Introducción a AWK

PID_00270631

Guillem Lluch Moll

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

Guillem Lluch Moll

Licenciado en Matemáticas por la Universidad Autónoma de Barcelona (UAB), máster en Software libre por la Universitat Oberta de Catalunya (UOC) y máster en Lenguajes y sistemas informáticos (UNED). De larga trayectoria docente, en diferentes niveles y asignaturas, actualmente trabaja como coordinador TIC en un instituto de secundaria.

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Julià Minguillón Alfonso (2020)

Primera edición: febrero 2020

Avda. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

1. Funcionamiento y sintaxis

El programa **AWK**, al igual que Sed, funciona línea a línea. Pero en lugar de ver cada línea como un todo, como un único elemento, AWK considera que **las líneas están formadas por campos**, separados por uno o más espacios, aunque este separador se puede cambiar.

La sintaxis con AWK consiste en un patrón y un procedimiento. Los dos son opcionales:

- Si no hay ningún patrón (pero hay una o más acciones), la acción se lleva a cabo para todas las líneas.

- Si no hay ningún procedimiento (pero hay un patrón), se imprimen las líneas que coinciden con el patrón.

Los patrones siguen el formato **ERE**.[1] Los comandos AWK se escriben entre comillas simples para evitar conflictos con la *shell*.

[1] Acrónimo de *extended regular expressions*.

En cada línea, AWK enumera los campos. Nos podemos referir a estos anteponiendo `$` en el lugar que ocupa: así `$1` es el primer campo, `$2` el segundo, etc., `$0` es toda la línea y `$NR`, el último campo.

Los procedimientos de AWK son muy variados e incluyen las acciones típicas de los lenguajes de programación estructurados (`if-else`, `for`, `while`, etc.). Su sintaxis es muy similar a la de C.

> Podemos entender AWK como un lenguaje de programación en el que la iteración línea a línea viene dada por defecto, automáticamente.

AWK se puede utilizar escribiendo las órdenes en la línea de comandos o desde un fichero, empleando la opción `-f`. Las acciones se pueden aplicar sobre un fichero o se pueden redirigir a la salida de un comando previo (como, por ejemplo, `echo` o `cat`). Ejemplos:

```
# Imprime la línea de entrada con la función print. Los procedimientos se ponen entre llaves.
echo "field1 field2 field3" | awk '{print $0}'

field1 field2 field3
```

```
# Se aplica el patrón de encontrar una f literal. Dado que no hay ningún procedimiento, se imprime.
echo "field1 field2 field3" | awk '/f/'

field1 field2 field3
```

```
# Se aplica el patrón de encontrar una c literal. No coincide y, por lo tanto, no se muestra nada.
```

```
echo "field1 field2 field3" | awk '/c/'
```

```
# Se puede emplear más de un procedimiento a la vez, separados por ;.
# Imprime el primer y el tercer campo.
echo "field1 field2 field2bis" | awk '{print $1; print $3}'

field1
field2bis
```

```
# Se puede emplear más de un argumento en print separados por ,.
# Imprime el primer y el tercer campo en la misma línea, separados por espacio.
echo "field1 field2 field2bis" | awk '{print $1,$3}'

field1 field2bis
```

```
# Para concatenar se deja un espacio en blanco.
# Imprime el primer y el tercer campo en la misma línea, sin ninguna separación.
echo "field1 field2 field2bis" | awk '{print $1 $3}'

field1field2bis
```

Para ilustrar las explicaciones, usaremos el fichero «sales.csv» (generado en mockaroo.com/).

```
head -n6 sales.csv

id,first_name,last_name,email,gender,country,2017,2018,2019
1,Lulu,Chaloner,lchaloner0@reuters.com,Female,France,77.73,85.9,60.46
2,Vivianne,Feeney,vfeeney1@washingtonpost.com,Female,United Kingdom,10.22,51.01,49.88
3,Waring,Craythorne,wcraythorne2@merriam-webster.com,Male,France,3.7,35.21,72.35
4,Wakefield,Shepton,wshepton3@sphinn.com,Male,France,78.39,86.02,86.35
5,Beale,Myrtle,bmyrtle4@bloomberg.com,Male,United Kingdom,68.06,77.78,40.26
```

Lo primero que tenemos que hacer es cambiar el separador de AWK por defecto. En lugar de espacio, usaremos la coma. Esto se hace con la opción `-F` (en mayúsculas) de AWK.

```
# Obtiene el nombre. Por razones de espacio, se redirige la salida y
# solo se muestran los 5 primeros nombres.
awk -F, '{print $2}' sales.csv | head -n6

first_name
Lulu
Vivianne
Waring
Wakefield
Beale
```

Hay acciones que solo interesa hacer una vez, no en cada línea. Con AWK, podemos realizar acciones no ligadas a ningún registro de entrada **antes**, dentro de la sección `BEGIN`, o **después**, en la sección `END`. Tanto una como otra son optativas.

```
# BEGIN se emplea mucho para asignar un valor a la variable FS,
# que es la que contiene el separador de los campos.
# Antes de hacer nada, establecemos la coma como separador de los campos.
# Para cada entrada, imprimimos el segundo campo.
awk 'BEGIN {FS=","} {print $2}' sales.csv | head -n6

first_name
Lulu
Vivianne
Waring
Wakefield
Beale
```

```
# Ejemplo de END.
# Antes de hacer nada, establecemos la coma como separador de los campos.
# Para cada entrada, imprimimos el segundo campo.
# Una vez procesadas todas las líneas, se imprime un mensaje que informa del final.
awk 'BEGIN {FS=","} {print $2} END {print "OK. Final"}' sales.csv |tail -n6

Sigismondo
Liuka
Tynan
Aylmer
Alexandr
OK. Final
```

Cuando hay varios procedimientos AWK que se quieren ejecutar, conviene ponerlos en un fichero y llamarlo con `-f`:

```
# Muestra el fichero del script, que produce el mismo resultado que el anterior.
cat surnames
```

```
# Muestra el segundo campo de un fichero separado por comas e imprime un mensaje final.
BEGIN {FS=","}
{print $2}
END {print "OK. Final"}
```

Fijaos que podemos incluir comentarios en los *scripts* de AWK. Para aplicar este *script* a un fichero, tenemos que emplear `-f`:

```
# Aplicamos el script de AWK surnames al fichero sales.csv
awk -f surnames sales.csv | tail -n6

Sigismondo
Liuka
Tynan
Aylmer
Alexandr
OK. Final
```

2. Variables y argumentos

Se pueden tener variables con AWK, simplemente asignando un valor al nombre de la variable con el símbolo =. Además de las variables que se pueden establecer, existen unas cuantas variables predefinidas, que se emplean mucho, como, por ejemplo, `FS`, que ya hemos visto, que designa el separador de los campos.

Debemos estar muy atentos con el uso de **variables**, ya que cualquier variable no declarada previamente se inicializa en 0 (que equivale a falso) y como cadena vacía.

Con AWK todas las variables tienen un valor como cadena y como número. AWK elige uno de los dos valores en función del contexto. Las cadenas que no son numéricas (completamente), tienen un valor numérico de 0.

```
# Uso de una variable, france, no inicializada.
# Cuenta el número de líneas en que aparece la palabra France.
# ATENCIÓN: dado que france no está declarada, inicialmente vale 0.
awk '/France/ {france=france+1} END {print france}' sales.csv

9

# Uso de una variable, país, inicializada.
# Cuenta el número de líneas en que aparece la palabra France.
cat countries

# Cuenta el número de líneas en que aparece France.
BEGIN {pais=0}
/France/ {pais=pais+1}
END {print pais}

awk -f countries sales.csv

9
```

2.1. Variables del sistema

Como ya hemos dicho, hay varias variables que ya están predefinidas. `FS` sirve para especificar el separador. Este separador no hace falta que esté formado por un único carácter, sino que puede estar formado por un conjunto de caracteres. De hecho, puede ser una expresión regular.

```
echo "inicio -> derecha -> arriba -> derecha" | awk 'BEGIN {FS=" -> "} {print $2,$3,$4}'

derecha arriba derecha
```

Del mismo modo que hay un campo separador para la entrada, también hay uno para la salida, que es `OFS` por defecto, el espacio en blanco.

```
echo "inicio -> derecha -> arriba -> derecha" | awk 'BEGIN {FS=" -> "; OFS=", "} {print $2,$3,$4}'
```

```
derecha, arriba, derecha
```

Otra variable del sistema útil es `NF`, que corresponde al recuento de campos para la línea actual. Cambiarlo, sumando, crea más campos vacíos y disminuirlo hará desaparecer los campos más allá del último considerado.

```
awk 'BEGIN {FS=","} {print NF}' sales.csv |tail -n4

9
9
9
9
```

`RS` guarda el separador de líneas, que por defecto es `\n`, es decir, una nueva línea. Pero lo podríamos cambiar para tratar registros de más de una línea (o menos).

`NR` es el número de la línea, o mejor dicho, del registro actual.

```
awk 'BEGIN {FS=","} {print NR}' sales.csv |tail -n6

10
11
12
13
14
15
```

`FILENAME` contiene el nombre del fichero sobre el cual estamos trabajando.

```
awk 'END {print FILENAME}' sales.csv

sales.csv
```

Para controlar cómo se convierten los números en cadenas, AWK ofrece `CONVFMT` y se especifica con la función `printf` de C. Por defecto, AWK coge hasta seis valores decimales.

Función `printf` de C

Para más información, ved «Códigos de formato de printf/scanf».

La forma en que se muestran los números se controla con `OFMT`, también empleando la sintaxis de `printf` de C.

Por defecto, AWK considera que el separador decimal es el punto, independientemente de *locale*. Para que considere el separador propio del idioma del sistema, se puede emplear la opción `-N`.

```
# Cambiamos el valor OFMT para que muestre 3 dígitos después del separador decimal.
# total es un acumulador de la suma de los valores anteriores. En este caso, tendríamos que haber
saltado el primer registro.
awk 'BEGIN {FS=","; OFMT="%.3f"} {total+=$7; print total}' sales.csv

2017
2094.730
2104.950
2108.650
2187.040
2255.100
2291.260
2301.840
2382.590
```

```
2390.330
2422.300
2490.710
2497.630
2546.860
2596.070
```

2.2. Operaciones

Las operaciones matemáticas permitidas son las habituales:

Operador	Operación
+	Suma
–	Resta
*	Multiplicación
/	División
%	Módulo
^	Exponenciación
**	Exponenciación

Observad que la exponenciación se puede llevar a cabo con cualquiera de los dos operadores especificados en la tabla.

Por ejemplo:

```
awk 'BEGIN { a = 20; b = 3; print "(a ** b) = ", (a ** b) }'

(a ** b) = 8000

awk 'BEGIN { a = 117; b = 7; print "(a % b) = ", (a % b) }'

(a % b) = 5
```

También se pueden emplear **operadores de asignación**, muchos de ellos típicos en los lenguajes de programación similares a C:

Operador	Operación
++	Suma 1. Unaria.
--	Resta 1. Unaria.
+=	Asigna a la variable de la izquierda el resultado de sumarle el valor de la derecha.
-=	Asigna a la variable de la izquierda el resultado de restarle el valor de la derecha.
/=	Asigna a la variable de la izquierda el resultado de dividirla por el valor de la derecha.
%=	Asigna a la variable de la izquierda el resultado de aplicarle el módulo por el valor de la derecha.

Operador	Operación
^=	Asigna a la variable de la izquierda el resultado de elevarla al valor de la derecha.
**=	Asigna a la variable de la izquierda el resultado de elevarla al valor de la derecha.

Por ejemplo:

```
awk 'BEGIN { a = 117; b = 7; a%=b ;print "a = ", a }'

a = 5

awk 'BEGIN {a++; a++; a++ ;print "a = ", a }'

a = 3

awk 'BEGIN { a = 117; b = 7; a-=b ;print "a = ", a }'

a = 110
```

Finalmente, están también las **operaciones lógicas**, booleanas, muy útiles para el control de flujo, que veremos más adelante. Cuando una expresión es cierta, produce como resultado 1, y cuando no lo es, 0.

Operador	Operación
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
==	Igual a
!=	Diferente de
~	Coincide con la expresión regular
!~	No coincide con la expresión regular
||	O
&&	I
!	Negación

Algunos ejemplos para probar las expresiones lógicas de la tabla:

```
awk 'BEGIN {a=(6<8); print "a= ", a}'

a= 1

awk 'BEGIN {a=(6>8); print "a= ", a}'

a= 0
```

```
awk 'BEGIN {a=(6==8); print "a= ", a}'

a= 0

awk 'BEGIN {a=(6!=8); print "a= ", a}'

a= 1

awk 'BEGIN {a=("sur,este,norte"~".+,.+"); print "a= ", a}'

a= 1

awk 'BEGIN {a=("sur,este,norte"~".+;.+"); print "a= ", a}'

a= 0

awk 'BEGIN {a=(val==0 && val>=0 ); print "a= ", a}'

a= 1

# A vale 1 para los registros pares o para el registro 13.
awk '{a= (NR%2==0 || NR==13); print "NR= ",NR,"a= ", a}' sales.csv

NR= 1 a= 0
NR= 2 a= 1
NR= 3 a= 0
NR= 4 a= 1
NR= 5 a= 0
NR= 6 a= 1
NR= 7 a= 0
NR= 8 a= 1
NR= 9 a= 0
NR= 10 a= 1
NR= 11 a= 0
NR= 12 a= 1
NR= 13 a= 1
NR= 14 a= 1
NR= 15 a= 0
```

2.3. Argumentos y paso de parámetros

Podemos pasar parámetros a un *script* AWK poniendo el nombre de la variable, el símbolo igual y el valor que queremos darle. Esto se tiene que poner antes del nombre de los ficheros sobre los cuales queremos que actúe. Como ejemplo, veamos el *script* siguiente:

```
cat countries_var2

# Cuenta el número de líneas en que aparece la palabra especificada para nombre_pais

$0 ~ country_name {country=country+1} # Si en la línea aparece el nombre del país
END {print country_name,"has appeared",country,"times"}
```

```
# Llamamos el script pero le decimos que ya tenemos 10 apariciones anteriores
awk -f countries_var2 country=10 country_name=France sales.csv

France has appeared 19 times
```

Si hay que repetir muchas veces el mismo comando, podemos crear *scripts* AWK que se llamen desde Bash con el objeto de no tener que asignarles explícitamente los nombres de las variables (eso sí, el orden sí será importante). Es lo que se conoce como ***wrapper***, un envoltorio.

Conviene recordar que, con Bash, `$1` es el primer argumento indicado en el *script*, `$2` el segundo y así sucesivamente. Para hacer el envoltorio, no debemos confundir estas expresiones con los campos de AWK. Lo que hacemos en el *wrapper* es decirle que el primer argumento bash irá a `country` y el segundo a `country_name`:

```
# Muestra el wrapper de bash. Es una sola línea.
cat countries_wrapper

awk -f countries_var2 "country=$1" "country_name=$2" sales.csv

# Recordad dar los permisos de ejecución a countries_wrapper
# Se ejecuta el script con la primera variable, para country, igual a 20, y la segunda,
# para country_name, igual a France.
./countries_wrapper 20 France

France has appeared 29 times
```

Otra opción, si no tenemos que interactuar con la *shell*, es emplear el *shebang* `#!/usr/bin/awk -f` en el inicio del *script* y darle permisos de ejecución. Podemos indicarle los parámetros directamente.

```
cat countries_shebang

#!/usr/bin/awk -f
BEGIN {print "BEGIN. pais=",pais}
END {print "END. pais=",pais}

# Ejecuta el script y le indica el valor de país.
./countries_shebang pais="United Kingdom" sales.csv

BEGIN. pais=
END. pais= United Kingdom
```

Aquí podemos ver que uno de los problemas que se pueden presentar a la hora de gestionar los argumentos con AWK es que no están disponibles hasta que no se ha leído la primera línea del fichero que se va a procesar, es decir, no se pueden emplear en `BEGIN`. La solución pasa por emplear la **opción -v delante de cada parámetro** que indicamos.

```
#
./countries_shebang -v pais="United Kingdom" sales.csv

BEGIN. pais= United Kingdom
END. pais= United Kingdom
```

Para acabar este apartado, debemos recalcar que AWK, igual que C o Java, dispone de las variables `ARGC` y `ARGV`. Para entenderlas, hay que saber qué son los *arrays*.

Ved también

Trataremos los *arrays* en el apartado «*Arrays*» del presente módulo.

3. Control de flujo

El **control de flujo** hace referencia a la posibilidad de que algunas instrucciones solo se ejecuten si se da una determinada condición o se repite su ejecución mientras o hasta que se dé una condición.

Más información

Para más información, consultad un manual de programación estructurada.

El control de flujo con AWK está totalmente inspirado en el lenguaje C. Hay dos tipos de control de flujo: las **bifurcaciones** (el prototipo es `if-else`) y las **iteraciones** con `while`, `do-while` y `for`.

3.1. Bifurcación

En programación, no siempre nos interesa llevar a cabo todas las acciones, sino que en muchas ocasiones nos interesa solo si se cumple una o más condiciones. La primera manera de limitar es con los **patrones**, ya que AWK permite limitar los procedimientos que se van a ejecutar a solo la línea que cumpla el patrón.

```
# Muestra el número de línea que contiene uno de los países tratados.
cat bifurcacion_patron

/France/ {print "Linea", NR, "contiene France"}
/Spain/ {print "Linea", NR, "contiene Spain"}
```

```
awk -f bifurcacion_patron sales.csv

Linea 2 contiene France
Linea 4 contiene France
Linea 5 contiene France
Linea 7 contiene France
Linea 8 contiene Spain
Linea 9 contiene France
Linea 10 contiene Spain
Linea 11 contiene France
Linea 12 contiene France
Linea 13 contiene France
Linea 15 contiene France
```

Obsérvese que las acciones se ejecutan solo cuando se cumple el patrón.

Otra forma de ejecutar condicionalmente las acciones es con `if` y, opcionalmente, `else`. La acción se llevará a cabo solo si se cumple la condición. La sintaxis es:

```
if (expresión) { acción1 acción2 ...}
```

Con `else` se puede conseguir que si la expresión de `if` no es cierta, se lleven a cabo una serie de acciones alternativas:

```
  if (expresión) { acción1 acción2 ... } else
{ accion_alternativa1 accion_alternativa2 ... }
```

```
# Muestra los registros en los que las ventas del año 2018 han bajado respecto al 2017.
# Si el cuerpo solo consta de una instrucción y no sale else, no hacen falta las llaves.
awk 'BEGIN {FS=","} {if (NR!=1 && $7>$8) print $0}' sales.csv

6,Gill,Nelius,gnelius5@gravatar.com,Female,France,36.16,12.72,56.83
11,Liuka,Feedham,lfeedhama@bravesites.com,Female,France,68.41,49.67,55.89
13,Aylmer,Gabey,agabeyc@sphinn.com,Male,United Kingdom,49.23,19.6,74.39
14,Alexandr,Blackboro,ablackborod@spiegel.de,Male,France,49.21,47.91,93.8
```

Otra forma de ejecución opcional se hace con el operador condicional, que es el interrogante. La sintaxis es:

```
expresión ? acción1 : acción2
```

Si la expresión es cierta (vale 1), se ejecuta la primera acción, si es falsa (vale 0), se ejecuta la segunda. No es una forma recomendable si las expresiones no son muy simples, puesto que puede hacer que el código sea muy difícil de seguir.

```
# Muestra los registros en los que las ventas del año 2018 han bajado respecto al 2017.
# Si no han bajado, escribe un mensaje de información.
awk 'BEGIN {FS=","} {(NR!=1 && $7>$8) ? ms=$0 : ms="2018 more than 2017"; print ms}' sales.csv

2018 more than 2017
2018 more than 2017
2018 more than 2017
2018 more than 2017
2018 more than 2017
2018 more than 2017
6,Gill,Nelius,gnelius5@gravatar.com,Female,France,36.16,12.72,56.83
2018 more than 2017
2018 more than 2017
2018 more than 2017
2018 more than 2017
11,Liuka,Feedham,lfeedhama@bravesites.com,Female,France,68.41,49.67,55.89
2018 more than 2017
13,Aylmer,Gabey,agabeyc@sphinn.com,Male,United Kingdom,49.23,19.6,74.39
14,Alexandr,Blackboro,ablackborod@spiegel.de,Male,France,49.21,47.91,93.8
```

3.2. Iteraciones

Las **iteraciones** sirven para repetir una o más acciones según una determinada condición.

AWK ya itera automáticamente sobre cada registro, típicamente sobre cada línea, pero esto no siempre es suficiente y, por tanto, se puede emplear `while`, `do-while` y `for`.

```
# While analiza una condición y mientras sea cierta se ejecutan las acciones.
# Hay que observar que si la condición no se cumple desde el principio, no se ejecutará ninguna acción.
cat while

BEGIN {
while (i<4){
    print i
    i++
```

```
    }
}

awk -f while

1
2
3

# Do-while ejecuta una acción y mientras la condición sea cierta, la sigue ejecutando.
# Es muy similar a la anterior, pero, en este caso, las acciones se ejecutarán una vez al menos.
cat do_while

BEGIN {
do{
    print i
    i++
    }
while (i<4)
}

awk -f do_while

1
2
3
```

Los `for` sirven para iterar un número determinado de veces. Cuando se usa `for`, se inicia una variable, la condición que marca cuándo se tiene que dejar de ejecutar el cuerpo del bucle, y el incremento o decremento que se tiene que hacer después de cada iteración.

```
echo "Today is a beautiful day" | awk '{for (i=1; i<=NF; i++) print $i}'

Today
is
a
beautiful
day
```

4. *Arrays*

Así como en una variable podemos guardar un valor, en un *array* podemos guardar muchos, aunque, desde un punto de vista lógico, tienen que tener alguna relación entre ellos. Cada elemento del *array* se asigna a su índice, que con AWK se pone entre corchetes. Así, si tenemos el *array* `año2018`, `año2018[1]` es el valor del elemento 1 del *array*.

AWK permite ***arrays* asociativos**, en los que el índice no es un número, sino una cadena. Es más, para AWK todos los índices son cadenas, es decir, `año2018[1]` es lo mismo que `año2018["1"]`.

```
awk 'BEGIN {a[1]=23; print a["1"]}'

23
```

```
# Suma todas las ventas del año 2018. Guarda todos los valores de 2018 (campo 7) en un array
# y después lo recorre sumando sus valores.
cat array2018

# Suma todas las ventas del año 2018.

BEGIN {FS=","} # Establecemos el separador a ,

# Excepto por el primer registro,
# el 7.º campo (correspondiente a 2018) se
# guarda en un array llamado year2018.
{if (NR!=1) year2018[NR-1]=$7}

# Recorre el array y se suman sus elementos.
END{

    for (i=0;i Para<NR;i++){
        sum+=year2018[i]
    }
    print sum

}

awk -f array2018 sales.csv

579.07
```

Para crear *arrays*, con AWK tenemos la función `split()` que es de gran utilidad. Dicha función admite tres argumentos:

- la cadena original que se va a separar,
- el nombre del *array* y
- el separador.

Retorna la longitud del *array* generado, no el *array*.

```
# Cogemos toda la línea, separamos por / y lo guardamos en date.
echo "01/07/2019" | awk '{n=split($0,date,"/"); print n,"campos: día",date[1],"de",date[2],
"del",date[3]}'
```

```
3 campos: día 01 de 07 de 2019
```

```
# Consideramos solo el primer campo, separamos por guion y lo guardamos en date.
echo "06-08-2020, coche, 15.620€" | awk 'BEGIN {FS=","}
{n=split($1,date,"-"); print n,"campos: día",date[1],"de",date[2],"del",date[3]} '

3 campos: día 06 de 08 de 2020
```

Los elementos de un *array* se pueden borrar con `delete`.

```
# Consideramos solo el primer campo, separamos por guion y lo guardamos en date.
# Entonces borramos el primer valor.
echo "06-08-2020, coche, 15.620€" | awk 'BEGIN {FS=","}
{n=split($1,date,"-");delete date[1];print n,"campos: día",date[1],"de",date[2],"del",date[3]} '

3 campos: día de 08 de 2020
```

Con `in` podemos saber si un valor es un **índice** en un *array*:

```
awk 'BEGIN {a[1]="a1"; a[2]="a2"; print ("1" in a); print (1 in a); print (3 in a)}'

1
1
0
```

Hay dos variables del sistema que son *arrays*: `ARGV` y `ENVIRON`. El primero guarda los argumentos con que se ha llamado AWK y el segundo guarda las variables del sistema (que se pueden consultar introduciendo `env` en la *shell*).

```
# Muestra el valor de DISPLAY del sistema.
awk 'BEGIN {print ENVIRON["DISPLAY"]}'

:0.0
```

Así como en `ARGV` están los valores indicados como argumentos, en `ARGC` está el número de argumentos indicados. `ARGV[0]` equivale al `"awk"` mismo. Por ejemplo:

```
awk 'BEGIN {
 for (i = 0; y ARGC < ; ++i) {
 print "ARGV ", i, " = ", ARGV[i]
 }
}' argu1 argu2 argu3

ARGV 0 = awk
ARGV 1 = argu1
ARGV 2 = argu2
ARGV 3 = argu3
```

Cuando ejecutamos AWK desde un fichero, `ARGV` ignora `-f` y el nombre del fichero.

5. Funciones

5.1. Funciones predefinidas

AWK consta de varias funciones predefinidas. Las **numéricas** son:

Funciones	Operaciones
`cos(x)`	Devuelve el coseno de x.
`sin(x)`	Devuelve el seno de x.
`atan2(y,x)`	Devuelve la arcotangente de y/x.
`exp(x)`	Devuelve e elevado a x.
`int(x)`	Devuelve x con entero truncado.
`log(x)`	Devuelve el logaritmo, con base e, de x.
`sqrt(x)`	Devuelve la raíz cuadrada de x.
`rand()`	Devuelve un número pseudoaleatorio del intervalo [0,1).
`srand(x)`	Establece una semilla para números aleatorios. Si no se especifica, se emplea la hora, día y año actuales.

Por ejemplo:

```
# Define la semilla y elige dos números aleatorios, que estarán entre 0 y 1.
# Muestra los números y su suma.
awk 'BEGIN {srand(10); x=rand(); y=rand(); print "x=",x,", y=",y,", suma=",x+y}'

x= 0.255219 , y= 0.898883 , suma= 1.1541
```

A la hora de elegir los número aleatorios, no siempre los queremos del intervalo por defecto. Entonces habría que hacer una proyección sobre el intervalo deseado. Por ejemplo, suponemos que queremos números aleatorios entre -3, incluido, y 5, excluido. Entonces, dado que el nuevo intervalo tiene una longitud de 8, tenemos que multiplicar por 8 y, al resultado, restarle 3, que es el origen.

```
# Elige números aleatorios de entre 0 y 1 y los proyecta sobre [-3,5).
awk 'BEGIN {srand(10); x=rand(); y=rand(); newx= x*8 -3 ; newy= y*8 -3; print "x=",x,",
newx=",newx"\ny=",y,", newy=",newy}'

x= 0.255219 , newx= -0.95825
y= 0.898883 , newy= 4.19106
```

Para **manipular cadenas**, tenemos:

Cadenas	Operaciones
`gsub(r,s,t)`	Sustituye s por cada aparición de la expresión regular r en t. Si no se especifica t, se emplea `$0`. Retorna el número de sustituciones hechas.
`sub(s,p,t)`	Igual que `gsub` pero solo sustituye la primera ocurrencia.
`index(s,t)`	Retorna la posición de t en s o cero si no existe.
`lenght(s)`	Retorna la longitud de s o de `$0` si no se especifica s.
`match(s,r)`	Retorna la posición de s, donde empieza la expresión regular r, o 0 si no existe. Esta función establece el valor de dos variables `RSTART` y `RLENGTH` que están donde empieza la coincidencia (igual que el valor retornado) y la longitud.
`split(s,a,sep)`	Construye un *array* a, a partir de s, empleando el separador `sep`. Si `sep` no se explicita, se emplea el valor de `FS`.
`printf(cadena,expr)`	Funciona del mismo modo que la función homónima de C y otros lenguajes. Se pone una cadena de caracteres con unos símbolos especiales donde irán los valores de las variables. Por ejemplo, `sprintf("Hola %s", user)`, mostrará `"Hola Pepito"` si la variable `user` es igual a «Pepito».
`substr(s,p,n)`	Retorna la subcadena que va desde la posición p hasta la n. Si n no se especifica, retorna la subcadena hasta el final.
`tolower(s)` `toupper(s)`	Retorna una cadena todo en minúsculas o todo en mayúsculas.

```
# Muestra la primera posición de "AG".
echo "ATAGGCTAGTAA" | awk '{pos=index($0,"AG"); print pos}'

3

# Sustituye "AG" por "AA".
echo "ATAGGCTAGTAA" | awk '{print $0; mutation=gsub("AG","AA"); print $0}'

ATAGGCTAGTAA
ATAAGCTAATAA

# Muestra dónde empieza la coincidencia con el patrón ".CTA" y su longitud.
echo "ATAGGCTAGTAA" | awk '{pos=match($0,".CTA"); print pos, RLENGTH}'

5 4

# Muestra dónde empieza y acaba la coincidencia con el patrón ".CTA".
echo "ATAGGCTAGTAA" | awk '{pos=match($0,".CTA"); printf("Desde %d hasta %d", pos,pos+RLENGTH)}'

Desde 5 hasta 9

# Muestra el fragmento que coincide con el patrón ".CTA".
echo "ATAGGCTAGTAA" | awk '{pos=match($0,".CTA"); print substr($0,pos,RLENGTH)}'

GCTA


# Transforma todos los caracteres a minúsculas.
echo "ATAGGCTAGTAA" | awk '{print tolower($0)}'

ataggctagtaa
```

5.2. Creación de funciones

La creación de funciones propias sigue la sintaxis de C. Se empieza con la palabra reservada `function`, el nombre y después los parámetros (opcionales) entre paréntesis. El cuerpo de la función se pone entre llaves. Con `return` (opcional) podemos hacer que la función retorne un determinado valor.

```
# En este fichero hay definidas dos funciones que traducen dos palabras del inglés
al castellano o al catalán.
cat function

# Script que traduce unas determinadas palabras del inglés al castellano y catalán.

# Traduce al castellano.
function spanish(moment){
    result="ERROR Moment of the day unknown"
    if (moment=="day") {result="día"}
    if (moment=="night") result="noche"
    return result
}
# Traduce al catalán.
function catalan(moment){
    if (moment=="day") {return "dia"}
    if (moment=="night") return "nit"
    return "ERROR Moment of the day unknown"
}

# Main
{
sp=spanish($0)
ca=catalan($0)
printf ("%s in spanish is %s and in catalan, %s\n",$0,sp,ca)
}

# Fichero donde se aplicará el script anterior.
cat moments.txt

day
night
afternoon

# Resultado de aplicar el script awk function a moments.txt
awk -f function momentos.txt

day in spanish is día and in catalan, dia
night in spanish is noche and in catalan, nit
afternoon in spanish is ERROR Moment of the day unknown and in catalan, ERROR Moment of the day unknown
```

6. Lectura de ficheros con `getline`

El comando `getline`, sin ningún otro valor, hace que se procese el registro siguiente y se dejen de ejecutar las acciones posteriores a `getline`. Por ejemplo:

```
# Cada vez que el primer campo es impar, se ejecuta getline y, por lo tanto, no ocurre nada.
# En caso contrario, sí que se ejecuta el print.
awk 'BEGIN {FS=","} {if ($1%2==1) { getline}; print $0}' sales.csv

id,first_name,last_name,email,gender,country,2017,2018,2019
2,Vivianne,Feeney,vfeeney1@washingtonpost.com,Female,United Kingdom,10.22,51.01,49.88
4,Wakefield,Shepton,wshepton3@sphinn.com,Male,France,78.39,86.02,86.35
6,Gill,Nelius,gnelius5@gravatar.com,Female,France,36.16,12.72,56.83
8,Moss,Avo,mavo7@merriam-webster.com,Male,France,80.75,81.27,93.91
10,Sigismondo,Winterborne,swinterborne9@sciencedirect.com,Male,France,31.97,71.87,32.91
12,Tynan,Coen,tcoenb@businesswire.com,Male,France,6.92,10.44,34.3
14,Alexandr,Blackboro,ablackborod@spiegel.de,Male,France,49.21,47.91,93.8
```

`Getline` también se utiliza **para leer otros ficheros**, además del que se aplica al propio *script*.

La sintaxis es `getline nombre_variable < "nombre_del_fichero"`. En un ejemplo anterior, hemos visto cómo podíamos traducir palabras concretas. Está claro que el método empleado no es escalable, puesto que las palabras equivalentes las tenemos en el mismo código de las funciones. Con la ayuda de `getline` podemos crear un fichero con parejas de palabras, leerlo desde AWK y hacer la traducción oportuna.

```
# Fichero donde están parejas de palabras.
cat en-es.txt

day, día
night, noche
afternoon, tarde
morning, mañana
midday, mediodía
midnight, medianoche

# Script que busca una palabra en un fichero y retorna su pareja.
cat translator

# Traduce unas determinadas palabras del inglés al castellano.

# Función que traduce MOMENT al castellano.
# Las parejas de palabras se guardan en un fichero.
function translate(MOMENT){
    res="NOT FOUND" # Valor si no existe la palabra
    while (getline words < "en-es.txt"){ # Lee el fichero línea a línea
        split(words,word) # Pone los tokens en un array llamado "word"
        if(word[1]==MOMENT) res=word[2] # Si el primero es igual al parámetro, hecho
    }
    close("en-es.txt") # Cierra el fichero
    return res
}

# MAIN
{
```

```
print $0, translate($0)
}

awk -f translator moments.txt

day día
night noche
afternoon mediodía
```

7. El «problema» de los separadores

En muchos ficheros csv, hay campos que contienen comas en su valor, es decir, **comas que no son separadores de campos**, sino que forman parte del propio campo. Por ejemplo, si un campo tiene la frase «uno, dos, tres», se ha de interpretar como un único campo. Muchos programas lo que hacen es poner entre comillas todos los campos de texto para evitar ambigüedades (y en los campos numéricos emplear el punto como separador decimal). Observad el fichero csv siguiente:

```
cat separators.csv

3,"a good number, prime one",3b
1,"a really boring number",1b
2,"a couple is always a beautiful choice",2b
```

Con lo que hemos visto hasta ahora, si quisiéramos reordenar los campos, podríamos intentarlo con:

```
awk 'BEGIN {FS=","} {print $3 " " $2,":",$1}' separators.csv

prime one" "a good number : 3
1b "a really boring number" : 1
2b "a couple is always a beautiful choice" : 2
```

Como ya hemos explicado antes, este resultado no debería sorprendernos. El segundo campo se ha partido en dos, puesto que hay una coma.

La solución a este problema pasa por cambiar la manera en que se separan los campos. Recordad que con `FS` elegimos el separador de los campos. Pero ahora lo que debemos hacer, en lugar de pensar en el separador, es crear una descripción, en forma de expresión regular, de cómo son los campos y emplear la variable del sistema `FPAT`. Los campos que tenemos ahora son de dos formas:

- Contenido entre comillas, formado por una doble comilla, elementos que no son doble comilla y doble comilla. La expresión regular correspondiente es `\"[^\"]+\"`.

- Contenido que no tiene comas, que es `[^,]+`.

Por tanto, la expresión regular que define cómo son nuestros campos es una o la otra:

```
awk 'BEGIN {FPAT="(\"[^\"]+\")|([^,]+)"} {print $3 " " $2,":",$1}' separators.csv

3b "a good number, prime one" : 3
1b "a really boring number" : 1
2b "a couple is always a beautiful choice" : 2
```

8. Formatear la salida

La intención original de los creadores de AWK fue crear una herramienta para hacer informes de forma automática.

Con muy poco esfuerzo (y conocimiento de `printf`) se pueden hacer documentos con una presentación bastante útil.

`printf`, a diferencia de `print`, no crea una nueva línea automáticamente después de su salida:

```
awk 'BEGIN {print "one"; print "two"}'

one
two

awk 'BEGIN {printf "one"; printf "two"}'

onetwo
# Para hacer una nueva línea, con GNU/Linux, se emplea \n
awk 'BEGIN {printf "one\n"; printf "two"}'

one
two
```

printf/scanf

Podéis encontrar un listado de los símbolos en «Códigos de formato de printf/scanf».

La gran utilidad que tiene `printf` es que podemos escribir unos símbolos donde queremos que aparezca el valor de una variable, haciendo así la escritura más cómoda y fácil de mantener. Donde tiene que aparecer el valor que queremos, se escribe el símbolo del tanto por ciento y una letra que indica el tipo de valor que aparecerá: entero (d o i), cadena (s), etc.

```
awk 'BEGIN {product="towels"; units="4"; printf("We need %d %s",units,product)}'

We need 4 towels
```

Además de los símbolos, también se puede especificar la anchura y la alineación, entre el % y el símbolo del tipo de valor. La anchura se establece poniendo el número de caracteres que se quiere para las cadenas. Para limitar el número de decimales, se escribe un punto justo después del % y antes del símbolo, por ejemplo:

```
awk 'BEGIN {units="4"; unit_price="3.987102"; printf("%d units cost %.2f",units,units*unit_price)}'

4 units cost 15.95
```

Por defecto, la alineación es a la derecha. Para cambiarlo, se pone un símbolo menos (-) después del porcentaje:

```
awk 'BEGIN {printf("Today I will see %-15s. ","John")}'

Today Y will see John .
```

```
awk 'BEGIN {printf("Today I will see %15s. ","John")}'

Today I will see                                                                John.
```

8.1. El informe de ventas

Para hacernos una idea de cómo hacer un informe con AWK, partiremos del fichero que hemos ido usando y crearemos un documento para poder analizar mejor la información que tenemos, mostrando la que nos interesa.

Haremos una tabla con el nombre del cliente, lo que ha gastado cada año y la diferencia con el año anterior. Después, haremos las sumas por año.

En el *script* hay dos funciones: una que imprime la cabecera y otra que hace los cálculos apropiados y muestra la información de un cliente.

```
# Script awk para hacer el informe.
cat informe

# Imprime la cabecera.
function print_head(){
    printf "\n\t\t\tSALES EVOLUTION 2017-2019\n\n"
    printf "Customer\t\t2017\t2018\tDif 2017\t2019\tDif 2018\n"
    printf "-------------------------------------------------"
    printf "----------------------\n\n"
}

# Crea cada línea a partir del número y las ventas por año.
function process_customer(NAME, s2017, s2018, s2019){
    CUSTOMER_LENGTH=16 # Constante específica para el número de líneas en la salida.

    dif2017=s2018-s2017 # Diferencia entre las ventas del año 2018 y 2017.
    difp2017=(dif2017/s2017)*100 # Diferencia anterior, en porcentaje.

    dif2018=s2019-s2018 # Diferencia entre las ventas de 2019 y 2018.
    difp2018=(dif2018/s2018)*100 # Diferencia anterior, en porcentaje.

    # name, 2 2017, 3 2018, 4 dif2017, 5 2019, 6 dif 2019
    # Pone uno o dos tabuladores según la longitud del nombre.
    if (length(NAME)>CUSTOMER_LENGTH) printf ("%s\t",NAME)
    else printf ("%s\t\t",NAME)
    printf ("%.2f\t%.2f\t%.2f(%d%%)\t%.2f\t%.2f(%d%%)\n",s2017,s2018,dif2017,difp2017,s2019,
    dif2018,difp2018 )
}

BEGIN {FS=","; print_head()}

# MAIN
{
    if (NR!=1) { # Si no es la primera línea
        name=$2 " " $3
        process_customer(name,$7,$8,$9)
        t2017+=$7
        t2018+=$8
        t2019+=$9
    }


}

END {
    printf "-------------------------------------------------"
    printf "----------------------\n"
    process_customer("Total:\t ",t2017,t2018,t2019) # NAME incluye un tabulador, por temas
```

```
    de presentación.
}
```

```
awk -f informe sales.csv

                              SALES EVOLUTION 2017-2019

Customer                   2017     2018    Dif 2017        2019    Dif 2018
-------------------------------------------------------------------------------------------

Lulu Chaloner             77.73    85.90    8.17(10%)      60.46    -25.44(-29%)
Vivianne Feeney           10.22    51.01    40.79(399%)    49.88    -1.13(-2%)
Waring Craythorne          3.70    35.21    31.51(851%)    72.35    37.14(105%)
Wakefield Shepton         78.39    86.02    7.63(9%)       86.35    0.33(0%)
Beale Myrtle              68.06    77.78    9.72(14%)      40.26    -37.52(-48%)
Gill Nelius               36.16    12.72    -23.44(-64%)   56.83    44.11(346%)
Kippy Marlen              10.58    69.81    59.23(559%)    50.30    -19.51(-27%)
Moss Avo                  80.75    81.27    0.52(0%)       93.91    12.64(15%)
Lona Garrison              7.74    74.90    67.16(867%)    14.06    -60.84(-81%)
Sigismondo Winterborne    31.97    71.87    39.90(124%)    32.91    -38.96(-54%)
Liuka Feedham             68.41    49.67    -18.74(-27%)   55.89    6.22(12%)
Tynan Coen                 6.92    10.44    3.52(50%)      34.30    23.86(228%)
Aylmer Gabey              49.23    19.60    -29.63(-60%)   74.39    54.79(279%)
Alexandr Blackboro        49.21    47.91    -1.30(-2%)     93.80    45.89(95%)
-------------------------------------------------------------------------------
Total:                   579.07   774.11    195.04(33%)    815.69    41.58(5%)
```

Bibliografía

Bibliografía básica

Dougherty, D.; Robbins, A. (1997). *Sed & Awk*. Massachusetts: O'Reilly Media.

Robbins, A. (2015). *Effective AWK Programming: Universal Text Processing and Pattern Matching* (4.ª ed.). Massachusetts: O'Reilly Media.

Vidal Cortés, J. A. (2002). *El Lenguaje de Programación AWK/GAWK*. Madrid.

Referencias

Es.cppreference.com (2012). «Códigos de formateo de printf/scanf». cppreference.com. Disponible en: https://es.cppreference.com/w/cpp/io/c/printf_format.