Tema 1
· Programación del shell ·

1.1 Introducción

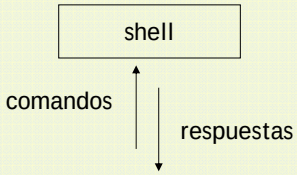
1.2 Programación de shell-scripts

1.3 AWK

1.1. Introducción

El shell de UNIX… un programa intérprete

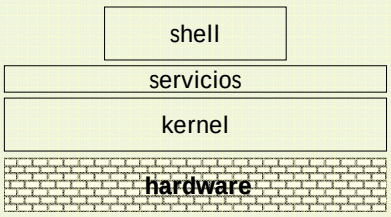
- Las funciones clásicas de los sistemas operativos:
  - gestión de recursos
  - ejecución de servicios para los programas
  - **ejecución de mandatos**
- El shell es el programa que en UNIX realiza las labores de *intérprete de comandos*.
- Nuestra comunicación con el shell…



1.1. Introducción

El shell de UNIX… arquitectura en 3 capas

- Es, por tanto, un programa interactivo "intermediario" entre el usuario y el núcleo del sistema operativo:



- Existen varios, entre los más destacados:
  - Bourne shell (sh)
  - Korn shell (ksh)
  - C-shell (csh)
  - GNU shell (bash)

1.1. Introducción

El shell de UNIX… ¿ cuándo y cual ?

- El shell que cada usuario de un sistema UNIX tiene especificado en el fichero **/etc/passwd** se inicia automáticamente cuando entra en sesión

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
```

1.1. Introducción

El shell de UNIX… un lenguaje de programación

- Además de servir como intérprete de comandos, el shell de UNIX permite crear programas, denominados **shell-scripts**, a partir de un lenguaje de programación proporcionado por el mismo, u otras utilidades como **awk**
- Se escribirán, por tanto, estos programas en ficheros cuyo contenido serán comandos pensados para ser leídos y ejecutados por el shell
- ¿Cómo ejecutar estos programas?
  1. Invocando al shell que se desee y a continuación el nombre del fichero con sus argumentos

     `$prompt> <nombre_del_shell> <nombre_fichero> {<arg1> … <arg2>}`
  2. Haciendo ejecutable el fichero previamente e invocando directamente el programa junto con sus argumentos

     `$prompt> chmod a+x <nombre_fichero>`
     `$prompt> <nombre_fichero> {<arg1> … <arg2>}`

1.2. Programación de shell-scripts

herramienta de programación… características

- Algunas de las principales características del lenguaje de programación del shell:
  - tratamiento de parámetros y variables
  - sentencias de control
  - lectura / escritura interactiva
  - definición de funciones
  - …
- Los programas que se pueden crear permiten incorporar funcionalidades más avanzadas que los ficheros batch de MS-DOS. Se utilizan a menudo como herramientas necesarias en la administración de los sistemas UNIX

1.2. Programación de shell-scripts

parámetros

- Parámetros *posicionales*
  - se denominan así los argumentos de un shell-script, puesto que se accede a ellos a partir de la posición que ocupan en la lista de argumentos
  - el nombre del procedimiento shell se considera el argumento 0 y se referencia con $0, el primer parámetro con $1, el segundo con $2 y así sucesivamente hasta $9
- Parámetros *especiales*
  - **$#** contiene el número de argumentos empleados en la llamada al shell-script excluyendo el propio nombre del shell-script
  - la orden **shift** quita el primer argumento de la lista y renumera
  - **$*** expande la lista de argumentos

1.2. Programación de shell-scripts

parámetros (ii)

- Variables
  - coincide con el concepto tradicional, aunque la funcionalidad asociada es más reducida
  - las variables no se declaran. Se crean cuando se les asigna un valor
  - no existen tipos de datos
  - acceder a una variable no existente no produce error, simplemente el valor devuelto es el valor nulo
  - existen una serie de variables globales accesibles desde cualquier shell-script como, por ej., $HOME (directorio de trabajo del usuario), $PATH (ruta/s donde el shell busca las órdenes)

1.2. Programación de shell-scripts

comentarios

- El carácter **#** es utilizado para introducir comentarios en un shell-script
- El shell de UNIX ignora todo lo que haya en la **misma línea** a continuación de dicho carácter

```
# inicio shell-script
echo Hola mundo #comentarios del autor
echo Adios
# fin shell-script
```

- Además de incluir comentarios aclaratorios, es habitual aplicar un sangrado a cada nuevo subnivel de órdenes en un shell-script con un tabulador adicional. Se trata de una práctica que nos facilita el entendimiento del programa y no afecta en la interpretación por parte del shell de UNIX

1.2. Programación de shell-scripts

expresiones

- La construcción de expresiones se realiza a partir de la orden **test**
- Presenta dos alternativas sintácticas equivalentes:
  - el propio nombre de la orden *test*, seguido de argumentos que se resuelven de acuerdo a reglas específicas. Por ejemplo:
    `test $# -eq 0`
  - sin utilizar la palabra test, incluyendo los argumentos *entre corchetes* (rodeados de espacios en blanco)
    `[ $0 –eq 0 ]`
- Esta orden devuelve 0 si la expresión formada se evalúa a cierto. En caso de evaluarse a falso devuelve otro valor.
- Es posible construir expresiones para realizar comprobaciones sobre…
  - *valores numéricos*
  - *tipos de ficheros*
  - *cadenas de caracteres*

1.2. Programación de shell-scripts

expresiones (ii)

Valores numéricos

- Examinan la relación entre dos números que pueden estar representados por variables

Formato: N <primitiva> M

primitiva	significado
-eq	los valores de N y M son iguales
-ne	los valores de N y M no son iguales
-gt	N es mayor que M
-lt	N es menor que M
-ge	N es mayor o igual que M
-le	N es menor o igual que M

1.2. Programación de shell-scripts

expresiones (iii)

Tipos de ficheros

- Hacen referencia a la existencia o no de ficheros y a las propiedades de los mismos

Formato: <primitiva> NOMBRE_FICHERO

primitiva	significado
-s	verifica si el fichero existe y no está vacío
-f	verifica si el fichero existe y es normal
-d	verifica si el fichero es un directorio
-w	verifica si el fichero puede ser escrito
-r	verifica si el fichero puede ser leído
-x	verifica si el fichero puede ser ejecutado

1.2. Programación de shell-scripts

expresiones (iv)

Cadenas de caracteres

- Permiten comparar cadenas o comprobar su longitud

primitiva	significado
Formato: S <primitiva> R	
=	comprueba si las cadenas S y R son iguales
!=	comprueba si S y R no son iguales
Formato: <primitiva> S	
-z	Comprueba si la cadena S tienen longitud cero.
-n	Comprueba si la cadena S tiene longitud mayor que cero.

1.2. Programación de shell-scripts

expresiones (v)

- Pueden construirse expresiones complejas a partir de expresiones combinadas con operadores lógicos:
  - *(negación) !*

    ```
    ! $# –eq 0
    ```
  - *(y lógica) –a*

    ```
    $# –eq 1 –a $1 = prueba
    ```
  - *(o lógica) -o*

    ```
    $# –eq 2 –o $1 = prueba
    ```

1.2. Programación de shell-scripts

sentencias de control… if

- Permite condicionar la ejecución de órdenes. Presenta la siguiente estructura general:

if *orden*
then *grupo de ordenes a realizar*
fi

```
# comprobación del número de parámetros
# en caso de ausencia mostrar error
if test $# -eq 0
then    echo Ejecución sin parámetros
        exit 1
fi
```


1.2. Programación de shell-scripts

sentencias de control… if-else

- Estructura general:

if *orden*
then *grupo de ordenes a realizar*
else *grupo de ordenes a realizar*
fi

```
# mostrar el contenido del fichero pasado como
# argumento
if test $# -eq 0
then    echo "Uso: $0 <nombre_fichero>" >&2
        exit 1
else cat $1
fi
```

1.2. Programación de shell-scripts

sentencias de control… "if" anidados

- Estructura general:

if *orden*
then *grupo de ordenes a realizar*
else if *orden*
then *grupo de ordenes a realizar*
else **if** *orden*
…
fi
fi
fi

1.2. Programación de shell-scripts

sentencias de control… "if" anidados (ii)

```
# procedimiento que recibe el nombre de un fichero
# y una cadena y busca la cadena en el mismo
# muestra un mensaje de error si:
# a. el fichero no existe  b. la cadena es nula
if test $# -ne 2
  then  echo "Uso: $0 <nombre fichero> <cadena>" >&2
        exit
  else  if test ! -s $1
          then  echo "Fichero $1 no existe"  >&2
                exit
        else if test -z $2
               then echo "Cadena de longitud cero" >&2
                    exit
               else grep $2 $1
             fi
        fi
fi
```

1.2. Programación de shell-scripts

sentencias de control… "elif"

- Estructura general:

if *orden*
then *grupo de ordenes a realizar*
elif *orden*
then *grupo de ordenes a realizar*
elif *orden*
…
fi

1.2. Programación de shell-scripts

sentencias de control… "elif" (ii)

```
# equivalente al anterior con elif
if test $# -ne 2
  then  echo "Uso: $0 <nombre fichero> <cadena>" >&2
        exit
elif test ! -s $1
  then  echo "Fichero $1 no existe"  >&2
        exit
elif test -z $2
  then echo "Cadena de longitud cero" >&2
       exit
else grep $2 $1
fi
```

1.2. Programación de shell-scripts

sentencias de control… case

- Permite el control condicional, al igual que "elif". Su estructura general es:

  case *cadena* **in**
  patron1) grupo de ordenes a realizar;;
  patron2) grupo de ordenes a realizar;;
  …
  {) grupo de ordenes a realizar}*
  esac
- Si *cadena* coincide con alguno de los valores presentados en lista de valores *(patron1,patron2,…)* se ejecutará el grupo de ordenes asociado a dicha alternativa (el cual debe finalizar con un doble punto y coma)
- El patrón * se utiliza como caso por defecto

1.2. Programación de shell-scripts

sentencias de control… case (ii)

```
# concatena dos ficheros, el  segundo después del primero,
# guardando el resultado en el primero
# por último presenta el contenido del primer fichero

case  $#  in
  1)  cat $1
      ;;
  2)  cat $2 >> $1
      cat $1
      ;;
  *)  echo "Uso: $0 fich1 fich2"
esac
```

1.2. Programación de shell-scripts

sentencias de control… for

- Permite la ejecución repetitiva de órdenes. Su estructura general es:
  for *variable {in lista de valores}*
  do *grupo de ordenes a realizar*
  done
- En cada iteración la variable toma el valor del siguiente elemento en la lista de valores
- Si se prescinde de la lista de valores, se tomará como tal la lista formada por los argumentos especificados en la ejecución del shell-script

```
# crear los ficheros especificados
# como parámetros del shell-script
for i
do >$i
done
```


1.2. Programación de shell-scripts

sentencias de control… while

- Permite la ejecución repetitiva de órdenes. Su estructura general es:
  while *orden*
  do *grupo de ordenes a realizar*
  done

```
# presentar el contenido de los ficheros pasados como argumentos
if test $# -eq 0
  then  echo "Uso: $0 <fichero_1> ... <fichero_n> " >&2
  exit
fi
while  test $# -gt 0
do
  if  test  ! –s $1
    then echo "no existe el fichero $1" >&2
  else
    cat $1
  fi
  shift
done
```

1.2. Programación de shell-scripts

sentencias de control… until

- Presenta una semántica diferente a la sentencia *while*. Su estructura general es:

  until *orden*
  do *grupo de ordenes a realizar*
  done

```
# presentar el contenido de los ficheros pasados como argumentos
if test $# -eq 0
  then  echo "Uso: $0 fichero1 ... " > &2
  exit
fi
until  test $# -eq 0
do
  if test ! –s $1
    then echo "no existe el fichero $1" > &2
  else cat $1
  fi
  shift
done
```


1.2. Programación de shell-scripts

otros aspectos

- Las sentencias **break** (suspender la ejecución del bucle) y **continue** (suspende la ejecución de la iteración actual, iniciando la siguiente iteración) permiten "alterar" la ejecución secuencial del cuerpo de un bucle

```
# presentar el contenido de cada uno de los ficheros pasados como
# parámetros. Si alguno no existe, debe pasarse al siguiente

for file
do if  test ! –s $file
     then   echo "no existe el fichero $file" >  &2
     continue
   fi
   cat $file
done
```

1.2. Programación de shell-scripts

otros aspectos (ii)

- Las órdenes **true** y **false** devuelven 0 y distinto de cero respectivamente. Son útiles para la creación de programas que deben ejecutarse indefinidamente (bucles infinitos)

```
while true
  do ordenes
done
```

- Existen unas herramientas de apoyo al desarrollador de shell-scripts para su depuración:
  - validación sintáctica del shell-script:

```
ruta_interprete_comandos –n ruta_shell-scripts
$prompt>/bin/bash –n mi_primer_script.sh
```

  - realización de trazas de ejecución o ampliación de información:

```
set -x
set -v
```


1.3. AWK

introducción

- Herramienta incluida en los sistemas UNIX. Fue desarrollada por Aho, Weinberger y Kernighan en 1977
- Incorpora un lenguaje de programación basado en la búsqueda de patrones sobre un líneas en ficheros u otras unidades de texto
- Un *programa awk* es un conjunto *reglas*. Sintácticamente, una regla consiste en un *patrón* seguido por una o varias acciones (encerrada/s entre llaves). Las reglas están separadas por saltos de línea:

  patrón {acción/es}
  patrón {acción/es}
  …
- Cuando en una línea de la entrada, se encuentra un patrón, awk realiza las acciones asociadas a dicho patrón sobre dicha línea

1.3. AWK

introducción (ii)

- Un primer ejemplo de programa awk…

```
awk '/Fernando/ {print $0}' lista_notas_jun04.txt

# La utilidad awk lee el fichero lista_notas_jun04.txt línea a línea,
# buscando la palabra Fernando e imprimiendo aquellas líneas en las
# que la encuentre hasta que se alcanza el final del fichero
```

- En una regla awk, el patrón o la acción pueden ser omitidas, pero no ambos. Si se omite el patrón, la acción se realiza para cada línea de la entrada. Si se omite la acción, por defecto se imprimen todas las líneas que concuerden con el patrón
- Si para una misma línea de la entrada concuerdan varios patrones, se ejecutan las distintas acciones asociadas a cada uno de ellos en el orden en el que aparecen en el programa awk. Si ninguno "encaja" no se ejecuta ninguna acción

1.3. AWK

introducción (iii)

- Los programas awk suelen ser muy útiles en la generación de informes a partir de ficheros en formato tabular o del resultado de otros programas o comandos UNIX
- Son programas normalmente simples y cortos para la realización de este tipo de tareas. En la implementación de tareas que impliquen desarrollos más extensos suelen ser recomendables otro tipo de herramientas

1.3. AWK

ejecución de programas awk

- Existen varias formas:
  - escribiendo directamente el programa en la línea de comandos entre **comillas simples**…

```
$prompt> awk 'programa' <fichero_entrada1> ... <fichero_entradaN>
```

  - escribiendo el programa en un fichero (¿.awk?) y ejecutándolo de la siguiente forma:

```
$prompt> awk –f fichero_programa <fichero_entrada1> ... <fichero_entradaN>
```

- La ejecución del programa awk puede realizarse sobre tantos ficheros de entrada como se especifiquen. Si no se incluyen ficheros de entrada se aplicará el programa a la entrada estándar hasta encontrar la marca de final de fichero (Control-d)

1.3. AWK

patrones

- Como se ha visto, los patrones en AWK controlan la ejecución de las reglas. Existen diferentes tipos:
  - expresiones regulares como patrones
  - expresiones de comparación como patrones
  - rangos como patrones
  - patrones compuestos
  - patrones especiales

1.3. AWK

expresiones regulares como patrones

- Permiten localizar, en el/los fichero/s de entrada, cualquier texto perteneciente a una clase de cadenas de texto.
- Las clases de cadenas de texto se especifican mediante **expresiones regulares** escritas entre **slashes** "/"
  - La expresión regular más simple es una secuencia de números, letras o ambos
  - Se emplean además un conjunto de símbolos, o **metacaracteres**:

símbolo	definición	símbolo	definición
|	alternación	+	al menos una ocurrencia
^, $	comienzo y fin de línea	*	cero o más ocurrencias
[ ] [-]	clase de carácter / rango	?	cero o una ocurrencia
.	caracter único	()	agrupación de expresiones

1.3. AWK

expresiones regulares como patrones (ii)

- Ejemplos de patrones construidos a partir de expresiones regulares:

```
/Fernando/   /245/   /^245/

/Fernando$/   /[245]/   /p+/

/p*/   /p?/   /[0-9]/

/Fernando|Oscar/ /[ABC]/ /.s/ /(pedro)+/
```

1.3. AWK

expresiones de comparación como patrones

- Chequean relaciones entre dos cadenas o números

símbolo	definición	símbolo	definición
<	menor que	<=	menor o igual
>	mayor que	>=	mayor o igual
==	es igual	!=	no es igual
~	identificación	!~	no identificación

- Si alguno de los dos operandos no es un número, ambos son convertidos y comparados como cadenas. Las cadenas son comparadas carácter a carácter
- Ejemplos:

```
variable == 20   "maria"<"pedro"
      variable ~ "manuel"
```


1.3. AWK

rangos como patrones

- Sintácticamente se construyen a partir de dos patrones separados por una coma:

patron_inicio , patron_fin

- Permiten actuar sobre aquellas líneas de la entrada consecutivas comprendidas entre la primera línea en la que "encaje" patron_inicio y la siguiente en la que "encaje" patron_fin
- La siguiente "búsqueda" se inicia a partir de la siguiente línea de la entrada en la que "encajó" patron_fin
- Podría suceder que ambos patrones "encajen" en la misma línea de la entrada
- Ejemplo de rango…

```
/^200/,/^400/
```

1.3. AWK

patrones compuestos

- Se trata de patrones combinados a partir de operadores booleanos

símbolo	operador
&&	y lógico
||	o lógico
!	negación

- Los patrones de rango y los patrones especiales no pueden combinarse para formar parte de un patrón compuesto
- Por ejemplo…

```
variable>25 && variable<100
```


1.3. AWK

patrones especiales

- BEGIN y END son patrones denominados "especiales". Se emplean en los programas awk para especificar las acciones a realizar antes de empezar a procesar la entrada (BEGIN) y después de haber finalizado (END)
- No existen acciones por defecto para estos patrones

```
BEGIN {contador=0}
…
END {print "El contador vale: " contador}
```

1.3. AWK

procesamiento de la entrada

- Los programas awk, según se ha expuesto, tomarán la entrada de la entrada estándar o de los ficheros especificados en la línea de comandos.
- La entrada se lee en unidades llamadas **registros** que serán procesados uno a uno por las reglas incluidas en el programa awk. Por defecto, el carácter separador de registro es el salto de línea, por lo que cada línea del fichero de entrada es un registro
- Cada registro es particionado a su vez en **campos**, a partir del carácter definido como separador de campo (por defecto el espacio en blanco).

1.3. AWK

variables

- awk permite crear variables, asignarles valores y realizar operaciones con ellas. Las variables en awk **no tienen tipos de datos permanentes**.
- El nombre de una variable es una expresión que representa el valor actual de la variable. Algunos nombres de variables están reservados por awk con carácter interno, dado su significado especial:

variable	valor	variable	valor
FS OFS	separador de campo de entrada / salida	RS ORS	separador de registro de entrada / salida
NF	número de campos del registro actual	NR	número de registros leídos hasta el momento
FILENAME	nombre del archivo de entrada	FNR	número de registros en el archivo actual

1.3. AWK

variables… identificadores de campo

- Son un tipo especial de variable interna. Se emplean para referenciar cada uno de los campos que componen un registro de la entrada
- Para referirse a un campo, en un programa awk, se emplea el signo $ seguido por el número de campo deseado. $1 se refiere al primer campo, $2 al segundo, … $NF
- A través de $0 se referencia al registro completo
- El número de un campo no necesita ser una constante. Cualquier expresión puede ser usada después de '$' para referirse a un campo.
- El valor de un campo, referenciado a través del identificador correspondiente, puede ser creado, borrado, modificado o intercambiado, aunque realmente todas estas operaciones **nunca modifican el fichero de entrada**.

1.3. AWK

variables… identificadores de campo (ii)

- Muestra utilizando identificadores de campo…

```
# los comentarios van precedidos, en cada línea, por # ...

# formato del fichero de entrada (en cada línea)
# <nombre_producto> <precio> <unidades>

{$4= $3 * $2}
# crea un cuarto campo...

{temp=$2; $2=$3; $3=temp}
#intercambio de posiciones...
```

1.3. AWK

operadores

- asignación de un valor a una variable (si el valor es una cadena debe ir entre comillas dobles " ")

  `=`

- aritméticos, asignación e incrementales

  `+ - * / % ^ ** = += /= %= ^= ++ --`

- la concatenación de cadenas se realiza sin ningún tipo de operador, escribiendo una a continuación de la otra
- booleanos (not, o, y)

  `! || &&`

1.3. AWK

arrays

- awk incorpora los arrays como mecanismo para almacenar números o grupos de cadenas que presentan algún tipo de relación
- Son **asociativos**. Esto es, están formados por un conjunto de pares: índice, valor asociado. Cualquier cadena o número puede ser un índice
  No es necesario especificar el tamaño de un array antes de empezar a usarlo, pudiendo añadirse nuevos pares (índice, valor) conforme se necesitan.
- awk crea automáticamente el array en la asignación del primer valor

```
array[indice] = valor
a["hola"]= 10   cadena[1]=$2
```

- Para referirse a un elemento de un array…

```
array[indice]
```

- Expresión para comprobar si un subíndice particular en el array…

```
subíndice in array
```

1.3. AWK

arrays (ii)

- La sentencia **delete** permite eliminar un elemento individual de un array…

```
delete array[indice]
```

- Para recorrer de todos los elementos de un array se utiliza la variante **for in** de la sentencia de control **for** (más adelante documentada)

```
for (variable in array)
    cuerpo
```

```
#imprimiendo el contenido de un array...
for (s in cadena) print s,cadena[s]
```


1.3. AWK

sentencias de control… if-else

- sentencia para la toma de decisiones en awk

if (condición) cuerpo_then [**else** cuerpo_else]

- También podría utilizarse la expresión condicional…

selector **?** if-true-exp **:** if-false-exp

```
#ejemplo if-else
BEGIN {cont=0}
$2<5 {cont++}
END { if (cont!=0)
        print "Número de alumnos suspensos: " cont}


#ejemplo expresión condicional
BEGIN {suspensos=0; aprobados=0}
{$2<5? suspensos++ : aprobados++}
END { print "Suspensos: " suspensos
        "Aprobados: " aprobados}
```

1.3. AWK

sentencias de control… while / do-while

- la sentencia **while** ejecuta repetidamente una sentencia o grupo de sentencias (cuerpo) mientras la condición planteada sea cierta

  while (condición) {cuerpo}
- la sentecia **do-while** ejecuta el cuerpo mientras la condición sea cierta igualmente. A diferencia de la anterior sentencia, la evaluación de la condición es posterior a la ejecución del cuerpo

  do cuerpo **while** (condición)

```
# suponer que de cada alumno se dispone de varias notas y
# se quiere presentar la nota media
{  sum = 0; i = 2;
    while(i <= NF){
        sum+= $i;
        i++;
     }
     media = sum/(NF -1);
     print “La nota media de “ $1 “es: “ media;
}
```


1.3. AWK

sentencias de control… for

- Presenta la siguiente forma:

  for (sentencia_inicialización; condición; incremento)cuerpo

```
#de forma equivalente...
{  sum = 0;
    for(i=2; i <= NF; i++) sum += $i;
    media = sum/(NF -1);
    print “La nota media de “ $1 “es: “ media;
}
```

1.3. AWK

sentencias de control… break, continue, next y exit

- **break** sale del bucle (for, while, do-while) más interno en el que está contenido
- **continue** fuerza la ejecución de la siguiente iteración "saltándose" las sentencias que falten por ejecutarse dentro del cuerpo del bucle
- **next** fuerza a awk a interrumpir el procesamiento del registro actual para "saltar" al siguiente registro de la entrada
- **exit** hace que awk detenga la ejecución de la regla actual y el procesamiento de la entrada. Si existe, antes de finalizar, ejecuta la regla END.

1.3. AWK

funciones

- awk permite al usuario definir sus propias funciones
- La definición de las funciones puede aparecer en cualquier parte entre las reglas de un programa awk. No existe la necesidad de poner la definición de la función antes de usarla
- La estructura general de la definición de una función es:

  function *nombre_función*(*lista_de_parámetros*) {
  cuerpo-de-la-función
  }
- No puede existir carácter alguno entre *nombre_función* y el paréntesis que delimita la *lista_de_parámetros*
- La invocación de la función sería:

```
nombre_función (lista_de_valores)
```

- awk soporta la **recursividad**

1.3. AWK

funciones (ii)

```
function factorial(n){
        if (n==0) return 1;
        else return n*factorial(n-1)}
```

```
# programa que presenta el factorial de una serie de
# números almacenados en un fichero
if (NF != 0){
        i = 1;
        do {
                print factorial($i);
                i++;
        }while(i < NF)
    }
}
```