuestas y cuerpos de funciones.

Número

Un objeto de dato que tiene valor numérico. La implementación gawk utiliza formato en punto flotante de doble precisión para representar los números.

Objetos Dato

Estos son números y cadenas de caracteres. Los números se convierten en cadenas y viceversa, según sea necesario. Ver la sección Conversiones de Cadenas y Números.

Patrón

Los patrones le dicen a awk qué registros de entrada están sometidos a qué reglas.

Un patrón es un expresión condicional arbitraria contra la que se chequea la entrada. Si la condición es satisfecha, se dice que el patrón encaja (match) con el registro de entrada. Un patrón típico podría comparar el registro de entrada contra una expresión regular. Ver la sección 6. Patrones.

Programa awk

Un programa awk consiste en una serie de *patrones y acciones*, que conjuntamente reciben el nombre de *reglas*. Para cada registro de entrada pasado al programa, las reglas del programa son procesadas todas por turno. Los programas awk podrían también contener definiciones de funciones.

Rango (de líneas de entrada)

Una secuencia de líneas consecutivas del fichero de entrada. Un patrón puede especificar rangos de líneas de entrada para que sean procesados por awk, o puede especificar líneas simples. Ver la sección 6. Patrones.

Recursión

Cuando una función se llama a si misma, directa o indirectamente.

Redirección

La redirección significa realizar la entrada desde otro sitio distinto al stream de entrada estándar, o la salida a otro sitio distinto del stream de salida estándar.

Puedes redireccionar la salida de las sentencias *print* y *printf* a un fichero o un comando del sistema, usando los operadores `>', `>>', y `|'. Ver la sección Redireccionando la Salida de print y printf.

Registro de Entrada

El pedazo de información simple leído por awk. Normalmente, un registro de entrada de awk consiste en una línea de texto. Ver la sección Cómo se particiona la Entrada en Registros.

Regla

Un segmento de un programa awk, que especifica como procesar registros de entradas. Una regla consiste en un patrón y una acción. Awk lee un registro de entrada; entonces, para cada regla, si el registro de entrada satisface el patrón de la regla, awk ejecuta la acción de la regla. Si no, la regla no realiza nada para ese registro de entrada.

Script de awk

Otro nombre para un programa awk

Sentencia Compuesta

Una serie de sentencias awk, encerradas entre llaves ({}). Las sentencias compuestas pueden presentarse anidadas. Ver la sección 9. Acciones: Sentencias de Control.

Secuencias de Escape

Una secuencia especial de caracteres usada para describir caracteres no imprimibles, tales como \n' para newline (salto de línea), o `\033' para el carácter ASCII ESC (escape). Ver la sección Expresiones Constantes.

Valor-izquierdo (Lvalue, valori)

Una expresión que puede aparecer en el lado izquierdo de un operador de asignación. En la mayoría de los lenguajes, pueden ser variables o elementos de array. En awk, un designador de campo puede ser también usado como valor-izquierdo.

Variable Implícita (Built-in)

Las variables `ARGC`, `ARGV`, `ENVIRON`, `FILENAME`, `FNR`, `FS`, `NF`, `IGNORECASE`, `NR`, `OFMT`, `OFS`, `ORS`, `RLENGTH`, `RSTART`, `RS`, y `SUBSEP`, tienen un significado especial para `awk`. Cambiar los valores de algunas de ellas afecta al entorno de ejecución de `awk`. Ver la sección 11. Funciones Implícitas (Built-In).

20. Apéndice

A – Ficheros de Datos para los Ejemplos

Muchos de los ejemplos de este manual toman su entrada desde ficheros de datos de ejemplo. El fichero, llamado 'Lista-BBS', presenta una lista de sistemas de tablones de anuncio por ordenador con información sobre dichos sistemas. El segundo fichero de datos, llamado 'inventario-enviado', contiene información sobre embarques realizados en un mes. En ambos ficheros, cada línea se considera que es un "registro".

En el fichero 'Lista-BBS', cada registro contiene el nombre de un tablón de anuncios de ordenador, su número de teléfono, la tasa(s) de baudios del boletín, y un código para el número de horas que está operativo. Una 'A' en la última columna significa que el tablón opera 24 horas al día. Una 'B' en la última columna significa que el tablón opera por las tardes y durante el fin de semana solamente. Una 'C' significa que el tablón opera solamente en fin de semanas .

```
aardvark     555-5553     1200/300          B
alpo-net     555-3412     2400/1200/300     A
barfly       555-7685     1200/300          A
bites        555-1675     2400/1200/300     A
camelot      555-0542     300               C
core         555-2912     1200/300          C
fooey        555-1234     2400/1200/300     B
foot         555-6699     1200/300          B
macfoo       555-6480     1200/300          A
sdace        555-3430     2400/1200/300     A
sabafoo      555-2127     1200/300          C
```

El segundo fichero de datos, llamado 'inventario-enviado', representa información sobre embarques durante el año. Cada registro contiene el mes del año, el número de canastas verdes embarcadas, el número de cajas rojas embarcadas, el número de bolsas de naranjas embarcadas y el número de paquetes azules embarcados respectivamente. Existen 16 entradas, que cubren los doce meses de un año y cuatro meses del año siguiente.

```
Jan  13  25  15 115
Feb  15  32  24 226
Mar  15  24  34 228
Apr  31  52  63 420
May  16  34  29 208
Jun  31  42  75 492
Jul  24  34  67 436
Aug  15  34  47 316
Sep  13  55  37 277
Oct  29  54  68 525
Nov  20  87  82 577
Dec  17  35  61 401
```

```
Jan  21  36  64 620
Feb  26  58  80 652
Mar  24  75  70 495
Apr  21  70  74 514
```