Introducción a AWK-GAWK.

Francisco Alonso Sarria

El intérprete sabe lo que debe hacer

- AWK es un lenguaje de programación cuya potencia estriba en la capacidad de obtener grandes resultados con programas de unas pocas lineas. Está especialmente
- El intérprete de awk "*sabe*" que el programa que va a ejecutar:
  - Lee un flujo de datos (entrada estandar, fichero, tubería)
  - Produce un flujo de salida (salida estandar, fichero, tubería)
  - El flujo está medianamente estructurado en registros y campos
- Por tanto no hacen falta instrucciones para explicarselo

El intérprete sabe lo que debe hacer

- AWK es un lenguaje de programación cuya potencia estriba en la capacidad de obtener grandes resultados con programas de unas pocas lineas. Está especialmente
- El intérprete de awk "*sabe*" que el programa que va a ejecutar:
  - Lee un flujo de datos (entrada estandar, fichero, tubería)
  - Produce un flujo de salida (salida estandar, fichero, tubería)
  - El flujo está medianamente estructurado en registros y campos
- Por tanto no hacen falta instrucciones para explicarselo

El intérprete sabe lo que debe hacer

- AWK es un lenguaje de programación cuya potencia estriba en la capacidad de obtener grandes resultados con programas de unas pocas lineas. Está especialmente
- El intérprete de awk "*sabe*" que el programa que va a ejecutar:
  - Lee un flujo de datos (entrada estandar, fichero, tubería)
  - Produce un flujo de salida (salida estandar, fichero, tubería)
  - El flujo está medianamente estructurado en registros y campos
- Por tanto no hacen falta instrucciones para explicarselo

El intérprete sabe lo que debe hacer

- AWK es un lenguaje de programación cuya potencia estriba en la capacidad de obtener grandes resultados con programas de unas pocas lineas. Está especialmente
- El intérprete de awk "*sabe*" que el programa que va a ejecutar:
  - Lee un flujo de datos (entrada estandar, fichero, tubería)
  - Produce un flujo de salida (salida estandar, fichero, tubería)
  - El flujo está medianamente estructurado en registros y campos
- Por tanto no hacen falta instrucciones para explicarselo

El intérprete sabe lo que debe hacer

- AWK es un lenguaje de programación cuya potencia estriba en la capacidad de obtener grandes resultados con programas de unas pocas lineas. Está especialmente
- El intérprete de awk "*sabe*" que el programa que va a ejecutar:
  - Lee un flujo de datos (entrada estandar, fichero, tubería)
  - Produce un flujo de salida (salida estandar, fichero, tubería)
  - El flujo está medianamente estructurado en registros y campos
- Por tanto no hacen falta instrucciones para explicarselo

El intérprete sabe lo que debe hacer

- AWK es un lenguaje de programación cuya potencia estriba en la capacidad de obtener grandes resultados con programas de unas pocas lineas. Está especialmente
- El intérprete de awk "*sabe*" que el programa que va a ejecutar:
  - Lee un flujo de datos (entrada estandar, fichero, tubería)
  - Produce un flujo de salida (salida estandar, fichero, tubería)
  - El flujo está medianamente estructurado en registros y campos
- Por tanto no hacen falta instrucciones para explicarselo

Un ejemplo sencillo

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

fichero.txt

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

Un ejemplo sencillo

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

fichero.txt

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

Un ejemplo sencillo

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

fichero.txt

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

Un ejemplo sencillo

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

fichero.txt

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

Lectura linea a linea

- Lectura linea a linea del flujo de texto en función de dos variables:
  - FS es el separador de campos, por defecto es un espacio
  - RS es el separador de registros, por defecto es el retorno de carro
- La lectura de una linea implica dar valor a una serie de variables predefinidas:
  - NR, número de registro
    NF, número de campos leidos
    $0, contiene todo el registro leido
    $1, ..., $NF, contiene los NF campos leidos

Lectura linea a linea

- Lectura linea a linea del flujo de texto en función de dos variables:
  - FS es el separador de campos, por defecto es un espacio
  - RS es el separador de registros, por defecto es el retorno de carro
- La lectura de una linea implica dar valor a una serie de variables predefinidas:
  - NR, número de registro
  - NF, número de campos leidos
  - $0, contiene todo el registro leido
  - $1, ..., $NF, contiene los NF campos leidos

Lectura linea a linea

- Lectura linea a linea del flujo de texto en función de dos variables:
  - FS es el separador de campos, por defecto es un espacio
  - RS es el separador de registros, por defecto es el retorno de carro
- La lectura de una linea implica dar valor a una serie de variables predefinidas:
  - NR, número de registro
  - NF, número de campos leidos
  - $0, contiene todo el registro leido
  - $1, ... , $NF, contiene los NF campos leidos

Lectura linea a linea

- Lectura linea a linea del flujo de texto en función de dos variables:
    - FS es el separador de campos, por defecto es un espacio
    - RS es el separador de registros, por defecto es el retorno de carro
- La lectura de una linea implica dar valor a una serie de variables predefinidas:
    - NR, número de registro
    - NF, número de campos leidos
    - $0, contiene todo el registro leido
    - $1, ... , $NF, contiene los NF campos leidos

Lectura linea a linea

- Lectura linea a linea del flujo de texto en función de dos variables:
  - FS es el separador de campos, por defecto es un espacio
  - RS es el separador de registros, por defecto es el retorno de carro
- La lectura de una linea implica dar valor a una serie de variables predefinidas:
  - NR, número de registro
  - NF, número de campos leidos
  - $0, contiene todo el registro leido
  - $1, ... , $NF, contiene los NF campos leidos

Lectura linea a linea

- Lectura linea a linea del flujo de texto en función de dos variables:
  - FS es el separador de campos, por defecto es un espacio
  - RS es el separador de registros, por defecto es el retorno de carro
- La lectura de una linea implica dar valor a una serie de variables predefinidas:
  - NR, número de registro
  - NF, número de campos leidos
  - $0, contiene todo el registro leido
  - $1, ... , $NF, contiene los NF campos leidos

Lectura linea a linea

- Lectura linea a linea del flujo de texto en función de dos variables:
  - FS es el separador de campos, por defecto es un espacio
  - RS es el separador de registros, por defecto es el retorno de carro
- La lectura de una linea implica dar valor a una serie de variables predefinidas:
  - NR, número de registro
  - NF, número de campos leidos
  - $0, contiene todo el registro leido
  - $1, ... , $NF, contiene los NF campos leidos

Un ejemplo sencillo

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

NR=1
NF=5
$0="Murcia 3 2 3 4"
$1="Murcia $2="3" $3="2" $4="3" $5="4"

Un ejemplo sencillo

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

NR=2
NF=5
$0="Albacete 3 4 5 3"
$1="Albacete" $2="3" $3="4" $4="5" $5="3"

Un ejemplo sencillo

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

NR=3
NF=4
$0=“Almería 3 3 2”
$1=“Almería $2=”3” $3=”3” $4=”2” $5=

Un ejemplo sencillo

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

NR=4
NF=5
$0="Alicante 4 5 2 2"
$1="Alicante" $2=''4" $3=''5" $4=''2" $5=2

Un ejemplo sencillo

```
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

Un ejemplo sencillo

```
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

Principales utilidades de AWK

- Producir resúmenes a partir de grandes listados de datos medianamente estructurados
- Los programas son mucho más cortos que los equivalentes en otros lenguajes
- Crear programas de una sola linea embebidos en *shell scripts* e integrados con otras utilidades del sistema mediante redirecciones o tubería: `ls -l|awk '$5>200000{print $0}'>grandes.txt`

Patrones y acciones

- Cada linea leída se compara con varios patrones y se ejecutan las acciones asociadas con aquellos patrones a los que se ajusta la linea patrón {acción}
- Existen tres patrones especiales
  - BEGIN, las acciones asociadas se ejecutan antes de comenzar a procesar el fichero. Util para dar valor a FS y RS
  - END, las acciones asociadas se ejecutan tras procesar el fichero
  - (patrón vacío), las acciones asociadas se ejecutan para todas las lineas

Patrones y acciones

- Cada linea leída se compara con varios patrones y se ejecutan las acciones asociadas con aquellos patrones a los que se ajusta la linea patrón {acción}
- Existen tres patrones especiales
  - BEGIN, las acciones asociadas se ejecutan antes de comenzar a procesar el fichero. Util para dar valor a FS y RS
  - END, las acciones asociadas se ejecutan tras procesar el fichero
  - (patrón vacio), las acciones asociadas se ejecutan para todas las lineas

Patrones y acciones

- Cada linea leída se compara con varios patrones y se ejecutan las acciones asociadas con aquellos patrones a los que se ajusta la linea patrón {acción}
- Existen tres patrones especiales
  - BEGIN, las acciones asociadas se ejecutan antes de comenzar a procesar el fichero. Util para dar valor a FS y RS
  - END, las acciones asociadas se ejecutan tras procesar el fichero
  - (patrón vacio), las acciones asociadas se ejecutan para todas las lineas

Patrones y acciones

- Cada linea leída se compara con varios patrones y se ejecutan las acciones asociadas con aquellos patrones a los que se ajusta la linea patrón {acción}
- Existen tres patrones especiales
  - BEGIN, las acciones asociadas se ejecutan antes de comenzar a procesar el fichero. Util para dar valor a FS y RS
  - END, las acciones asociadas se ejecutan tras procesar el fichero
  - (patrón vacio), las acciones asociadas se ejecutan para todas las lineas

Patrones y acciones

- Cada linea leída se compara con varios patrones y se ejecutan las acciones asociadas con aquellos patrones a los que se ajusta la linea patrón {acción}
- Existen tres patrones especiales
  - BEGIN, las acciones asociadas se ejecutan antes de comenzar a procesar el fichero. Util para dar valor a FS y RS
  - END, las acciones asociadas se ejecutan tras procesar el fichero
  - (patrón vacio), las acciones asociadas se ejecutan para todas las lineas

Comparación con patrones

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

Muestra en pantalla las lineas cuya tercera columna sea mayor o igual a 4

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

Utiliza el patrón BEGIN para mostrar los nombres de las columnas

Comparación con patrones

El patrón `BEGIN` es un buen lugar para dar valor a las variables de AWK.

```
BEGIN {FS=";"}
```

Comparación con patrones

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

Variables en awk

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

Arrays en awk

- Un array es una tabla de varios elementos que se distinguen por sus índices que puede ser tanto un número como una cadena array[índice]
- En awk no es necesario declarar el tamaño del array
- awk maneja arrays asociativos de dimensión 1 por lo que puede utilizarse cualquier índice, incluso simular arrays ndimensionales array[i,j]

Arrays en awk

- Un array es una tabla de varios elementos que se distinguen por sus índices que puede ser tanto un número como una cadena array[índice]
- En awk no es necesario declarar el tamaño del array
- awk maneja arrays asociativos de dimensión 1 por lo que puede utilizarse cualquier índice, incluso simular arrays ndimensionales array[i,j]

Arrays en awk

- Un array es una tabla de varios elementos que se distinguen por sus índices que puede ser tanto un número como una cadena array[índice]
- En awk no es necesario declarar el tamaño del array
- awk maneja arrays asociativos de dimensión 1 por lo que puede utilizarse cualquier índice, incluso simular arrays ndimensionales array[i,j]

Arrays en awk

- Un array es una tabla de varios elementos que se distinguen por sus índices que puede ser tanto un número como una cadena array[índice]
- En awk no es necesario declarar el tamaño del array
- awk maneja arrays asociativos de dimensión 1 por lo que puede utilizarse cualquier índice, incluso simular arrays ndimensionales array[i,j]

```
awk '    {nombre[1]=”Pepe”
          apellido[“Pepe”]=”López”
          lluvia[“enero”]=23
          lluvia[1,1996]=23}'
```

Arrays en awk

La función `in` permite determinar si un determinado valor se ha utilizado como clave en un array asociativo:

```
if ("enero" in lluvia)
```

o recorrer los valores de las claves utilizadas:

```
for (mes in lluvia){
 print mes,lluvia[mes]
}
```

Los valores leidos puede también utilizarse como claves de un array asociativo

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

Entrada de datos

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*:

`getline` : Lee una nueva linea del flujo de entrada

Entrada de datos

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*:

`getline` :	Lee una nueva linea del flujo de entrada
`getline` linea :	Lee una nueva y la guarda en linea

Entrada de datos

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*:

`getline` :	Lee una nueva linea del flujo de entrada
`getline` linea :	Lee una nueva y la guarda en linea
`getline` linea < “fic” :	Lee una linea de “fic” y la guarda en linea

Entrada de datos

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*:

`getline` :	Lee una nueva linea del flujo de entrada
`getline` linea :	Lee una nueva y la guarda en linea
`getline` linea < “fic” :	Lee una linea de “fic” y la guarda en linea
`close`(fichero) :	Cierra un fichero o flujo de texto abierto

Organización de la salida

- La función básica de salida es print
- `nombre="Pepe";print "Hola",nombre`
  `Hola Pepe`
- `print "Hola a un fichero">"fichero_salida.txt"`
- `print "Hola a un fichero">>"fichero_salida.txt"`
- El separador de campos para la salida del programa es, por defecto el espacio en blanco; el separador de registros es el retorno e carro.
- Ambos separadores pueden modificarse con las variables OFS y ORS

Organización de la salida

- La función básica de salida es print
- `nombre=”Pepe”;print “Hola”,nombre`
  `Hola Pepe`
- `print “Hola a un fichero”>”fichero_salida.txt”`
- `print “Hola a un fichero”>>”fichero_salida.txt”`
- El separador de campos para la salida del programa es, por defecto el espacio en blanco; el separador de registros es el retorno e carro.
- Ambos separadores pueden modificarse con las variables OFS y ORS

Organización de la salida

- La función básica de salida es print
- nombre="Pepe";print "Hola",nombre
  Hola Pepe
- print "Hola a un fichero">"fichero_salida.txt"
- print "Hola a un fichero">>"fichero_salida.txt"
- El separador de campos para la salida del programa es, por defecto el espacio en blanco; el separador de registros es el retorno e carro.
- Ambos separadores pueden modificarse con las variables OFS y ORS

Organización de la salida

- La función básica de salida es print
- `nombre="Pepe";print "Hola",nombre`
  `Hola Pepe`
- `print "Hola a un fichero">"fichero_salida.txt"`
- `print "Hola a un fichero">>"fichero_salida.txt"`
- El separador de campos para la salida del programa es, por defecto el espacio en blanco; el separador de registros es el retorno e carro.
- Ambos separadores pueden modificarse con las variables `OFS` y `ORS`

Organización de la salida

- La función básica de salida es print
- `nombre="Pepe";print "Hola",nombre`
  `Hola Pepe`
- `print "Hola a un fichero">"fichero_salida.txt"`
- `print "Hola a un fichero">>"fichero_salida.txt"`
- El separador de campos para la salida del programa es, por defecto el espacio en blanco; el separador de registros es el retorno e carro.
- Ambos separadores pueden modificarse con las variables `OFS` y `ORS`

Organización de la salida

- La función básica de salida es print
- `nombre="Pepe";print "Hola",nombre`
  `Hola Pepe`
- `print "Hola a un fichero">"fichero_salida.txt"`
- `print "Hola a un fichero">>"fichero_salida.txt"`
- El separador de campos para la salida del programa es, por defecto el espacio en blanco; el separador de registros es el retorno e carro.
- Ambos separadores pueden modificarse con las variables `OFS` y `ORS`

Vuelta al ejemplo sencillo

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

```
awk '   $1=="Alicante"{print $1}
        $3+0>3{print $0}
        {suma=suma+$2}
        END{print "suma=",suma}' fichero.txt
```

Vuelta al ejemplo sencillo

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

```
awk '   $1=="Alicante"{print $1}
        $3+0>3{print $0}
        {suma=suma+$2}
        END{print "suma=",suma}' fichero.txt
```

```
Albacete 3 4 5 3
```

Vuelta al ejemplo sencillo

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

```
awk '   $1=="Alicante"{print $1}
        $3+0>3{print $0}
        {suma=suma+$2}
        END{print "suma=",suma}' fichero.txt
```

Albacete 3 4 5 3

Vuelta al ejemplo sencillo

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

```
awk '   $1=="Alicante"{print $1}
        $3+0>3{print $0}
        {suma=suma+$2}
        END{print "suma=",suma}' fichero.txt
```

```
Albacete 3 4 5 3
Alicante
Alicante 4 5 2 2
```

Vuelta al ejemplo sencillo

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

```
awk '  $1=="Alicante"{print $1}
       $3+0>3{print $0}
       {suma=suma+$2}
       END{print "suma=",suma}' fichero.txt
```

```
Albacete 3 4 5 3
Alicante
Alicante 4 5 2 2
 suma 13
```

Toma de decisiones

if (condicion) {expresiones} *else* {expresiones}

if (x%2 == 0) { print x,“es par” } else { print x,”es impar” }

Toma de decisiones

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"
}
```

Bucle while

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

Sentencias de control

do {expresiones} *while* (condicion)

```
i = 1
do { print $i ; i++ } while (i <= NF)
```

Bucle for

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

Break y continue

```
awk '
    BEGINfor (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
```

while + getline

Este script muestra como leeer un fichero diferente al que se le pasa al intérprete del lenguaje.

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

next

`next` lee la siguiente linea del fichero de entrada y reinicia con él el proceso de comparación de patrones.

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

Operadores aritmético-lógicos

- Expresiones de asignación

  $=, +=, -=, *=, /=, \hat{}=, ++, --$

- Expresiones aritméticas

  $+, -, *, /, \%, \hat{}$

- Expresiones de comparación

  $>, <, >=, <=, ==, !=$

- Expresiones booleanas

  &&, ||

Operadores aritmético-lógicos

- Expresiones de asignación

  $=, +=, -=, *=, /=,$ ^=, $++, --$

- Expresiones aritméticas

  $+, -, *, /,$ %, ^

- Expresiones de comparación

  $>, <, >=, <=, ==, !=$

- Expresiones booleanas

  &&, ||

Operadores aritmético-lógicos

- Expresiones de asignación

  $=, +=, -=, *=, /=,$ ^=, $++, --$

- Expresiones aritméticas

  $+, -, *, /,$ %, ^

- Expresiones de comparación

  $>, <, >=, <=, ==, !=$

- Expresiones booleanas

  &&, ||

Operadores aritmético-lógicos

- Expresiones de asignación
  $=, +=, -=, *=, /=,$ ^=, $++, --$
- Expresiones aritméticas
  $+, -, *, /,$ %, ^
- Expresiones de comparación
  $>, <, >=, <=, ==, !=$
- Expresiones booleanas
  &&, ||

Funciones numéricas

Función: *int* (x)

Objetivo: Obtener el valor entero

Ejemplo: print 5.4 , int (5.4)

Resultado: 5.4 5

Funciones numéricas

Función: *sqrt* (x)

Objetivo: Calcula la raiz cuadrada

Ejemplo: print 9 , sqrt (9)

Resultado: 9 3

Funciones numéricas

Función: *exp* (x)

Objetivo: Calcula e^x

Ejemplo: print 5 , exp (5)

Resultado: 5 148.413

Funciones numéricas

Función: *log* (x)

Objetivo: Calcula el logaritmo neperiano de x

Ejemplo: print 5 , log (5)

Resultado: 5 1.60944

Funciones numéricas

Función: *sin* (x)

Objetivo: Calcula el seno de x

Ejemplo: print 5 , sin (5)

Resultado: 5 1.60944

Funciones numéricas

Función: *cos* (x)

Objetivo: Calcula el coseno de x

Ejemplo: print 5 , cos(5)

Resultado: 5 0.283662

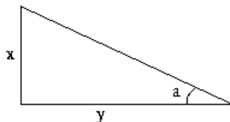
Funciones numéricas

Función: *atan2* (x , y)

Objetivo: Calcula el ángulo *a*

Ejemplo: print 5 , 2 ,atan2 (5 , 2)

Resultado: 5 1.19029



Funciones numéricas

rand(): Genera un número aleatorio

Funciones numéricas

rand(): Genera un número aleatorio
srand(x) : Cambia la semilla de generación de números aleatorios

Funciones de texto

Función: *index*(cad1,cad2)

Objetivo: Devuelve la posición, dentro de cad1 en que aparece cad2

Ejemplo: print index("Introducción a AWK","AWK")

Resultado: 17

Funciones de texto

Función:	*length*(cadena)
Objetivo:	Devuelve la longitud de cadena
Ejemplo:	print length("Introducción a AWK")
Resultado:	19

Funciones de texto

Función: *split*(cadena,array,separador)

Objetivo: Divide una cadena y coloca los resultados en un array

Ejemplo: `k=split("Introducción a AWK",cad," ");print cad[1],k,cad[k]`

Resultado: Introducción 3 AWK

Funciones de texto

Función:	*substr*(cadena,inicio,longitud)
Objetivo:	Obtiene una subcadena
Ejemplo:	print substr("Introducción a AWK",2,5)
Resultado:	ntrod

Funciones de texto

Función: *tolower*(cadena)

Objetivo: Pasa el texto a minúsculas

Ejemplo: print tolower("Introducción a AWK")

Resultado: Introducción a awk

Funciones de texto

Función: *toupper*(cadena)

Objetivo: Pasa el texto a mayúsculas

Ejemplo: print toupper("Introducción a AWK")

Resultado: INTRODUCCIÓN A AWK

Funciones de texto

Función: *sub*(cadena,patrón,sust)

Objetivo: Sustituye la primera aparición de `patrón` por `sust`

Ejemplo: print sub("Introducción a AWK","Introducción","Inicio")

Resultado: Inicio a AWK

Funciones de texto

Función:	*gsub*(cadena,patrón,sust)
Objetivo:	Sustituye todas las apariciones de `patrón` por `sust`
Ejemplo:	print gsub("Introducción a AWK","Introducción","Inicio")
Resultado:	Inicio a AWK

Funciones de texto

Función: *sprintf*(formato,expresión,...)

Objetivo: Crea una cadena a partir de varias variables

Ejemplo: k="AWK";print sprintf(“Introducción a %s”,k)

Resultado: Introducción a AWK

Funciones de texto

Función: *match*(cad,regexp)

Objetivo: Comprueba una expresión regular

Ejemplo: match("Introducción a AWK","n")

Resultado: 2

Funciones definidas por el usuario

- Se colocan dentro del programa antes de cualquier patrón
- function nombre (parámetros) {expresiones}

Funciones definidas por el usuario

- Se colocan dentro del programa antes de cualquier patrón
- function nombre (parámetros) {expresiones}

Funciones definidas por el usuario

- Se colocan dentro del programa antes de cualquier patrón
- function nombre (parámetros) {expresiones}

```
awk '  function menor ( x , y ) { if (x > y ) then
       {suma = suma + menor ( $2 ,$3)}
       END{print "suma=",suma}'
```

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 %d:  Prov.=%s Variable 1=%d \
       Dato=%d\n",NR,$1,$2,dato)
}' fichero.txt
```

```
Registro 1:  Prov.=Murcia Variable 1=3 Dato=26
Registro 2:  Prov.=Albacete Variable 1=3 Dato=26
Registro 3:  Prov.=Almería Variable 1=3 Dato=26
Registro 4:  Prov.=Alicante Variable 1=4 Dato=26
```

Cálculos con números reales

AWK puede utilizarse para hacer cálculos con números reales en un script de BASH.

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

Formateo de ordenes

Este script consulta a PostgreSQL los nombres de los 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
```

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")
```

Conclusión: AWK es un buen lenguaje para:

- Producir resúmenes a partir de grandes listados de datos medianamente estructurados
- Los programas son mucho más cortos que equivalentes en otros lenguajes
- Programas de usar y tirar integrados en *shell scripts*. Por ejemplo:

```
awk '{np=np+NF;nc=nc+length($0)} END{print
NR,np,nc}'
```

  es equivalente a wc.

Conclusión: AWK es un buen lenguaje para:

- Producir resúmenes a partir de grandes listados de datos medianamente estructurados
- Los programas son mucho más cortos que equivalentes en otros lenguajes
- Programas de usar y tirar integrados en *shell scripts*. Por ejemplo:
  `awk '{np=np+NF;nc=nc+length($0)} END{print NR,np,nc}'`
  es equivalente a wc.

Conclusión: AWK es un buen lenguaje para:

- Producir resúmenes a partir de grandes listados de datos medianamente estructurados
- Los programas son mucho más cortos que equivalentes en otros lenguajes
- Programas de usar y tirar integrados en *shell scripts*. Por ejemplo:

  ```
  awk '{np=np+NF;nc=nc+length($0)} END{print
  NR,np,nc}'
  ```

  es equivalente a wc.