Sistemas Informáticos

1º de DAM/DAW

Tema 7 - Anexo

Introducción a AWK

IES San Vicente

Curso 2020/2021

Índice de contenido

1. Uso básico de AWK

1.1 Introducción

El programa **awk** más que un simple comando, es un intérprete de un lenguaje de programación llamado de la misma manera (con una sintaxis basada en C, Java, …). Al ser tan potente, se puede hacer lo mismo y mucho más que con comandos tipo grep, sed, cut, tr, etc.

Awk está pensado sobre todo para procesar ficheros y texto. Sobre todo tiene una gran flexibilidad a la hora de trabajar con ficheros de texto que contienen campos separados por algún tipo de patrón (como por ejemplo el punto y coma ';'). Estos archivos pueden ser, listas de usuarios, o de cualquier tipo de elemento, pequeñas bases de datos en formato texto, listas de correo, etc.

Este comando se puede utilizar para mostrar uno o varios campos de un fichero (tipo cut pero con más opciones), para contar las líneas (tipo wc), para buscar patrones (grep o egrep), y muchas más cosas. En esencia, se trata de un lenguaje de programación con su propia sintaxis (muy sencilla para los programadores familiarizados con la programación estructurada clásica), variables, y estructuras de control (if, for, while).

1.2 Estructura básica

Las instrucciones que daremos al comand awk se pueden incluir (entre comillas simples) como parámetro del comando → awk -F ';' '**{print $1}**' archivo.txt, o bien, si es una estructura de cierta longitud, guardarla en un archivo con extensión **.awk** (las instrucciones ya no irían entre comillas) y ejecutarlo de la siguiente manera utilizando la opción -f para indicar el archivo con el código → **awk -F ';' -f programa.awk archivo.txt.**

Un programa awk está compuesto de bloques delimitados entre llaves **{}**. Cada bloque está precedido (no obligatoriamente) de un patrón que indica cuando se ejecutará, y de una serie de acciones o instrucciones:

patrón { acciones }

Como hemos dicho, awk es un lenguaje especializado en procesar archivos o entradas de texto, y lo hace línea a línea. Los patrones suelen indicar las condiciones que debe cumplir una línea para ser procesada por las acciones del bloque correspondiente. Los tipos de patrones pueden ser:

- **Vacío** → Aplica las acciones del bloque a cada una de las líneas del archivo.
- **BEGIN** → Patrón especial que se ejecuta antes de empezar a procesar el archivo. Puede usarse para inicializar variables, o imprimir una salida previa, por ejemplo.
- **END** → Patrón especial que se ejecuta después de terminar de procesar el archivo. Se puede utilizar para imprimir un resultado final del procesamiento.
- /**expresión regular**/ → Procesa sólo las líneas que cumplen con la expresión regular.
- **Condición booleana** → Procesa sólo las líneas que cumplen con la condición. Una condición puede ser que el tercer campo sea mayor que 500 → **$3 > 500**. También

se puede comprobar si un campo cumple una expresión regular con el operador '~' → **$1 ~ /expresión/**.

Ejemplo

Tenemos un archivo llamado **pedido.txt** que representa a un pedido con la siguiente estructura (referencia, categoría, producto, precio unidad, y unidades):

```
B5361;Componentes;Procesador;145.65;2
A4315;Componentes;Placa base;109.95;2
A0043;Periféricos;Memoria USB 32GB;19.50;6
B0435;Componentes;Disco duro 1TB;54.15;4
B1324;Periféricos;Impresora;67.89;1
C4359;Redes;Router 802.11n;34.90;1
B4251;Redes;Cable Ethernet 3m;3.95;15
```

Si el objetivo fuera sumar el precio total de los productos de la categoría componentes el código awk necesario sería el siguiente (lo guardamos en **ejemplo1.awk**):

```
BEGIN {
    FS = ";"
    total = 0
}
$2 ~ /Componentes/ {
    total += $4 * $5
}
END {
    print "Total componentes: "total"€"
}
```

Al ejecutar dicho código con **awk -f ejemplo1.awk pedido.txt** obtenemos la siguiente salida:

```
Total componentes: 727.8€
```

Vamos a analizar lo que hace el código arriba escrito:

1. Antes de empezar se ejecuta el bloque BEGIN. Se establece la variable FS (separador de campos) al valor ';' (equivale a usar la opción -F ';' con el comando awk). También inicializamos una variable que guardará el precio total a 0.

2. Seguidamente para cada línea del fichero, se filtrarán las que su segundo campo $2 (categoría) cumplan con la expresión /Componentes/ (Vamos, que su valor sea ese). **Sólo** en esas líneas se acumulará en la variable total el resultado de multiplicar el campo $4 (precio unidad) por $5 (cantidad). Si queremos procesar todas las líneas basta con quitar la comparación del principio y no poner nada.

3. Finalmente se ejecuta el bloque END, donde imprimimos por pantalla el valor de la variable total. En **awk** (al menos la implementación incluida en Linux y que se llama gawk) las variables no van precedidas de $.

Los comentarios en awk son de 1 línea y van precedidos de almohadilla '#'.

1.3 Imprimir por pantalla

print

La instrucción **print** imprime una cadena de texto por pantalla. El texto se debe expresar entre comillas dobles, mientras que las variables o resultados de operaciones deben de ir fuera de las comillas. No existe un carácter de concatenación, así que el texto y las variables deben de ir una al lado de la otra, o separadas por coma (la coma introduce un espacio en blanco adicional en la salida).

```
{print "Valor: "$1} ↔ {print "Valor:", $1}
```

La operaciones, tanto aritméticas como lógicas es recomendable escribirlas entre paréntesis, así los operadores de comparación (< o >) por ejemplo, no se confunden con operaciones de redireccionamiento. La salida de una operación lógica es 0 (false) o 1 (true).

```
awk '{print $1" mayor que 100?: "($1 > 100)}'
```

Separadores de salida

Por defecto, print incluye un salto de línea después de imprimir una salida, y al separar los diferentes elementos por coma ‘,’, hemos comentado que añade un espacio en blanco.

Este comportamiento se puede modificar, cambiando el valor de las variables OFS (separador que añade la coma, por defecto espacio) y ORS (Lo que se añade al final de la línea que imprime, por defecto salto de línea → \n). Ejemplo (**ejemplo2.awk**):

```
BEGIN {
    FS = ";" # Separador de campos en el fichero (pedido.txt)
    ORS = "\n--------\n" # Después de cada print se añadirá esto
    OFS = "-" # La coma añadirá un guion en print
}

{ print "producto ",$3," Total: "($4 * $5) }
```

Salida:

```
producto -Procesador- Total: 291.3
--------
producto -Placa base- Total: 219.9
--------
producto -Memoria USB 32GB- Total: 117
--------
...
```

printf

Esta instrucción está disponible en la mayoría de lenguajes de programación (C, Java, PHP, Bash, …), y es una versión más potente de print. Se utiliza para imprimir una salida formateada y por ejemplo, hacer que una cadena ocupe un ancho fijo, o un número no entero aparezca con una cantidad de decimales fija. Las variables anteriores (ORS y OFS) no tienen efecto aquí, ya que no se añade nada al final de cada línea (el salto de línea lo debemos incorporar nosotros), ni la coma significa lo mismo.

La instrucción **printf** consta de una cadena que describe un formato seguido de varios

elementos separados por coma. Dichos elementos, que pueden ser cadenas de texto o variables, se incluirán en la salida final según lo que indique la cadena de formato.

```
BEGIN {
    FS = ";"
    print "Producto           Cantidad   Precio U.  Total"
    print "-------------------------------------------"
}

{ printf "%s  %d  %.2f€ %.2f€\n", $3, $5, $4, $5*$4 }
```

Estas instrucciones (**ejemplo3.awk**) aplicadas sobre **pedidos.txt** producen la siguiente salida:

```
Producto           Cantidad   Precio U.  Total
-------------------------------------------
Procesador  2  145.65€ 291.30€
Placa base  2  109.95€ 219.90€
Memoria USB 32GB  6  19.50€ 117.00€
Disco duro 1TB  4  54.15€ 216.60€
Impresora  1  67.89€ 67.89€
Router 802.11n  1  34.90€ 34.90€
Cable Ethernet 3m  15  3.95€ 59.25€
```

Como se puede observar, los símbolos %s, %d y %.2f se sustituyen en orden por cada uno de los elementos o campos que se incluyen a la derecha separados por comas. Sin embargo, aparte de que %.2f obliga a que aparezcan 2 decimales esta salida no está bien estructura, lo ideal es que los campos ocupen un ancho fijo. Esto se soluciona cambiando el formato de printf así (**ejemplo4.awk**):

```
{ printf "%-18s %8d %10.2f€ %8.2f€\n", $3, $5, $4, $5*$4 }
```

Esto produce la siguiente salida:

```
Producto           Cantidad   Precio U.    Total
-------------------------------------------------
Procesador                2     145.65€   291.30€
Placa base                2     109.95€   219.90€
Memoria USB 32GB          6      19.50€   117.00€
Disco duro 1TB            4      54.15€   216.60€
Impresora                 1      67.89€    67.89€
Router 802.11n            1      34.90€    34.90€
Cable Ethernet 3m        15       3.95€    59.25€
```

Esto es lo que significa cada símbolo en **printf**:

- **%s** → Se sustituye por un string.
  - **%20s** → Si el string tiene menos de 20 caracteres, te rellena por la izquierda con espacios. Si tiene más te lo muestra todo.
  - **%.20s** →Como el anterior pero limita como máximo a 20 caracteres.
  - **%20.12s** → Te muestra 12 caracteres rellenando 8 espacios a la izquierda (20)
  - **%-20s y %-20.12s** → Rellena con espacios por la derecha.
- **%d** → Número entero.

- **%3d** → Si el número tiene menos de 3 cifras, rellena con espacios a la izquierda
- **%.3d** → Si el número tiene menos de 3 cifras, rellena con 0 a la izquierda.

- **%f** → Número decimal.
  - **%8.2f** → Muestra el número con 2 decimales, y si (contando el separador y los decimales) ocupa menos de 8 caracteres, rellena con espacios a la izquierda.
  - **%.3f** →Muestra el número con un formato fijo de 3 decimales.

http://es.wikipedia.org/wiki/Printf

Cabe destacar que la salida de print y printf se puede redireccionar a un archivo con > y >> en lugar de mostrarla por pantalla, o pasársela a otro comando mediante una tubería '|' para que la procese, de la misma forma que se hace en la línea de comandos de bash.

1.4 Variables

Las variables se pueden inicializar en cualquier momento del programa. Estas variables no tienen tipo ni necesitan ninguna palabra delante para declararlas, simplemente darles valor, que puede ser una cadena (entre comillas) o numérico.

variable = valor

En este caso, no se utiliza el dólar precediendo a los nombres de variables (como en PHP o Bash) en ningún caso (asignar o imprimir valor). Lo único a lo que precede el dólar es a las variables especiales como $0 (línea/registro completa/o), $1 → primer campo, $2 → segundo campo,etc… Por ejemplo:

```
BEGIN { v = 1 }
# Imprime el número 1 y a continuación el valor del campo 1 ($1).
{ print v, $v }
```

Se puede inicializar una variable al ejecutar el comando awk, bien con la opción -v variable="valor" (sin espacios entre '=') lo que inicializa la variable antes de empezar a hacer nada, o directamente sin la opción -v. En este último caso, se puede asignar variables o valores diferentes antes de cada fichero que se procese (se pueden procesar varios en orden):

```
awk -f programa.awk n=1 arch1.txt n=2 arch2.txt
```

Variables implícitas

Algunas variables implícitas del lenguaje son las siguientes:

- **FS** → Separador de campos de entrada. Equivale a usar -F al ejecutar awk.
- **IGNORECASE** → Valores 1 (ignora minúsculas/mayúsculas en las expresiones de búsqueda), o 0 (valor por defecto).
- **OFS** → Separador de campos de salida. Su valor sustituye a la coma en la instrucción **print** (por defecto es un espacio).
- **ORS** → Separador de registros de salida. Su valor se imprime al final de la instrucción **print** (por defecto es un salto de línea).
- **RS** → Separador de registros. Es el valor que utiliza awk para separar los registros que procesa internamente. Por defecto procesa línea a línea. Si establecemos por

ejemplo **RS=";"**, procesa cada campo separado por punto y coma como si estuvieran en líneas diferentes.

- **FILENAME** → Nombre del fichero que se está procesando actualmente.
- **ENVIRON** → Array asociativo con las variables de entorno. **ENVIRON["HOME"]** equivale a usar **$HOME** en la línea de comandos.
- **FNR** → Número de registro que se está procesando en el archivo actual. Parecido a la variable **i** si se recorrieran las líneas con un for en otro lenguaje.
- **NR** → Número de registro total desde el comienzo de la ejecución. Si sólo se procesa un archivo, es totalmente equivalente a **FNR**.
- **NF** → Número de campos que contiene el registro (o línea) actual.

1.5 Operadores aritméticos

Las operaciones aritméticas en awk son las comunes: suma '**+**', resta '**-**', multiplicación '*****', división '/', resto '**%**' y potencia '**^**' o '******'. Internamente todos los números decimales (coma flotante de doble precisión o 64 bits), por lo que los resultados con decimales se representarán correctamente. Obviamente, se pueden utilizar paréntesis para agrupar expresiones, e incluso está recomendado en las operaciones simples dentro de **print**.

1.6 Operadores lógicos

Awk también soporta operaciones lógicas, tanto de comparación (<, >, <=, >=, ==, !=), incluyendo también **valor** ~ /**exp**/ → el valor cumple con la expresión regular, y **valor !**~ /**exp**/ → El valor no cumple con la expresión regular.

También incluimos la concatenación lógica || (o), y **&&** (y), además de la negación **!**. Este tipo de expresiones lógicas se pueden utilizar tanto como condiciones de estructuras de control → if ($1 < 10 && $2 ~ /[0-9]{5}[a-z]{3}/), como también para establecer patrones sobre las líneas a procesar:

```
NR > 5 && $3 !~ /[0-9]+/ { instrucciones }
```

1.7 Conversión cadena-número

La operación de suma '+' convierte las cadenas a números antes de realizarse. Por ejemplo sumar 0 a una cadena sería una forma de convertirla a número. Mientras que la concatenación (no hay operador de concatenación, simplemente separar las variables por espacio) convierte un número a cadena (por ejemplo concatenar un número con cadena vacía y guardar el resultado en una variable → cad = ("" 13). La operación de concatenación se recomienda hacerla entre paréntesis.

2. Estructuras de control

Awk tiene las estructuras de control (decisiones, bucles) propias de un lenguaje de programación estructurado con una sintaxis basada en la del lenguaje C (del cual también derivan Java, C#, y muchos más).

2.1 if - else

Como acabamos de mencionar, la estructura if – else es idéntica a la de los lenguajes estructurados cuya sintaxis está basa en C. La mejor forma de verlo es con un ejemplo (**ejempo5.awk**) para procesar el archivo **pedido.txt**:

```
BEGIN { FS = ";" }
{
      if ($4 >= 100) {
          print "El producto "$3" es caro"
      } else if ($4 >= 50) {
          print "El producto "$3" tiene un precio medio"
      } else {
          print "El producto "$3" es barato"
      }
}
```

En este caso, cuando sólo se tiene que procesar una instrucción por bloque, las llaves se pueden omitir sin problema.

2.2 while

El lenguaje **awk** también soporta las estructuras **while** y **do – while**. Su propósito es el mismo que en el resto de lenguajes, y las instrucciones de incremento y decremento también se utilizan sin problemas, como se puede ver a continuación (**ejemplo6.awk**):

```
BEGIN { FS = ";" }
{
    i = 1
    while (i <= $5) {
        printf "%d unidades de %s cuestan: %.2f€\n", i, $3,($4*i)
        i++
    }
}
```

2.3 for

Lo mismo pasa con la estructura **for** (**ejemplo7.awk**):

```
BEGIN { FS = ";" }
{
    for (i = 1; i <= $5; i++)
        printf "%d unidades de %s cuestan: %.2f€\n", i, $3,($4*i)
}
```

2.4 break y continue

Como en el resto de lenguajes que soportan las estructuras while y for, existen las instrucciones **break**, que interrumpe la ejecución del bucle actual (si hay varios anidados, el más interno), o **continue**, que salta las instrucciones siguientes y se mueve a la siguiente iteración del bucle y volviendo a evaluar la condición (y ejecutando el incremento en el caso del for).

Aquí tampoco está recomendado abusar de estas instrucciones si no está bien justificado, ya que harían poco predecible el comportamiento del programa. Normalmente estas instrucciones se pueden sustituir mediante un simple if para evitar que se ejecute algo cuando no queremos (continue) o aprovechando la condición del bucle para detenerlo (break). A veces para evitar anidar muchos bloques, o hacer más compleja la estructura, sí que tiene sentido usarlos, pero con cuidado

Ejemplo de uso de continue (en este caso con un if sería mejor):

```
BEGIN { FS = ";" }
{
   total = 0;
   #Recorremos los campos de la fila
   for (i = 1; i <= NF; i++) {
       # Si el campo no es numérico lo saltamos y no sumamos
        if($i !~ /[0-9]+/)
            continue

        total += $i #$i referencia al valor del campo
   }
   print "El total de la fila",NR,"es:",total
}
```

2.5 next y exit

La sentencia **next** lee el siguiente registro del archivo, por lo que las instrucciones posteriores a dicha sentencia operan sobre el siguiente registro. Esto es útil para comprobaciones al principio del programa (después de BEGIN si está).

```
NF != 3 {
      print "El registro",NR,"no tiene 3 campos"
      next
}
{ ... }
```

La sentencia **exit** hace que el programa pare de procesar registros deteniéndose su ejecución. Eso sí, si existe un bloque END, este sí que se ejecutará. Después de la instrucción exit se indica un número, que será **0** para indicar que todo ha ido bien, y un valor diferente para indicar error (**exit 1**).

2.6 Operador ternario

En awk también existe el operador ternario → **valor = $1 > 100?"caro":"barato"**

3. Arrays

Los arrays en **awk** funcionan de forma similar a muchos lenguajes de programación estructurados, sobre todo a lenguajes de script. Los arrays no hace falta crearlos de forma explícita, sino que se crean con el primer valor que se le asigna a una posición. Por ejemplo:

```
array[0] = "valor"
```

Además, los arrays en awk, son por defecto asociativos (como en PHP por ejemplo). Esto quiere decir que no tiene por qué ser numérico el índice (puede ser una cadena). Además, los índices numéricos no tienen por qué inicializarse en orden, se pueden dejar índices intermedios sin asignar, y no hay que preocuparse por el tamaño máximo del array.

Para comprobar si un índice existe en un array se utiliza la expresión **indice in array**. Por ejemplo:

```
if ( "precio" in array ) {
      print "El precio del producto es:",array["precio"],"€"
}
```

Si se quiere borrar un índice se utiliza: **delete array[índice]**.

Cuando se accede a un índice que no existe, no se produce ningún error, simplemente devuelve cadena vacía.

3.1 Recorrido de un array (for..in)

Existe en awk una expresión equivalente a lo que sería **foreach** en otros lenguajes. Esto es útil ya que el for de toda la vida no sirve en todos los casos, ya que puede haber índices numéricos intermedios sin asignar, o simplemente, que algunos índices o todos no sean numéricos. De esta forma nos aseguramos que se recorren todos los elementos:

```
for ( indice in array ) {
      print "El valor de array["indice"] es:",array[indice]
}
```

3.2 Arrays multidimensionales

Aunque se pueden utilizar varios índices en un array en **awk**, internamente estos índices se convierten a cadena utilizando el valor que haya en la variable SUBSEP para concatenarlos (por defecto es un carácter no imprimible). Para referirnos a un índice compuesto se hace así → **array[x, y]**. Los índices pueden ser numéricos o cadenas.

A la hora de recorrerlos se puede hacer con un for..in (todos los elementos), o con bucles anidados, uno para índice:

```
for (i = 0; i < MAX_I; i++) {
      for (j = 0; j < MAX_J; j++) {
           print array[i,j] }
}
```

4. Funciones

En los programas awk, también se pueden definir funciones con parámetros y que devuelvan un valor de salida. Además, existen varias funciones predefinidas en el lenguaje que simplifican tareas como operaciones aritméticas o con cadenas.

4.1 Funciones predefinidas

Funciones numéricas

- **int(n)** → Devuelve la parte entera de n, siendo este un número decimal (no redondea el número, simplemente quita los decimales).
- **sqrt(n)** → Raíz cuadrada de n.
- **exp(n)** → Te devuelve el exponencial de n (e^n)
- **log(n)** → Logaritmo natural de n.
- **sin(n)** → Seno de n. El número n debe estar expresado en radianes.
- **cos(n)** → Coseno de n.
- **atan2(y,x)** → Arcotangente de x/y.
- **rand()** → devuelve un número aleatorio decimal entre 0 y 1.
- **srand(x)** → Cambia la semilla para generar números aleatorios (si no, en cada ejecución del programa siempre se generan los mismos). Si se omite el parámetro x, se usa la fecha y hora del momento para establecer la semilla. Esta instrucción debe ejecutarse antes de llamar por primera vez a **rand()**.
- **systime()** → Devuelve el tiempo en segundos desde el 1 de Enero de 1970. Más información y funciones para manejo de fechas:
  - https://www.gnu.org/software/gawk/manual/html_node/Time-Functions.html

Funciones de cadenas

- **index(cadena,cadBuscar)** → Devuelve la posición de ***cadBuscar*** dentro de ***cadena***, empezando por 1. Si no la encuentra devuelve 0.
  - `index("hola que tal","que")` → 6
- **length(cadena)** → Devuelve la longitud de la cadena.
- **match(cadena,regexp)** → Equivale a la función index pero soporta expresiones regulares. Devuelve la posición a partir la cual se encuentra una coincidencia con la expresión regular (empezando en 1). También se le da valor a las variables implícitas RSTART → mismo valor que devuelve la función, y RLENGTH → longitud de la coincidencia con la expresión regular.
- **split(cadena,array,separador)** → Divide una cadena en trozos y los guarda en un array (que se le pasa por parámetro) a partir de un separador. El primer índice del array **es 1, y no 0**. Devuelve el número de trozos generados.

- `split("hola-que-tal",trozos,"-")` → 3
- `trozos[1]` → "hola"

- **sprintf("formato",item1,item2,...)** → Devuelve la cadena formateada al estilo printf en lugar de imprimirla por pantalla.
- **sub(regexp, reemplazo, cadenaDest)** → Reemplaza la primera coincidencia de la expresión regular dentro de la variable cadenaDest, modificando esta y devolviendo un 0 si no ha reemplazado nada o 1 si ha encontrado coincidencia. El tercer parámetro (cadenaDest) se puede omitir y entonces actúa sobre $0, el registro completo, modificándolo.

```
str="Me voy a EEUU";
sub(/EEUU/,"Estados Unidos", str);
print str #Imprime "Me voy a Estados Unidos"
```

- **gsub(regexp, reemplazo, cadenaDest)** → Actúa igual que sub, pero reemplaza todas las coincidencias de la expresión. Devuelve el número de coincidencias reemplazadas.
- **substr(cadena, inicio, longitud)** → Devuelve una subcadena, de ***longitud*** caracteres a partir de ***inicio***. Si se omite la longitud, devuelve hasta el final.
- **tolower(cadena)** → Devuelve la cadena en minúsculas.
- **toupper(cadena)** → Devuelve la cadena en mayúsculas.

Funciones de arrays

- **asort(array,arrayDest)** → A partir de un array, genera otro array (**arrayDest**) que contendrá los valores del primero ordenados. Devuelve el número de elementos del array, y guarda en **arrayDest** los valores ordenados.
- **asorti(array,arrayDest)** → A partir de un array asociativo, genera otro array (**arrayDest**), cuyos valores son los índices del primero ordenados.

Funciones del sistema

- **system(comando)** → Ejecuta un comando del sistema operativo. Devuelve el estado de finalización del comando (0 → correcto) y no su salida.

4.2 Funciones definidas por el usuario

En **awk** se pueden definir funciones en cualquier parte siempre que no sea dentro de un bloque. Lo ideal es definirlas todas al principio del archivo, antes de empezar con el programa. El formato para definir una función es el siguiente:

```
function nombreFunción(parámetros) {
        cuerpo de la función
}
```

Al llamar a una función no hace falta pasarle todos los parámetros. A los parámetros omitidos se les asignará una cadena nula o vacía por defecto. Los parámetros se pasan por

valor, es decir, no influye que luego se cambie su contenido dentro de la función, a excepción de los arrays, que sí se pasan por referencia. Ejemplo (**ejemplo8.awk**):

```
#Intercambia las posiciones en un array
function swapPosArray(array,pos,pos2) {
    aux = array[pos]
    array[pos] = array[pos2]
    array[pos2] = aux
}
BEGIN {
    a[0] = 5
    a[1] = 12
    a[2] = 50
    a[3] = 13
    swapPosArray(a,0,3)
    for(i in a) {
        print a[i]
    }
}
```

awk -f ejemplo8.awk → 13 12 50 5

Las variables a las que se les da valor dentro de una función son locales a la misma, no existen fuera. Las funciones en awk retornan un valor utilizando la instrucción **return**.

```
function precioSinIva(precio) {
    return precio/(1+IVA/100)
}
BEGIN {
    FS=";"
    IVA=21 #Esta variable es global → Accesible en la función.
}
{
  printf "Producto: %s. Precio sin IVA: %.2f€\n",$3,precioSinIva($4)
}
```

5. Ejemplos

5.1 Contar letras

En el siguiente ejemplo trabajaremos con un archivo llamado letras.txt que contendrá varias filas con un número indeterminado de letras separadas por coma en este formato:

```
a,b,m,f,g,h,j,e,t,a,d,b,d,g,s
h,g,k,s,a,d,b,s,d,a,w,e,r,e
a,g,h,f,h,v,d,a,f,d,s,g,d
k,g,h,n,m,d,s,f,e,r
b,v,n,c,m,j,g,k,a,s,e,q,w
a,s,d,f,x,c,d,f,s,a,q
```

El objetivo es contar las letras que aparecen en el archivo y mostrarlas ordenadas alfabéticamente junto al número de veces que aparece cada una. Para ello vamos a apoyarnos en la función asorti(array,array_dest) que a partir de un array asociativo, genera otro array cuyos valores son los índices del primero ordenados (**ejemplo10.awk**):

```
BEGIN { FS="," }
{
    for(i = 1; i <= NF; i++ ) {
        if($i in a == 0) { #No existe el índice (letra)
            a[$i] = 0
        }

        a[$i]++
    }
}
END {
    asorti(a,orderedIndex)
    for(i in orderedIndex) {
        print orderedIndex[i],"=",a[orderedIndex[i]]
    }
}
```