Introducción a AWK

Francisco Alonso Sarría

Índice

1 Introducción **2**

1.1 Adaptar awk 4

2 Patrones y acciones **4**

3 Variables y arrays **5**

4 Entrada y salida de datos **6**

4.1 Entrada de datos 6

4.2 Salida de datos 6

5 Estructuras de control **8**

5.1 Toma de decisiones 8

5.2 Bucles 9

6 Funciones **11**

6.1 Funciones matemáticas 11

6.2 Funciones de manejo de cadenas de caracteres 11

6.3 Funciones definidas por el usuario 13

7 AWK y BASH **13**

7.1 Entrada de parámetros 13

7.2 Formateo de ordenes 14

7.3 Llamadas al sistema 15

1 Introducción

AWK es un lenguaje de programación cuya potencia estriba en la capacidad de obtener grandes resultados con programas de unas pocas lineas. Está especialmente diseñado para leer y procesar archivos de texto por lo que resulta muy útil su uso combinado con utras utilidades del sistema operativo.

Para ejecutar un programa escrito en AWK es necesario llamar al programa intérprete del lenguaje (*awk*) utilizando como parámetros un programa, escrito entre comillas simples, y uno o varios ficheros para procesar de acuerdo con ese programa.

AWK *asume* que va a tener que procesar un flujo de datos (entrada estandar, fichero de texto, tubería) y que este flujo está medianamente estructurado en registros (lineas) y campos (columnas).

Por tanto *sabe* que tiene que leer cada una de sus lineas como si fuese un registro, separar ese registro en campos, hacer lo que se le ordene con esos campos y finalmente producir un flujo de salida. Así que el programador apenas tiene que introducir ningún tipo de instrucción al respecto en el código.

Por ejemplo en la orden:

```
awk '{print}' fichero.txt
```

el programa de AWK es sólo `{print}`. Este programa lee todas las lineas del archivo `fichero.txt` y las muestra en pantalla. En casi todos los ejemplos vamos a trabajar con el archivo `fichero.txt` cuyo contenido es:

Murcia	3	2	3	4
Albacete	3	4	5	3
Almería	3	3	2	
Alicante	4	5	2	2

En algunos de los ejemplos que se verán el programa completo será lo suficientemente corto para poder aparecer entero en el texto, en estos casos el código empieza con la llamada al intérprete (`awk`)[1]. En otros casos se presentarán lineas de código de AWK aisladas.

Una de las peculiaridades que permite al intérprete de AWK trabajar como lo hace, es que al leer cada linea da valores a una serie de variables predefinidas:

- `NR` es el número de la linea que ha leido

[1]En linux puedes encontrar otos intérpretes de AWK como gawk o nawk.

- `NF` es el número de campos en la linea que ha leido
- `$0` contiene toda la linea leida
- `$1`, `$2`, ... `$NF` cada uno de los campos leidos

Por ejemplo la orden

```
awk '{print $NR,$0 }' fichero.txt
```

muestra en pantalla las lineas del fichero numeradas.

1	Murcia	3	2	3	4
2	Albacete	3	4	5	3
3	Almería	3	3	2	
4	Alicante	4	5	2	2

La orden:

```
awk '{print $1," ",$4}' fichero.txt
```

mostrará los campos primero y cuarto separados por 3 espacios (uno por cada coma y otro que corresponde a la cadena de caracteres entre las comas):

Murcia	3
Albacete	5
Almería	2
Alicante	2

Entre las principales utilidades de AWK destaca:

- Permite producir resúmenes a partir de grandes listados de datos medianamente estructurados
- Los programas son mucho más cortos que los equivalentes en otros lenguajes
- La posibilidad de crear programas de una sola linea embebidos en *shell scripts* e integrados con otras utilidades del sistema mediante redirecciones o tubería. Por ejemplo la orden $ls - l$ genera un listado largo de ficheros en los que el quinto campo corresponde al tamaño, así la orden:

  ```
  ls -l|awk '$5>200000{print $0}'>grandes.txt
  ```

  modifica la orden ls para que obtengamos sólo los ficheros mayores de 200 Kbytes y guarda ese listado en el fichero `grandes.txt`.

1.1 Adaptar awk

La adaptación del sistema linux al lenguaje español ha traido algunas complicaciones, como por ejemplo la sustitución del punto decimal por la coma. Esto no sería un problema en programas de AWK que funcionasen aislados, pero si es un inconveniente cuando se utiliza para procesar información de o hacia otros programas que si usan el punto decimal. Para evitar este inconveniente basta con ejecutar, antes de la llamada a AWK, las siguientes órdenes de BASH.

```
unset LC_ALL
export LC_NUMERIC=C
```

2 Patrones y acciones

En el último ejemplo de la sección anterior se ha visto como awk trataba de manera diferente a las lineas cuyo quinto campo era mayor que 200000 (las reconducía a la salida) que a las que no (se eliminaban). En general, en cualquier programa, las acciones que deben desarrollarse van a depender de los valores leidos. Para facilitar el trabajo, awk dispone de una estructura `patrón {acción}`. Es decir se definen varias acciones (separadas con llaves) que se ejecutan sólo si la linea leida cumple el patrón que aparece al inicio. Por ejemplo:

```
awk '$3>=4{print $0)' fichero.txt
```

presentará en pantalla sólo aquellos casos en los que la tercera columna tenga una valor mayor que 4.

Aparte de permitirnos definir los patrones que queramos, awk introduce dos patrones especiales BEGIN y END. El primero permite definir acciones que se ejecutarán antes de empezar a procesar el fichero de entrada, el segundo acciones que se ejecutaran al final del proceso.

```
awk '
    BEGIN{print "Provincia V1 V2 V3 V4"}
    $3>4{print $0}
    END{print "ADIOS."}
' fichero.txt
```

El patrón BEGIN permite tambien modificar algunas variables clave para el funcionamiento de AWK como `FS` y `RS` que definen, respectivamente, el separador de campos (por defecto un espacio en blanco) y el separador de registros (por defecto la nueva linea).

Así `BEGIN {FS=";"}` permite procesar ficheros en los que los campos se separen por punto y coma.

En definitiva la estructura de un programa de AWK se basa en el siguiente esquema:

```
BEGIN {acción}
patrón {acción}
  ..      ..
patrón {acción}
END {acción}
```

Hay que tener en cuenta que si una linea cumple con varios patrones se ejecutarán todas las acciones asociadas en el mismo orden en que aparecen en el programa.

3 Variables y arrays

Ya has visto como el interprete de AWK da valor a una serie de variables internas conforme va leyendo el flujo de datos de entrada. Lógicamente el usuario puede también definir sus propias variables. Por ejemplo:

```
awk '{V4=$4;print NR,V4}' fichero.txt
```

También se admite el uso de arrays asociativos, es decir vectores de datos indexados por una variable arbitraria, una cadena de caracteres por ejemplo:

```
awk '
    {V1[$1]=$2;V2[$1]=$3}
    END{print V2["Albacete"]}
' fichero.txt
```

El anterior ejemplo utiliza el primer campo del flujo de entrada como variable para indexar dos arrays.

4 Entrada y salida de datos

4.1 Entrada de datos

Ya se ha visto como **awk** está especialmente concebido para leer ficheros sin necesidad de funciones especiales. Sin embargo en algunos casos se hace necesario combinar varios ficheros de entrada en un sólo programa. Hay dos opciones:

- Leerlos uno detrás del otro. Para ello es necesario darle al programa algún método para distinguir un fichero del siguiente.
- Leerlos con la función $getline$, por ejemplo $linea = getline <'' fichero.txt''$
  - `getline`, lee una nueva linea del flujo de entrada
  - `getline linea`, lee una nueva y la guarda en `linea`
  - `getline linea < "fichero"`, lee una linea de `fichero` y la guarda en `linea`
  - `close(fichero)`; cierra un fichero o flujo de texto abierto con alguna llamada a getline.

4.2 Salida de datos

La función básica de salida en AWK es `print`

```
awk '{nombre="Pepe";print "Hola",nombre}
```

dará como resultado:

```
Hola Pepe
```

Dentro de un programa de AWK pueden utilizarse las mismas redirecciones que en BASH para llevar la salida a un fichero:

```
print "Hola a un fichero">"fichero_salida.txt"
print "Hola a un fichero">>"fichero_salida.txt"
```

Lógicamente el fichero de salida podría venir definido por una variable:

```
salida="fichero_salida.txt"; print "Hola a un fichero">"$salida
```

%d	Número entero
%nd	Número entero formateado a n caracteres
%f	Número real
%$m.n$f	Número real con n decimales formateado a m caracteres
%s	Cadena de carácteres

Tabla 1: Códigos de formato de printf (versión simplificada)

El separador de campos para la salida del programa es, por defecto el espacio en blanco; el separador de registros es el retorno de carro. Ambos separadores pueden modificarse con las variables `OFS` y `ORS`

Más sofisticada que `print` es la función `printf` donde la f final significa *formateada*, es decir permite determinar exactamente que forma tendrá la linea de salida de manera muy similar a como lo hace la orden printf en BASH o C. Por ejemplo[2]:

```
awk '{
    printf("Registro número %d:   Provincia=%s   Variable 1=%d \
            Variable 2=%5d\n",NR,$1,$2,$3)
}' fichero.txt
```

produce la siguiente salida:

```
Registro número 1:   Provincia=Murcia   Variable 1=3   Variable 2=    2
Registro número 2:   Provincia=Albacete   Variable 1=3   Variable 2=    4
Registro número 3:   Provincia=Almería   Variable 1=3   Variable 2=    3
Registro número 4:   Provincia=Alicante   Variable 1=4   Variable 2=    5
```

`printf` es una función y por tanto sus argumentos se sitúan entre paréntesis. El primer argumento es el formato entrecomillado, posteriormente se incluyen como argumentos las variables que se van a escribir. En el formato se puede incluir cualquier conjunto de carácteres más combinaciones especiales de caracteres que se inician con el carácter % y que indican como se van a imprimir el resto de los argumentos de la función (ver tabla 1):

Utiliza también caracteres especiales como `\n` que significa salto de linea o `\t` que significa tabulador.

[2]En la siguiente orden se utiliza la contrabarra (\) para indicar que la linea se ha partido por problemas de espacio y que continua en la linea siguiente. A la hora de escribir el código deberás juntar ambas lineas en una.

5 Estructuras de control

En la gran mayoría de los programas, el proceso de los datos requiere ejecutar alguna tarea repetidas veces o decidir realizar una tarea u otra en función de los valores de alguna variable. Las instrucciones para llevar a cabo estas acciones constituyen las estructuras de control.

Todas las estructuras de control comienzan con una palabra clave como **tt** `for` o `while`

5.1 Toma de decisiones

En ocasiones se debe romper el flujo de un programa y ejecutar un grupo de instrucciones u otras en función de los valores que adopta una variable. Existen dos tipos de funciones para la toma de decisión. las `if ... else` y las `switch`

El esquema básico de las instrucciones `if ... else` es:

```
if (condición) {instrucciones1} else {instrucciones2}
```

Donde instrucciones1 es el conjunto de ordenes que se ejecutan si condición se cumple e instrucciones2 las que se ejecutan en caso contrario.

```
if (a>20){ printf("%d es mayor que 20\n",a)}
else{printf("%d no es mayor que 20\n",a)}
```

Esta orden puede modificarse en caso de que existan varias condiciones:

```
if (a<5){
      print "Pequeño"
} else if (a<10){
      print "Mediano"
} else if (a<20){
     print "Grande"
} else{
     print "Muy grande"
}
```

5.2 Bucles

Existen dos tipos fundamentales de bucles, los bucles *FOR* y *WHILE* cuyo comportamiento es ligeramente distinto. La orden

```
for (i=1;i<=10;i=i+1){printf("Número %d\n",i)}
```

comienza dando el valor 1 a la variable contador `i`, y ejecutará la orden entre llaves mientras se cumpla que `i<=10`, en cada ejecución se incrementará en 1 el valor de `i`.

El bucle `while` actúa de forma algo diferente. Por ejemplo la orden:

```
i=0
while (i<=10){i=i+1;printf("Número %d\n",i)}
```

hacen los mismo, sólo que ahora la variable contador se inicializa fuera de la orden y se incrementa dentro de la orden (en realidad el contador de un bucle FOR también podría modificarse dentro del bucle).

Las instrucciones `break`, `continue` y `exit` alteran también el orden de ejecución. La primera fuerza al flujo del programa a salir de un bucle `for` o `while` y continuar con la siguiente orden. La segunda vuelve a la orden inicial del bucle (e incrementa el contador en el caso del bucle FOR. La última sale totalmente del programa.

Así el siguiente programa mostrará en pantalla los números del 1 al 10:

```
awk '
   BEGIN{for (i=1;i<=10;i++){print i}}
' fichero.txt
```

La siguiente modificación se saltará el número 3.

```
awk '
   BEGIN{for (i=1;i<=10;i++){if(i==3){continue};print i}}
' fichero.txt
```

Finalmente el siguiente programa mostrará sólo el 1 y el 2 ya que saldrá totalmente del bucle al llegar a 3.

```
awk '
   BEGIN{for (i=1;i<=10;i++){if(i==3){break};print i}}
' fichero.txt
```

En los tres ejemplos anteriores se incluye un archivo de entrada aunque no se utiliza, sino el intérprete quedaría parado esperando un flujo de datos de entrada.

El siguiente ejemplo muestra como combinar `while` con `getline` y `close` para leer adecuadamente un fichero distinto al que se le pasa como parámetro al intérprete del lenguaje:

```
awk '{
      igual=0
      while ((getline linea < "nuevo_fichero")>0){
          if($0==linea){igual=1;break}
      }
      if (igual==0){print $0}
      close("nuevo_fichero")
}' fichero.txt
```

Este programa mostrará sólo las lineas de `fichero.txt` que no aparezcan en `nuevo_fichero`.

Se puede modificar en cualquier momento el orden de ejecución de un programa con la orden `next` que detiene el procesamiento del registro actual y pasa a leer el siguiente. El programa:

```
awk '{
   if(NR==2){next}
   print $0
}' fichero.txt
```

saltará la segunda linea de `fichero.txt` sin mostrarla en pantalla.

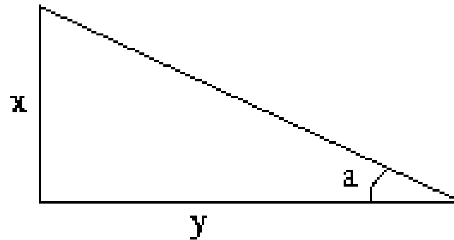

Figure 1: Función `atan2`

6 Funciones

Uno de los elementos básicos de cualquier lenguaje de programación son las funciones. Su objetivo es encapsular, como un *subprograma*, una serie de instrucciones que tienen consistencia por si mismas y que son llamadas de forma habitual y siempre de la misma forma por el programa. Las funciones generan uno o más datos de salida a partir de unos datos de entrada.

Existen múltiples funciones ya disponibles en cualquier lenguaje de programación. Pueden dividirse en:

6.1 Funciones matemáticas

int(x),sqrt(x),exp(x),log(x),sin(x),cos(x),atan2(y,x),rand(),srand()

La función `atan2(y,x)` calcula el ángulo `a` de la figura 1 a partir de los valores de los catetos `y` y `x`.

.

La función `rand()` genera números aleatorios mientras que `srand()` modifica la semilla del generador de números aleatorios.

6.2 Funciones de manejo de cadenas de caracteres

- `length(cadena)`, devuelve la longitud de la cadena de caracteres `cadena`.

  ```
  awk '{print $1,length($1)}' fichero.txt
  ```

- `index(cadena,caracter)`, devuelve el orden en que `caracter` aparece por primera vez en `cadena`.

```
awk '{print $1,index($1,"a")}' fichero.txt
```

- `split(cadena,subcadena,sep)`, divide la cadeda de caracteres `cadena` utilizando `sep` como carácter separador. Las subcadenas resultantes se almacenan en el array `subcadena`. Esta función devuelve el número de elementos en `subcadena` tras el proceso.

```
echo "fichero1.jpg fichero2.png"|awk '{split($1,cad,".")}' fichero.txt
```

- `sprintf(formato,argumento1,...)`, funciona igual que `printf` salvo que la cadena de caracteres producida se almacena en una variable:

```
awk '{var=sprintf("El valor es %f\n",$1);print var}' fichero.txt
```

  en este ejemplo en la variable `var`.

- `sub(expresion,sustituto,cadena)`, sustituye la primera aparición de `expresion` por `sustituto` en `cadena`

```
awk '{sub("l","L",$1);print $1}' fichero.txt
```

- `gsub(expresion,sustituto,cadena)`, igual que en la expresión anterior pero el cambio se realiza para toda la cadena.

- `substr(cadena,inicio,longitud)`, devuelve una subcadena de `cadena` que comienza en el carácter tt inicio y tiene un tamaño de `longitud` caracteres.

```
awk '{print substr($1,1,3)}' fichero.txt
```

- `toupper(cadena)`, convierte todos los caracteres de `cadena` a mayúsculas.

```
awk '{print toupper($1)}' fichero.txt
```

- `tolower(cadena)`, convierte todos los caracteres de `cadena` a minúsculas.

- `match(cadena,regexp)`, comprueba que la expresión regular `regexp` case con `cadena` que contiene un patrón de texto.

```
awk '{if(match($1,"te")){print $1}}' fichero.txt
```

En este ejemplo se ha utilizado un patrón de texto sencillo, pueden utilizarse también expresiones regulares que permiten búsquedas mucho más potentes, aunque también son bastante más complejas de manejar.

6.3 Funciones definidas por el usuario

Para utilizar una función, es necesario definirla; esta definición puede situarse al principio o al final del programa, antes o despues por tanto del bloque de instrucciones `patrón {acción}` y se corresponde con el siguiente esquema:

```
function nombre(lista\_de\_argumentos)
{
   instrucciones
}
```

`nombre` es el nombre de la función, con el cual será llamada en el programa. La `lista_de_argumentos` incluye todas las variables que necesita la función para ejecutarse y que son externas a ella. Las instrucciones procesan la lista de argumentos y generan un resultado que la función devuelve mediante la instrucción `return`. En el siguiente ejemplo aparece la definición de una función que cálcula el máximo de dos valores y una llamada a la misma.

```
awk '{print $1,maximo($2,$3)}
function maximo(a,b)
{
   if (a>b){return a}
   else{return b}
}' fichero.txt
```

7 AWK y BASH

7.1 Entrada de parámetros

La opción `-v` en la llamada a AWK permite al usuario, o al script que hace la llamada, introducir una variable al programa:

```
awk -v dato=26 '{
    printf("Registro número %d:   Provincia=%s   Variable 1=%d \
            Dato=%d\n",NR,$1,$2,dato)
}' fichero.txt
```

```
Registro número 1:   Provincia=Murcia   Variable 1=3   Dato=26
Registro número 2:   Provincia=Albacete   Variable 1=3   Dato=26
Registro número 3:   Provincia=Almería   Variable 1=3   Dato=26
Registro número 4:   Provincia=Alicante   Variable 1=4   Dato=26
```

AWK puede utilizarse como un apoyo matemático a la programación en BASH. En ese caso en lugar de leer los datos de entrada desde un fichero es más eficiente pasarlos a traves de una tubería. El siguiente ejemplo:

```
cateto1=4
cateto2=3
hipotenusa=$(echo $cateto1 $cateto2|awk '{print sqrt($1^2+$2^2)}')
echo $hipotenusa
```

asignará a la variable `hipotenusa` el valor 5 y lo presentará en pantalla. Como ves las variables `cateto1` y `cateto2` se han convertido en las variables posicionales `$1` y `$2`. La salida de awk se ha almacenado como una variable de BASH poniendo el proceso entre paréntesis y situándole un `$` delante.

7.2 Formateo de ordenes

AWK puede utilizarse para crear órdenes para otros programas. En este caso, la salida de AWK debe redirigirse mediante una tubería al intérprete de instrucciones del lenguaje de programación que se ha utilizado.

La siguiente orden hace una consulta a una base de datos postgres para obtener un listado de usuarios y a partir de ella genera las ordenes necesarias para dar a cada uno de los usuarios permiso de consulta a la tabla cuyo nombre se pasa a awk mediante la opción `-v`.

```
 psql -c "\du" grass|awk -F "|" -v tabla=mi_tabla 'NR>3 && NF>1{
       gsub(" ","",$1);
       printf("GRANT SELECT ON %s TO %s;\n",tabla,$1)
}'|psql grass
```

Al final, la salida del programa de AWK se redirige de nuevo a la base de datos mediante una tuberia para que se ejecuten las órdenes creadas.

7.3 Llamadas al sistema

En ocasiones puede ser interesante hacer una llamada a la BASH desde un programa de AWK, para ello se utiliza la orden `system`.

```
listado=system("ls")
```

El ejemplo anterior hace un listado y guarda el resultado en una variable.