BLOQUE II: PROGRAMACIÓN DE SISTEMAS
LENGUAJE AWK

Bibliografía

Tutorial de awk

http://www.grymoire.com/Unix/Awk.html

The GNU Awk User's Guide

http://www.gnu.org/software/gawk/manual

Manual de AWK por Jesús Alberto Vidal Cortés

http://www.lawebdelprogramador.com/cursos/AWK/5752-Guia_del_usuario_para_AWK.html

Dougherty D. and Robbins A. sed & awk. Second Edition. O'Reilly Media, 1997

1.INTRODUCCIÓN.

AWK es un lenguaje de programación diseñado para procesar datos basados en texto, ya sean ficheros o flujos de datos. El nombre AWK deriva de las iniciales de los apellidos de sus autores: Alfred **A**ho, Peter **W**einberger, y Brian **K**ernighan. La versión original de AWK fue escrita en 1977 en los Laboratorios de AT&T. En 1985 una nueva versión hizo a este lenguaje de programación más potente, introduciendo funciones definidas por el usuario, múltiples *flujos* de entrada y evaluación de expresiones regulares. Esta nueva versión estuvo disponible de forma general con el Unix System V versión 3.1. La versión 4 de System V añadió algunas características nuevas y también corrigió algunos de los agujeros que presentaba el lenguaje.

La implementación GNU de AWK, **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.

AWK es un lenguaje compacto con una sintaxis muy parecida al C, lo que lo hace muy sencillo de aprender para aquellos con conocimientos en C. Entre sus múltiples utilidades, se pueden destacar las siguientes: generación de informes, manejo de Bases de Datos pequeñas, traducción de formatos de fichero y la facilidad para la realización de operaciones matemáticas en ficheros numéricos.

¿Por qué es tan importante AWK? Por ser una excelente herramienta para informes y filtro de información. Muchas utilidades de UNIX proporcionan información con formato de filas y columnas. AWK es de gran ayuda en el procesamiento de filas y columnas, y es más fácil de usar que la mayoría de los lenguajes de programación convencionales. Puede ser considerado como un intérprete de pseudo-C; dispone, entre otros elementos, de los mismos operadores aritméticos de C, de funciones de

manipulación de cadenas y de matrices asociativas (indexadas por cadenas). Las matrices asociativas pueden convertir un problema complejo en un ejercicio trivial.

El lenguaje AWK se usará cuando no haya una solución más sencilla al problema a resolver. Teniendo en cuenta que AWK puede llevar a cabo muchas de las tareas/funcionalidades que presentan otros comandos, como por ejemplo:

- *grep* para buscar texto,
- *head/tail* para extraer parte de un texto,
- *cat, paste, join* para unir ficheros,
- *sed* para hacer búsquedas/reemplazos,
- *tr, unique, sort* para otras transformaciones básicas.

Para ejecutar un programa escrito en AWK es necesario invocar al programa intérprete del lenguaje. Entre los intérpretes están: gawk de GNU, mawk, nawk, …

2. Sintaxis linea de comandos

La sintaxis para invocar al intérprete de AWK tiene dos posibles formas:

Sintaxis:

```
awk  [-F valor] [-v var=val] 'patrón {acción}'  [fichero1_entrada  …
                             ficheroN_entrada]
awk  [-F valor] [-v var=val] -f   fichero_script [fichero1_entrada  …
                             ficheroN_entrada]
```

Opciones:

1. La opción **-F** especifica el valor del separador de campos o columnas (el valor de la variable predefinida FS). Por defecto, se toman espacios y tabuladores.
2. La opción **-v** asigna valor *val* a la variable var antes de comenzar la ejecución del script.
3. La opción **–f** indica que las órdenes a ejecutar (script) se deben tomar del fichero especificado.

3. Como funciona awk

En un programa AWK, toda la entrada se lee de la entrada estándar (normalmente el teclado), tubería o de los ficheros cuyos nombres se especifican en la línea de comandos. Si se especifican ficheros de entrada, se leen los datos del primer fichero hasta que alcanza el final del mismo; después se lee el segundo fichero hasta que llega al final, y así sucesivamente.

El intérprete de AWK analiza un fichero de entrada línea a línea, buscando patrones. Cuando AWK encuentra una coincidencia, realiza una acción preestablecida. Si no se especifica un patrón, AWK realiza la acción para todas las líneas.

A diferencia de sed, que sólo proporciona un conjunto de comandos para procesar la entrada, AWK reconoce un lenguaje completo, de aspecto similar a C, como se indicó en el apartado anterior. También, tiene mayor potencia a la hora de reconocer patrones de la entrada, ya que permite especificar combinaciones de expresiones regulares en vez de una sola. Tampoco existe el límite de tener que

procesar la entrada línea a línea; AWK permite escoger el carácter que indica el fin de un registro (línea), y automáticamente separa cada registro en campos separados que pueden utilizarse individualmente.

4. Registros y campos

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. Los registros están separados por un carácter llamado separador de registros. Por defecto, el separador de registros es el carácter *newline* (\n). Por lo tanto, un registro se corresponde con una línea de texto. En el caso de querer utilizar un carácter diferente para separar los registros se puede usar la variable RS (*Record Separator*). El valor de RS es una cadena que dice como separar los registros. 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.

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. Se puede cambiar el separador de campo usando la opción -F de la línea de comandos o a través de la variable FS. Esta operación se debe realizar antes de que se procese ninguna entrada, de forma que el primer registro se lea con el separador adecuado. Para ello se puede utilizar el patrón especial BEGIN (ver apartado 5.1 Patrones).

Para referirse a un campo en un programa, se usa el signo de dólar '$', seguido por el número de campo. Por lo tanto, $1 se refiere al primer campo, $2 se refiere al segundo, y así sucesivamente. El número de un campo no necesita ser una constante, puede ser cualquier expresión del lenguaje AWK que siga al carácter '$'. El valor de la expresión especifica el número de campo.

5. Formato de un script awk

Un script AWK es un conjunto de reglas de la forma patrón-acción, con el formato que se describe a continuación. Cada regla especifica un patrón a buscar y una acción a realizar cuando encuentre dicho patrón en el registro de entrada. En la figura 1 se puede observar tanto la estructura de un script AWK, en la que se pueden diferenciar 3 partes: operaciones que se llevan a cabo antes, durante y después del procesamiento de la entrada, así como el flujo de ejecución del script.

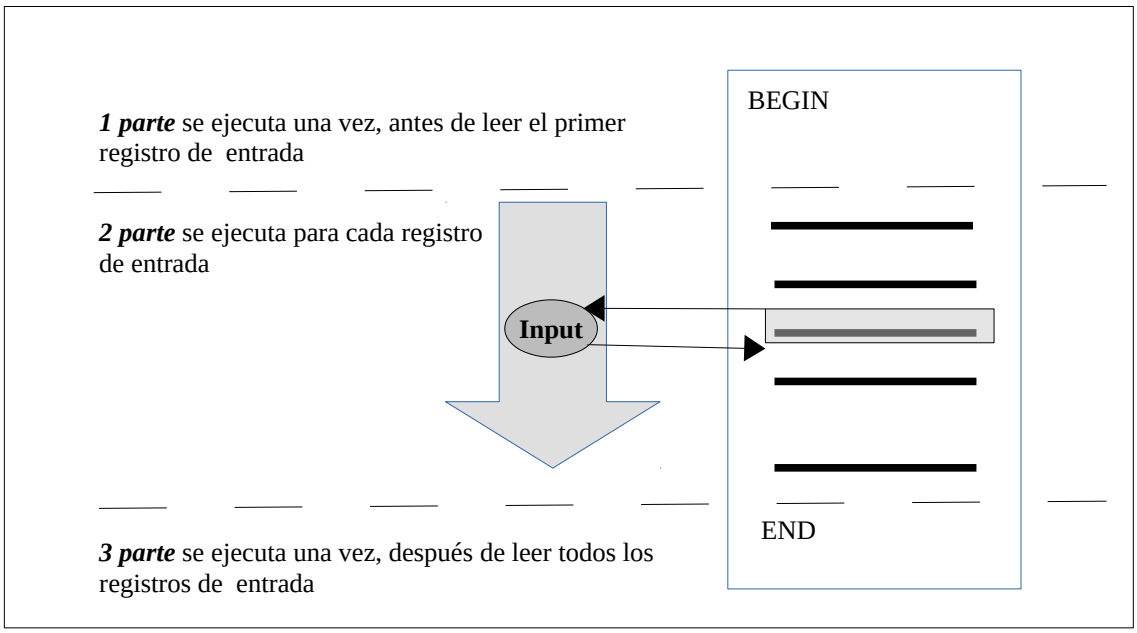

Figura 1: Flujo de control de un script AWK

Una acción es una o más sentencias que se ejecutarán si en el registro actual se cumple el patrón. Sintácticamente, una regla consiste en:

patrón {acción}

donde:

- las **llaves** son necesarias en AWK, por lo que suele ser necesario encerrar los programas de AWK entre comillas, para evitar que el shell las interprete como caracteres especiales;
- si no hay patrón, se ejecutarán las acciones en todos los registros;
- si no hay acciones, se ejecuta la acción por defecto, es decir, copiar el registro en la salida estándar.

A continuación se puede ver un resumen de algunas acciones que ofrece AWK como lenguaje de programación:

- Asignación de variables y arrays (=)
- Operaciones aritméticas simples (+,-,*,/,%, ++, --, ^)
- Comparaciones aritméticas (==,!=,>,<,>=,<=)
- Operaciones aritméticas avanzadas (sqrt, log, exp, int)
- Salida por pantalla (print y printf)
- Procesamiento de cadenas (substr, split, index, length)
- Estructuras de control (if…else, for, while). La estructura for se puede usar como en bash (for var in lista) o como en C.

Los **comentarios** empiezan por el carácter # y terminan con el carácter newline (\n)

Ejemplos:

- muestra información de aquellos usuarios que incluyan la cadena *bin* entre sus datos almacenados en el fichero */etc/passwd*

```
awk  -F":"  '/bin/ {print}' /etc/passwd
```

- muestra el grupo al que pertenece el usuario. El grupo es la columna 4 del fichero */etc/passwd*.

```
awk  -F":" ' {print $(2*2)}' /etc/passwd
awk  -F":" ' {print $4}' /etc/passwd
```

- muestra el primer campo del primer registro, el segundo campo del segundo registro, y así sucesivamente. Si el número de campos es menor que el número de registros, se mostraría una línea en blanco

```
awk '{ print $NR }' prueba.txt
```

siendo NR el número de registros leídos hasta el momento: 1 para el primer registro, 2 para el segundo, etc

- muestra aquellos usuarios definidos en el sistema que tienen en su *login* la cadena "*bin"*.

```
awk  -F":"  '$1~/bin/ {print}' /etc/passwd
```

El operador '~' se le llama operador de encaje, búsqueda o *matching*. Chequea si una cadena (en este caso, el campo $1) encaja con la expresión regular dada.

En los siguientes ejemplos se usará el fichero de entrada *telefonos.txt*. Los datos de este fichero están separados por espacios en blanco y su contenido es:

```
1 Profesor1 988111116
2 Profesor2 988222116
3 Profesor3 988333333
```

- Al no existir patrón, se muestra el primer campo de todas las líneas del fichero precedido por la cadena "numero: ":

```
awk '{ print "numero: " $1 }' telefonos.txt
```

Si los datos del fichero *telefonos.txt* están separados por el carácter ":", y no por un espacio en blanco, la orden anterior mostraría el contenido de todas las líneas precedidas por la cadena "numero: ". Si lo que se desea es sólo visualizar el primer campo precedido por dicha cadena, se deberá usar la opción -F para especificarlo, tal como se indica a continuación:

```
awk  -F":"  '{ print "numero: " $1 }' telefonos.txt
```

- Al omitir la acción, se muestran todas las líneas que contienen un 6:

```
awk   '/6/'   telefonos.txt
```

- Para aplicar más de una orden AWK, se repite las veces que se desee la construcción *patrón{acción}*, tal como se observa en el siguiente ejemplo donde se visualiza el nombre (segundo campo) de aquellas líneas que contienen el carácter 6 y el teléfono (tercer campo) de aquellas líneas que contienen la cadena *Profesor1*.

```
awk '/6/ {print $2} /Profesor1/ {print $3}' telefonos.txt
```

- También se pueden poner varias sentencias en la misma acción separándolas con el carácter ";". En el siguiente ejemplo se muestra el segundo y tercer campo (en líneas distintas) de aquellas líneas (registros) que contienen el carácter 6.

```
awk '/6/ {print $2; print $3}' telefonos.txt
```

Nota: La salida del ejemplo anterior es distinta a la salida de las siguientes órdenes:

```
awk '/6/ {print $2,   $3}' telefonos.txt
awk '/6/ {print $2 "  " $3}' telefonos.txt
```

En ambos casos se muestra el segundo y tercer campo en la misma línea separados por un espacio en blanco. Si no se especifica el espacio en blanco entre comillas dobles o no se pone la coma separando los campos, estos se muestran juntos. La sentencia *print* sustituye cada coma por el valor de la variable OFS que por defecto posee el espacio en blanco. (Ver apartado 5.3. Variables.)

5.1. 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.

AWK busca en el fichero de entrada todas las líneas que contengan el patrón de búsqueda. A continuación, se describen algunos de los patrones que se pueden emplear en las reglas:

- **BEGIN**
  END
  Los patrones BEGIN y END se usan 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 { cont=0;print " TOTAL DE APARICIONES" }
     /art1/ { ++cont }
    END { print "art1 aparece " cont " veces" }' articulos.txt
```

Este programa averigua cuantas veces aparece la cadena *"art1"* en el fichero de entrada *articulos.txt*. La regla BEGIN imprime un título para el informe de salida. La segunda regla incrementa el valor de la variable *cont* cada vez que se lee de la entrada un registro que contiene el patrón *"art1"*. La regla END imprime el valor de la variable *cont* al final de la ejecución.

Si un programa AWK tiene solamente una regla BEGIN, y ninguna otra regla, entonces el programa termina, 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.

- **/expresión regular/**
  Una expresión regular como patrón. Encaja cuando el contenido del registro de entrada concuerda con la expresión regular. Si a la expresión regular le precede el carácter !, selecciona los registros que no concuerdan con la expresión regular. AWK usa expresiones regulares extendidas (^ $. [ ] | ( ) * + ?).

  /expresión regular/, /expresión regular/
  Rango de líneas. Para seleccionar desde la línea que contenga la primera expresión regular hasta la primera línea (incluida) que contenga la segunda expresión regular. En el caso de que no se encuentre la segunda expresión regular, la acción se aplicará en todas las líneas hasta el final de los datos de entrada. Si el intervalo conteniendo las dos expresiones se repite varias veces, la acción se aplicará a cada intervalo sucesivamente.

- **expresiones relacionales**
  Una expresión relacional incluye constantes, variables, operadores y funciones. Una constante es una cadena o un valor numérico. Una variable es un símbolo que referencia un valor. Los operadores se utilizan para comprobar si algún campo verifica alguna condición. La condición se cumple si el resultado de evaluar la expresión es un valor de texto no nulo o un valor numérico distinto de cero.

- **var ~ /expresión regular/**
  Expresión de comparación. Esto es cierto si la expresión regular (tomada como una cadena de caracteres) encaja con el contenido de la variable var.

- **var !~ /expresión regular/**
  Expresión de comparación. Esto es cierto si la expresión regular (tomada como una cadena de caracteres) no encaja con el contenido de la variable var.

- **patrón1, patrón2**
  Un par de patrones separados por una coma, especificando un rango de registros o líneas.

Los patrones se pueden combinar utilizando los operadores lógicos && (operador and), || (operador or) o el signo de exclamación ! (not).

En los siguientes ejemplos se usará el fichero de entrada *alumnos.txt* descrito en la figura 2. Los datos de este fichero están separados por espacios en blanco:

```
Apellidos y Nombre, Grupo reducido, Convocatorias consumidas

Garcia Garcia Jose 1 primera
Lopez Lopez Alba 2 primera
Gonzalez Gonzalez Maria 2 primera
Martinez Martinez Carmen 2 primera
Morales Morales Pedro 1 segunda
Alvarez Alvarez Manuel 1 primera
Jimenez Jimenez Luisa 2 segunda
# Alumno repetidor
Gomez Gomez Antonio 1 primera
Sanchez Sanchez Teresa 1 segunda
Vazquez Vazquez Angel 2 primera
```

Figura 2: Fichero de texto *alumnos.txt*

- muestra todas las líneas del fichero:

```
awk ' {print}' alumnos.txt
awk '{print $0}' alumnos.txt
```

- muestra todas las líneas con la palabra *primera*

```
awk '/primera/ {print}' alumnos.txt
```

- guarda el nombre y primer apellido de los alumnos cuyo grupo reducido sea el 1 en el fichero GrupoSOII_1.txt:

```
awk ' $4 == 1 {print $3, $1}' alumnos.txt >GrupoSOII_1.txt
```

- muestra todas las líneas que contengan la letra 'G' en cualquier posición dentro del primer campo:

```
awk '$1 ~ /G/' alumnos.txt
awk '{ if ($1 ~ /G/) print }' alumnos.txt
```

- muestra todas las líneas cuyo primer campo (primer apellido) no contiene la letra 'G' en ninguna posición:

```
awk '$1 !~ /G/' alumnos.txt
```

- muestra desde la línea 4 a la 7:

```
awk 'NR==4,NR==7' alumnos.txt
```

- muestra el nombre y apellidos de los alumnos pertenecientes al grupo reducido 2, incluyendo al final del informe el número total de alumnos de este grupo:

```
awk 'BEGIN { cont=0} $4 == 2 {print  $3, $1; cont+=1} END {print
"Numero total de alumnos", cont}' alumnos.txt
```

- muestra el contenido del fichero incluyendo al principio un título y al final un texto con el número total de alumnos

```
awk 'BEGIN
      { print "LISTADO DE ALUMNOS DE SISTEMAS
OPERATIVOS\n"}
      {print $0 }
END {print "\nTOTAL DE ALUMN0S", NR}' alumnos.txt
```

- añade una línea en blanco después de cada línea

```
awk '{print; print " "}' alumnos.txt
```

- añade una línea en blanco después de cada línea (sin duplicar las líneas en blanco)

```
awk '{print; if (NF>0) print " "}' alumnos.txt
```

5.2. EXPRESIONES

Las expresiones son los bloques de construcción básicos de las acciones AWK.

Operadores aritméticos

El lenguaje AWK utiliza los operadores aritméticos comunes a otros lenguajes cuando se evalúan expresiones. Todos estos operadores aritméticos siguen las reglas de precedencia normales, y se comportan como se podría esperar de ellos.

Operadores	Descripción
+ -	Suma, resta
* / %	Multiplicación, división y módulo
^ **	Exponenciación

En el siguiente ejemplo se 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 modificado:

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

Operadores de asignación

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

Precedencia de Operadores

La precedencia de operadores determina como se agrupan los mismos, cuando aparecen distintos operadores cerca unos de otros en una expresión. Para saltar la precedencia de operadores se utilizan los paréntesis.

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 ejemplo, la expresión *a - b + c* se agrupa de la siguiente forma *(a - b) + c* y la expresión *a = b = c* se agrupa *a = (b = c)*.

A continuación se muesta una tabla de operadores de AWK, en orden de precedencia creciente:

Tipo	**Operadores**
Asignación	=, +=, -=, *=, /=, %=, ^=, **=
Condicional	?:
"o" lógico	||
"y" lógico	&&
Pertenencia a array	in
Encaje o patrón	~, !~
Relacional y direccionamiento	Los operadores relacionales son <, <=, ==, !=, >= y > Los operadores de redireccionamiento de la Entrada/Salida son <, >, >>, |
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).
Suma, resta	+, -
Multiplicación, división, módulo	*, /, %
Más y menos unario, "not"	+, -, !
Exponenciación	^, **
Incremento, decremento	++, --
Referencia a campo	$

5.3. VARIABLES

Las variables no tienen tipo ni tampoco hay que declararlas. Se asignan como en los programas del shell. Hay tres tipos: variables definidas por el usuario, variables del sistema o predefinidas y variables de campos. 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.

Las variables predefinidas tienen significados especiales, tal y como FS, el separador de campo, y NF, el número de campos en el registro de entrada actual. Estas variables se pueden usar y asignar como cualesquiera otras variables, pero sus valores son también usados o cambiados automáticamente por AWK. Todos los nombre de variables predefinidas 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 en AWK.

Se puede asignar un valor a cualquier variable AWK, incluyendo una asignación de variable entre los argumentos en la línea de comandos. La sintaxis sería:

var=expresión

Con ello, se puede dar un valor a una variable antes de la ejecución de un programa o entre el procesamiento de los ficheros de entrada.

Si se precede la asignación con la opción -v, tal y como se muestra a continuación:

```
awk -v hora=40 -f informeEmpleados.awk  empleados.txt
```

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 a todos los argumentos de nombres de ficheros.

Supongamos el siguiente script *usuariosgrupos.sh:*

```
#!/bin/bash
echo $1
cat /etc/group | awk -F":" '/^a/ {print $0} /^s/ {print $1}'
```

Los parámetros $0 y $1 de la línea 3 hacen referencia al registro y primer campo del registro, respectivamente. Mientras que el parámetro $1 de la línea 2 referencia a un parámetro de posición (argumento en la línea de comandos).

Una forma de pasar un parámetro de posición a AWK es usando la opción -v como en el siguiente ejemplo:

```
echo "El usuario $1 aparece en los siguientes grupos"
cat /etc/group | awk -F: -v user=$1 '$4 == user {print $0}'
```

donde se muestran los grupos (líneas) que contienen sólo al usuario que se le pasa como argumento al script. El primer $1 es el parámetro de posición y $4 hace referencia al campo 4 de cada una de las líneas del fichero */etc/group*. Este script *usuariogrupos.sh* se invocaría desde la línea de comandos:

```
./usuariogrupos.sh user1
```

En la siguiente tabla se muestra un resumen de las variables predefinidas que posee AWK:

Variables	Descripción
$1, $2, $3, ...	Variables de campos, contiene el valor de los campos 1, 2, 3, ...
$0	Contiene el valor de todo el registro
NR	Número del registro de entrada
FNR	Número del registro de entrada en el fichero actual
NF	Número de campos o columnas
$NF	Valor del último campo (columna)
FS	Separador de campos. Por defecto es "espacio en blanco", refiriéndose a " " y "\t". Se puede modificar con la opción de comando –F seguido del delimitador (Ejemplos: -F:, -F/)
RS	Separador de registros (filas). Por defecto es "newline", refiriéndose a "\n"
OFS	Separador de campo para el fichero de salida, por defecto tiene asignado el espacio en blanco
ORS	Separador de registros (filas) para el fichero de salida, por defecto tiene el valor "\n"
OFMT	Formato numérico. Por defecto "%.6g"
FILENAME	Nombre del fichero de entrada

6. IMPRIMIENDO LA SALIDA

Una de las operaciones más habituales que realizan las acciones es mostrar o imprimir parte o toda la entrada. Para salida simple, se utiliza la sentencia **print.** Para un formateo más elegante la sentencia **printf**.

La sentencia print

La sentencia print realiza la salida con un formato estándar y simple. Se especifica solamente las cadenas o números a imprimir 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 sintaxis:

print item1, item2, item3 …

La lista completa de ítems podría ser opcionalmente encerrada entre paréntesis. Los ítems 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 cualesquiera valores a imprimir. Con una excepción, lo que no se puede hacer es especificar como imprimirlos. Para eso, hay que usar la sentencia printf.

La sentencia simple print sin ningún ítem es equivalente a 'print $0': imprime el registro actual entero. Para imprimir una línea en blanco, se usa print "", donde "" es la cadena vacía o nula. Para imprimir una parte de texto fija, hay que utilizar una constante. Si no se ponen los caracteres comillas dobles, el texto será interpretado como una expresión AWK, y probablemente se obtendrá un error.

Ejemplo:

- muestra tres cadenas de texto en líneas distintas:
  ```
  awk 'BEGIN { print "línea uno\nlínea dos\nlínea tres" }'
  ```

- muestra los dos primeros campos de cada registro de entrada, con un espacio entre ellos:
  ```
  awk '{ print $1,$2 }' notas.txt
  ```

- añade una cabecera al listado anterior. Para ello se usa el patrón BEGIN, para hacer que la cabecera se imprima una sola vez:
  ```
  awk 'BEGIN {
  print "Prueba1 Prueba2"
  print "----- --------" }
  {
  print $1,$2 }' notas.txt
  ```

Separadores de la Salida

Como se mencionó anteriormente, una sentencia print contiene una lista de ítems, separados por comas. En la salida, los ítems son separados normalmente por simples espacios en blancos (separador por defecto). Se puede especificar cualquier cadena de caracteres para usarla como separador de campos de salida fijando la variable predefinida OFS. El valor inicial de esta variable es la cadena que contiene un espacio en blanco " ".

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 predefinida 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.

Se pueden asignar nuevos valores a las variables OFS y ORS. Lo normal es hacerlo en el patrón BEGIN, de modo que tomen sus valores antes de que se procese ninguna entrada. También se puede hacer con asignaciones en la línea de comandos, antes de los nombres de los ficheros de entrada.

En el siguiente ejemplo se 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 }'
listado.txt
```

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

Sentencia printf

Permite tener un control más preciso sobre el formato de la salida. La función printf permite elaborar verdaderos informes, ya que permite formatear la salida utilizando las mismas órdenes de formato que emplea el lenguaje de programación C, así como las secuencias de escape.

Su sintaxis es:

printf ("cadena de caracteres a visualizar junto con órdenes de formato", variables y/o campos a visualizar separados por comas)

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 cadena de formato u orden de formato. Entre comillado deben existir tantas órdenes de formato como el número de variables y/o campos pasados como argumentos a dicha función. Cada orden de formato comienza por el carácter de tanto por ciento, "%". La orden de formato *s* se usa para visualizar cadenas de caracteres, la *d* para números enteros y la *f* para números reales. Cuando se usa esta última orden de formato (%f), se puede especificar el número de decimales mediante un entero después del punto, mientras que el número de dígitos en la parte entera se especifica con el número entero que precede al punto o a la orden de formato *d*.

En los siguientes ejemplos se usará el fichero de entrada *empleados.txt* descrito en la figura 3. Los datos de este fichero están separados por espacios en blanco:

```
Apellido1 Apellido2 Nombre SueldoBase NumeroHorasExtraordinarias

Sanchez Prieto Juan Ramón 1800 22
Gómez Pérez Sonia  2100 35
Gutierrez Alonso Carmen 1900 0
González Álvarez Francisco Javier 2000 12
González Rufino María Encarnación 2000 45
De la Torre Martínez Juan Gabriel 1970 23
Estevez Morales Pablo 2200 0
Ruiz Pineda Maria 2300 6
Carrión Pardo Pilar 2600 4
```

Figura 3: Fichero de texto *empleados.txt*

El siguiente script AWK, *informeEmpleados.awk*, presenta un informe del fichero descrito anteriormente usando órdenes de formato, saltos de línea y tabuladores, así como cálculos aritméticos.

```
BEGIN {printf ("SALIDA DEL FICHERO %s CON FORMATO.\n", nombreFichero);
       printf ("NUMERO REG.     SUELDO BASE      NUMERO HORAS EXTRA
            SALARIO \n");
        horas=0;
        total=0; }
```

```
{ printf (" %d \t\t %.2f \t\t %2d \t\t %.2f \n", NR, $(NF-1), $NF,
($(NF-1) + ($NF*hora) ) );
      horas+=$NF;
      total+=$(NF-1)+($NF*hora);
}
END { printf ("\nNUMERO DE HORAS EXTRAORDINARIAS: %d \nTOTAL DE
NOMINAS: %.2f \n", horas, total); }
```

La ejecución de este script se realizaría mediante la siguiente línea de comandos:

```
awk -v nombreFichero="empleados.txt" -v hora=40 -f informeEmpleados.awk
                              empleados.txt
```

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 se quiere un carácter newline, se debe incluir uno en el formato. Las variables de separación de la salida OFS y ORS no tienen efecto sobre las sentencias printf.

7. ARRAYS

En AWK todos los arrays de una dimensión son **asociativos.** Esto significa que el array está indexado por cadenas. Cada array es una colección de pares: un índice y su valor. No es necesario especificar el tamaño de un array antes de empezar a usarlo. Solamente los valores son almacenados; los índices son implícitos por el orden de los valores. Una ventaja de un array asociativo es que se pueden añadir nuevas parejas en cualquier momento. Los arrays suelen implementarse internamente como tablas "hash". El acceso a cada elemento se realiza mediante un índice que puede ser de tipo entero o de tipo cadena. Por ejemplo:

array[1] o bien array["1"] array["Lopez"] o bien array[Lopez]

Para asignar valores a los elementos de un array se utiliza el operador de asignación:

array[índice] = valor

El siguiente ejemplo 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 y usando los números como índices. Después imprime las líneas ordenadas por sus números. Es un programa muy simple, y se confunde si encuentra números repetidos, huecos o líneas que no comiencen con un número.

```
{
if ($1 > max)
max = $1
lineas[$1] = $0
}
END {
for (x = 1; x <= max; x++)
print lineas[x]
}
```

Se pueden simular los arrays de dos o más dimensiones refundiendo varios índices en uno solo.

Para comprobar si un elemento corresponde o no a uno de los índices de un array asociativo se utiliza la siguiente sintaxis:

clave in array

que devolverá verdadero si clave es uno de los índices del array y falso en caso contrario. En el siguiente ejemplo, se puede observar que solo se visualizarán las líneas que tiene el fichero de entrada ordenadas por el primer campo, suponiendo que éste es un número entero (positivo o negativo), y no se visualizarán líneas en blanco si en el fichero de entrada no existe ninguna línea que comience por el número entero correspondiente.

```
{
      if (NR == 1){
            max = $1;
            min = $1;
      }
      else {
            if ($1 > max)
                  max = $1 ;
            if ($1 < min)
                  min = $1;
      }
      lineas[$1] = $0;
}
END {
      for (x = min; x <= max; x++)
            if (x in lineas)
                  print lineas[x];
}
```

Si el contenido del fichero de entrada fuese:

5 cinco
7 siete
9 nueve
2 dos
-1 menos uno
-7 menos siete

la salida de ejecutar el script sería:

-7 menos siete
-1 menos uno
2 dos
5 cinco
7 siete
9 nueve

Si se elimina del script la comprobación de que el valor de x corresponde o no a un índice del array lineas, la salida contendría tantas líneas en blanco como el número de líneas que faltan en el fichero de entrada para los distintos valores de x.

Recorrido de todos los elementos de un array

En programas que utilizan arrays, a menudo se necesita que se ejecute un conjunto de instrucciones para cada elemento de un array. Esto es fácil en otros lenguajes donde los arrays son contiguos y los índices están limitados a enteros positivos: el índice mayor es una unidad menor que el tamaño del array, por lo tanto el índice tomaría valores 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 (*var* in *miarray*)
acción

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

El siguiente ejemplo escanea los registros de entrada y anota las palabras que aparecen (al menos una vez), almacenando un 1 en el array, usando la palabra en cuestión como índice. La segunda regla escanea los elementos del array *palabras* para encontrar todas las palabras distintas que aparecen en la entrada. Imprime todas las palabras que tienen más de 10 caracteres y también imprime el número de veces que aparecen dichas palabras en el fichero de entrada.

```
# Almacena el valor 1 por cada palabra que aparece al menos una vez
{
for (i = 1; i <= NF; i++)
palabras[$i] = 1
}

# Encuentra cuantas palabras distintas que tengan más de 10 caracteres
END {
num_palabras = 0
for (x in palabras)
if (length(x) > 10) {
++num_palabras
print x
}
print num_palabras, "palabras con una longitud mayor que 10"
}
```

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 incluidas en el cuerpo del bucle.

La sentencia delete

Para eliminar un elemento individual de un array:

delete miarray[index]

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

El siguiente ejemplo elimina el elemento del array *freq* cuyo índice corresponde a la cadena "*teclado*":

```
delete freq["teclado"]
```

Para borrar un array completo se utiliza la sentencia delete sin especificar índice.

delete miarray

Siguiendo el ejemplo anterior, la siguiente sentencia eliminaría el array *freq*:

```
delete freq
```

8. SENTENCIAS DE CONTROL

Las sentencias de control 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.

La sentencia if

La sentencia if es una sentencia para la toma de decisiones de AWK. La sintaxis es:

if (expresión)
acción1
[else acción2]

El siguiente script indica si un numero es par o no:

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

Operador condicional

AWK proporciona un operador condicional que se encuentra también en el lenguaje de programación C. Su sintaxis es:

expr ? acción1 : acción 2

La sentencia *if then else* tiene su equivalente utilizando este operador.

Ejemplo:

```
(nota>=5) ? print "aprobado" : print "suspenso"
```

La sentencia while

La sintaxis de un bucle while es:

```
while (condición)
acción
```

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

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

La sentencia do-while

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

```
do {
acción
}while (condición)
```

El siguiente ejemplo imprime cada registro de entrada 10 veces:

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

La sentencia for

El bucle *for* es más conveniente cuando se conoce el número de iteraciones. La sintaxis es:

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

El siguiente ejemplo imprime los tres primeros campos de cada registro de entrada, un campo por línea:

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

La sentencia break

La sentencia break provoca la salida del bucle *for*, *while* o *do-while* más interno en el que está contenida.

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.

La sentencia next

La sentencia *next* fuerza a AWK a que detenga inmediatamente el procesamiento del registro actual y vaya a leer el siguiente registro. Esto significa que no se ejecutará ninguna sentencia más para el registro en curso. AWK lee el siguiente registro y comienza un nuevo ciclo aplicando la primera regla.

Por ejemplo, si un programa AWK trabaja sólo sobre registros con cuatro campos, y no se quiere que procese una entrada incorrecta, se podría usar esta regla cerca del principio del programa:

```
NF != 4 {
printf("linea %d se ha saltado: No tiene 4 campos", FNR) >
"/dev/stderr"
next }
```

de forma que las reglas siguientes no verán el registro incorrecto. El mensaje de error se redirecciona al flujo de salida de error estándar, como debiera ser para los mensajes de error.

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.

9. FUNCIONES

AWK dispone de un conjunto de funciones predefinidas (*Built-in*) que se pueden invocar desde los scripts. Para llamar a una función, se escribe el nombre de la función seguido por sus argumentos entre paréntesis.

Funciones numéricas

int(x) devuelve la parte entera de x, truncado hacia 0. Devuelve el entero más cercano a x, localizado entre x y 0. Ejemplo, int(3) es 3, int(3.9) es 3, int(-3.9) es -3, e int(-3) es -3 también.

sqrt(x) devuelve la raíz cuadrada positiva de x. Devuelve un error si x es un número negativo. Ejemplo, sqrt(4) es 2.

exp(x) devuelve el exponencial de x, o un error si x está fuera de rango.

log(x) devuelve el logaritmo natural de x, si x es positivo; sino, devuelve un error.

sin(x) devuelve el seno de x, con x en radianes.

cos(x) devuelve el coseno de x, con x en radianes.

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

rand() devuelve un número aleatorio. Los valores de *rand* son distribuidos uniformemente entre 0 y 1.

srand(x) fija el punto de inicio, semilla, para la generación de números aleatorios al valor de x.

systime() devuelve la hora actual en segundos desde el 1 de Enero de 1970.

strftime() toma un argumento numérico en segundos y devuelve una cadena que representa la fecha correspondiente.

Funciones de cadenas

index(cad_fuente, cad_buscar) busca la primera ocurrencia de la cadena *cad_buscar* en la cadena *cad_fuente* , y devuelve la posición donde se produce esa ocurrencia dentro de la cadena *cad_fuente*. Si no se encuentra *cad_buscar*, index devuelve un 0. Ejemplo:

```
awk 'BEGIN { print index("animal", "an") }'
```
devuelve un '1'

length(cad) devuelve el número de caracteres de la cadena *cad*. Si *cad* es un número, la función length devolverá el número de dígitos que representa dicho número. 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(cad, expreg) busca en la cadena *cad*, la subcadena más grande, y más a la izquierda que concuerde con la expresión regular, *expreg*. Devuelve la posición del carácter, o índice, de donde dicha subcadena empieza (1, si empieza al principio de la cadena). Si no se encuentra una subcadena que concuerde con la expresión regular, *expreg*, devuelve un 0.

Ejemplo:

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

Este programa busca las líneas que contienen la expresión regular almacenada en la variable *expreg*. Esta expresión regular se puede cambiar. Si la primera palabra (columna) de la línea es 'FIND', *expreg* se cambia por 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(cad, array, *sep*) divide la cadena *cad* utilizando *sep* como separador. Las subcadenas resultantes se almacenan en el vector *array.* La primera subcadena se almacena en array[1], la segunda en array[2], y así hasta el final. El valor de *sep* puede ser una expresión regular. Si se omite el argumento sep, se utiliza el valor de FS. La función split devuelve el número de elementos de *array*.
La función split, divide la cadena que se le pasa como primer argumento de forma similar a como las líneas de entrada son divididas en campos. Ejemplo:

```
split("999-432-221", codigo, "-")
```

parte la cadena `"999-432-221"` en tres campos usando '-' como separador. Fija los valores del array codigo tal y como sigue: codigo[1] = "999", codigo[2] = "432" y codigo[3] = "221" ; el valor devuelto por esta llamada a la función split es 3.

sprintf(formato, expresion1, expresion2, ...) devuelve (sin imprimirla) la cadena que la función printf habría impreso utilizando los mismos argumentos. Ejemplo:

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

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

sub(expreg, cad_re, cad_destino) sustituye la primera aparición de *expreg* por *cad_re* en la cadena *cad_destino*. Si se omite *cad_destino*, entonces por defecto se usa y modifica $0. Ejemplo:

```
cad = "ejem, ejemp, ejemplo"
sub(/ej/, "xxx", cad)
```

fija cad a `"xxxem, ejemp, ejemplo"`, reemplazando la ocurrencia más a la izquierda y más larga de 'ej' por 'xxx'.

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

Si el carácter especial '&' aparece en cad_re, se mantiene la subcadena que coincide con *expreg*. Por ejemplo:

```
awk '{ sub(/sistemas/, "& operativos"); print }'
```

cambia la primera ocurrencia de "sistemas" a "sistemas operativos" en cada línea de entrada.

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

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

gsub(expreg, cad_rep, cad_destino) es similar a la función sub, excepto que gsub reemplaza todas las coincidencias que encuentre con la expresión regular, *expreg*. El carácter "g" en gsub significa "global", lo cual significa reemplazar en cualquier sitio. Por ejemplo:

```
awk '{ gsub(/cn/, "Con núcleo"); print }'
```

reemplaza todas las ocurrencias de la cadena 'cn' por 'Con núcleo' para todos los registros de entrada. La función gsub devuelve el número de substituciones realizadas. Si *cad_destino*, se omite, entonces se utiliza el registro de entrada completo, $0.

substr(cad, inicio, long) devuelve una subcadena de longitud *long*, empezando en el carácter *inicio* de la cadena *cad*. Por ejemplo,

`substr("mariposa", 5, 3)` devuelve "pos"

Si no se especifica longitud, esta función devuelve una subcadena que comienza en *inicio* y su longitud es el número de caracteres hasta el final de la cadena *cad*. Por ejemplo,

`substr("mariposa", 5)` devuelve "posa".

tolower(cad) devuelve una copia de cad, 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("Sistemas Operativos")` devuelve "sistemas operativos"

toupper(cad) devuelve una copia de *cad,* 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("Sistemas Operativos")` devuelve "SISTEMAS OPERATIVOS".

Funciones 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 una tubería; entonces la tubería es cerrada.

system(comando_sistema) 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*.

Funciones definidas por el usuario

La sintaxis para las funciones definidas por el usuarios es:

function nombre_funcion(lista de parámetros) {sentencias}

Una función puede ser invocada desde cualquier punto del script. La definición de la función puede situarse al principio o final del programa, antes o después del bloque de instrucciones patrón{acción}. No se pueden dejar espacios en blanco entre el nombre de la función y el paréntesis de apertura de los argumentos. En el siguiente ejemplo se puede ver la definición de una función que calcula el mínimo de dos números y una llamada a la misma.

```
awk '{print $1, minimo($2,$3)}
function minimo(num1, num2)
{
if (num1 < num2) {menor=num1}
else  {menor=num2}
return menor
}' numeros.txt
```

La sentencia *return* no es obligatoria.

10. ENTRADA CON GETLINE

Hasta ahora se han obtenido los ficheros de entrada desde la entrada estándar o los ficheros especificados en la línea de comandos. El lenguaje AWK tiene una función o comando especial llamado **getline** que se usa para leer la entrada. Este comando permite combinar varios ficheros de entrada en el mismo programa.

getline

El comando *getline* se puede usar 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 se termina de procesar el registro actual y no se va a realizar ninguna alteración del mismo y se quiere procesar justo en ese momento el siguiente registro.

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.

Por ejemplo, el siguiente programa AWK lee el primer registro y lo muestra añadiendo "Linea impar". A continuación, getline lee el siguiente registro y lo almacena en $0 (columnas en $1, $2…) para mostrarlo de nuevo añadiendo "Linea par". Finalmente, no es necesario volver a indicar getline dado que al llegar al final del programa automáticamente AWK salta al siguiente registro.

```
awk '{
     print "Linea impar: " NR " " $0
     getline
     print "Linea par:   " NR " " $0
 }'
```

Esta forma del comando getline fija el valor de NF (el número de campos), NR (el número de registros leídos hasta ahora), FNR (el número de registros leídos del fichero de entrada actual), y el valor de $0. Hay que tener en cuenta que el nuevo valor de $0 se usa en el chequeo de los patrones de las reglas siguientes. 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.

getline var

Esta forma de getline lee un registro en la variable *var.* Esto es útil cuando interesa que el programa lea el siguiente registro del fichero de entrada actual, pero no interesa que ese registro sea procesado como una entrada normal.

Por ejemplo, suponer que la siguiente línea es un comentario, o una cadena especial, e interesa leerla, pero no se quiere que dispare ninguna regla. Esta versión de getline 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.

Por ejemplo, este programa intercambia cada dos líneas de entrada.

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

Dada esta entrada:

```
uno
dos
tres
cuatro
```

produce la siguiente salida:

```
dos
uno
cuatro
tres
```

En este caso la función getline fija solamente las variables NR y FNR (y por supuesto, var). 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 < fich

Esta forma de la función getline toma su entrada desde el fichero *fich*.

Por ejemplo, el siguiente programa lee su registro de entrada del fichero *otrofich.txt* cuando encuentra un primer campo con un valor igual a 10 en el fichero de entrada actual.

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

Debido a que el flujo 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 modificados por el contenido del registro leído del fichero *fich*. Lo mismo le ocurre al valor de NF. Esto hace que el registro leído del fichero *fich* no sea chequeado contra todas las reglas del programa AWK. Sin embargo, este registro es chequeado contra las reglas restantes, del mismo modo que ocurría cuando se usaba getline sin la redirección.

getline var < fich

Esta forma de la función getline toma su entrada del fichero *fich* y la pone en la variable *var*. En esta versión de getline, ninguna de las variables predefinidas cambian su valor, y el registro no es dividido en campos. La única variable que cambia es *var*.

Por ejemplo, el siguiente programa copia todos las líneas de la entrada a la salida, excepto los registros cuyo primer campo coincide con *@include*. Para tales registros se muestra en la salida el contenido de todo el fichero *librerías.txt*.

```
awk '{
if ($1 == "@include") {
   while ((getline linea < “librerias.txt”) > 0)
      print linea
   close(“librerias.txt”)
} else
   print
}'
```

La función *close* se llama para asegurarse que si aparecen dos o más líneas *@includ*e, el fichero *librerias.txt* se incluye completo para cada una de ellas.

11. SCRIPT CON AWK

En los script del shell, los parámetros de posición o argumentos se guardan en las variables $1, $2, $3 … En AWK las variables $1, $2, $3 … guardan los valores de los campos o columnas. Por lo tanto, para usar parámetros del script en llamadas a AWK hay que almacenarlos previamente en otras variables.

Por ejemplo, hacer un script llamado *prueba.sh* que, mediante AWK, devuelva un campo de un fichero. Campo y fichero deben ser parámetros del script.

```
#!/bin/bash
file=$1
awk -v num=$2 '{print $num}' $file
```

La línea de comandos para ejecutar el script anterior podría ser: ***./prueba.sh fich.txt 2*** (mostraría el segundo campo de cada línea del fichero *fich.txt*).

En estos casos, no podemos usar *awk –f* y tener el programa AWK en otro fichero, pues no podríamos pasarle los parámetros del script.