TEMA 9: SHELL SCRIPTING EN #!/bin/sh

Basado en: Tema 9 (diapositivas) y "Fundamentos de sistemas operativos: una aproximación práctica usando Linux"

PARTE I: CONCEPTOS FUNDAMENTALES

1. ¿QUÉ ES UN SCRIPT DE SHELL?

Un **script de shell** es un fichero de texto que contiene una serie de comandos que la shell ejecuta secuencialmente.

Estructura mínima:

```
#!/bin/sh
# Comentario: esto no se ejecuta
echo "Hola mundo"
exit 0
```

Requisitos:

- Primera línea: `#!/bin/sh` (shebang) - indica qué intérprete usar.
- Permisos de ejecución: `chmod +x script.sh`
- Si no termina con `exit`, el script devuelve el status del último comando ejecutado.

Política POSIX: Los scripts con `#!/bin/sh` deben usar **solo características POSIX** (IEEE Std 1003.1-2017) para ser **portables** entre sistemas.

PARTE II: PARÁMETROS POSICIONALES

2. ACCESO A ARGUMENTOS

La shell proporciona variables especiales para acceder a los argumentos pasados al script:

Variable	Significado
`$0`	Nombre con el que se invocó el script
`$1` , `$2` , `$3` , ...	Argumentos posicionales (primero, segundo, tercero, ...)
`$#`	Número de argumentos (sin contar `$0`)
`$*`	Todos los argumentos como una única cadena: `"$1 $2 $3..."`
`$@`	Todos los argumentos separados: `"$1" "$2" "$3"...`

Ejemplo: `prueba_params.sh`

```
#!/bin/sh
echo "Nombre del script: $0"
echo "Número de parámetros: $#"
echo "Todos juntos (\$*): $*"
echo "---"

for arg in "$@"
do
    echo "  Argumento: $arg"
done

exit 0
```

Cómo probarlo:

```
chmod +x prueba_params.sh
./prueba_params.sh uno dos "tres cuatro"
```

Salida esperada:

```
Nombre del script: ./prueba_params.sh
Número de parámetros: 3
Todos juntos ($*): uno dos tres cuatro
---
  Argumento: uno
  Argumento: dos
  Argumento: tres cuatro
```

Diferencia importante entre `"$*"` y `"$@"` :

```
#!/bin/sh
# Con "$*": argumentos se tratan como una única palabra
for x in "$*"
do
    echo "palabra: $x"
done

# Con "$@": cada argumento es una palabra separada
for x in "$@"
do
    echo "palabra: $x"
done
```

3. COMANDO SHIFT

`shift` desplaza los argumentos posicionales: `$2` pasa a ser `$1` , `$3` pasa a ser `$2` , etc. Se actualiza `$#` .

Uso común: Procesar argumentos optativos sin complicar el resto del código.

Ejemplo: **`prueba_shift.sh`**

```
#!/bin/sh
echo "Antes de shift: $# argumentos -> $@"

shift
echo "Después de 1 shift: $# argumentos -> $@"

shift 2
echo "Después de 2 shifts más: $# argumentos -> $@"

exit 0
```

Cómo probarlo:

```
./prueba_shift.sh a b c d e
```

Salida esperada:

```
Antes de shift: 5 argumentos -> a b c d e
Después de 1 shift: 4 argumentos -> b c d e
Después de 2 shifts más: 2 argumentos -> d e
```

Patrón típico de parámetro optativo:

```
#!/bin/sh
debug=false

if test $# -gt 0 -a "$1" = "-d"
then
    debug=true
    shift  # Eliminamos -d de los argumentos
fi

# Ahora el resto del código solo trabaja con argumentos sin opciones
echo "debug = $debug"
echo "Otros argumentos: $@"
```

PARTE III: AGRUPACIONES DE COMANDOS

4. AGRUPACIONES EN SUBSHELL Y EN LA SHELL ACTUAL

Podemos agrupar comandos para ejecutarlos como una unidad.

4.1 Agrupación en **subshell** `( ... )`

Crea un nuevo proceso hijo. Los cambios de entorno (cd, export) no afectan a la shell padre.

```
#!/bin/sh
pwd
( cd /tmp && ls && pwd )  # Se ejecuta en subshell
pwd  # Seguimos en el directorio original
```

4.2 Agrupación en la **misma shell** `{ ... }`

Se ejecuta en el contexto de la shell actual. Los cambios de entorno SÍ afectan.

```
#!/bin/sh
pwd
{ cd /tmp; ls; pwd; }  # Se ejecuta en la misma shell
pwd  # Ahora estamos en /tmp (cambio persiste)
```

Ejemplo completo: `prueba_agrupaciones.sh`

```
#!/bin/sh

echo "=== SUBSHELL ==="
echo "Antes (pwd): $(pwd)"
(
    cd /tmp
    echo "Dentro subshell (pwd): $(pwd)"
)
echo "Después (pwd): $(pwd)"

echo ""
echo "=== MISMA SHELL ==="
echo "Antes (pwd): $(pwd)"
{
    cd /tmp
    echo "Dentro mismo shell (pwd): $(pwd)"
}
echo "Después (pwd): $(pwd)"

exit 0
```

Redirecciones con agrupaciones:

```
# Redirigir la salida de múltiples comandos
{ echo "línea 1"; echo "línea 2"; } > fichero.txt

# Procesar con filtros
{ cat a.txt; cat b.txt; } | sort | uniq
```

PARTE IV: SUSTITUCIÓN DE COMANDOS

5. SUSTITUCIÓN: EJECUTAR COMANDOS Y USAR SU SALIDA

Dos formas equivalentes:

```
variable=$(comando)   # Forma moderna (recomendada)
variable=`comando`    # Forma antigua (backticks)
```

Ejemplo: `prueba_substitucion.sh`

```
#!/bin/sh

# Guardar número de líneas de un fichero
lineas=$(wc -l < /etc/passwd)
echo "El archivo /etc/passwd tiene $lineas líneas"

# Usar sustitución en un argumento
echo "Contenido de /tmp:"
ls /tmp | wc -l
archivos=$(ls /tmp | wc -l)
echo "Total: $archivos ficheros en /tmp"

# Sustitución anidada
primero=$(echo "hola mundo" | cut -d' ' -f1)
echo "Primera palabra: $primero"

exit 0
```

Prueba:

```
./prueba_substitucion.sh
```

PARTE V: CONTROL DE FLUJO: if, elif, else, fi

6. SENTENCIAS CONDICIONALES IF

Las condiciones se basan en el **estado de salida de un comando**:

- Estado `0` = éxito = verdadero
- Estado ≠ `0` = fallo = falso

```
if comando
then
    comandos_si_éxito
elif comando
then
    comandos_si_éxito_segunda_condición
else
    comandos_si_fallo
fi
```

Ejemplo simple: `prueba_if.sh`

```
#!/bin/sh

fichero="/etc/hostname"

if test -f "$fichero"
then
    echo "El fichero $fichero existe"
    cat "$fichero"
else
    echo "El fichero $fichero no existe"
fi

exit 0
```

Negar condiciones con `!`:

```
if ! grep -q "usuario" /etc/passwd
then
    echo "El usuario no existe"
fi
```

7. COMANDO TEST Y OPERADORES

`test` (o `[ ... ]`) comprueba condiciones:

Pruebas de ficheros:

```
test -f fichero    # ¿Existe y es fichero regular?
test -d directorio # ¿Existe y es directorio?
test -e ruta       # ¿Existe (fichero o directorio)?
test -s fichero    # ¿Fichero existe y tiene tamaño > 0?
```

Pruebas de cadenas:

```
test -z cadena     # ¿Cadena vacía?
test -n cadena     # ¿Cadena no vacía?
test "$a" = "$b"   # ¿Cadenas iguales?
test "$a" != "$b"  # ¿Cadenas distintas?
test -n "$var"     # ¿Variable definida?
```

Pruebas de enteros:

```
test $a -eq $b     # ¿Iguales?
test $a -ne $b     # ¿Distintos?
test $a -gt $b     # ¿Mayor que?
test $a -ge $b     # ¿Mayor o igual?
test $a -lt $b     # ¿Menor que?
test $a -le $b     # ¿Menor o igual?
```

Ejemplo: `prueba_test.sh`

```
#!/bin/sh

# Pruebas de ficheros
if test -f "/etc/passwd"
then
    echo "/etc/passwd es un fichero regular"
fi

# Pruebas de cadenas
nombre="Juan"
if test -n "$nombre"
then
    echo "La variable nombre contiene: $nombre"
fi

# Pruebas de enteros
edad=25
if test $edad -ge 18
then
    echo "$edad años: ¡Mayor de edad!"
fi

# Prueba: ¿fichero no está vacío?
if test -s "/var/log/syslog"
then
    echo "El log /var/log/syslog tiene contenido"
fi

exit 0
```

Sintaxis alternativa con `[ ... ]`:

```
# Equivalentes:
test -f fichero
[ -f fichero ]

# En un if:
if [ -f fichero ]
then
    echo "Existe"
fi
```

8. SENTENCIAS CASE

`case` compara una palabra contra **patrones de globbing**:

```
case palabra in
patrón1)
    comandos
    ;;
patrón2 | patrón3)
    comandos (múltiples patrones)
    ;;
*)
    comandos_default
    ;;
esac
```

Ejemplo: `prueba_case.sh`

```
#!/bin/sh

archivo="documento.txt"

case "$archivo" in
*.txt)
    echo "Es un fichero de texto"
    ;;
*.pdf)
    echo "Es un PDF"
    ;;
*.sh)
    echo "Es un script de shell"
    ;;
*)
    echo "Tipo desconocido"
    ;;
esac

exit 0
```

PARTE VI: BUCLES

9. BUCLE WHILE

Se ejecuta mientras el comando devuelve éxito (status 0):

```
while comando
do
    comandos
done
```

Ejemplo: `prueba_while.sh`

```
#!/bin/sh

contador=1
while test $contador -le 5
do
    echo "Iteración $contador"
    contador=$((contador + 1))
done

echo "Fin"
exit 0
```

10. BUCLE FOR

Itera sobre una lista de palabras:

```
for variable in palabra1 palabra2 palabra3
do
    comandos  # Usa $variable
done
```

Ejemplo: `prueba_for.sh`

```
#!/bin/sh

# Iterar sobre literales
for fruta in manzana plátano cereza
do
    echo "Fruta: $fruta"
done

# Iterar sobre sustitución de comando
echo "Números del 1 al 5:"
for num in $(seq 1 5)
do
    echo "  - $num"
done

# Iterar sobre argumentos
echo "Argumentos recibidos:"
for arg in "$@"
do
    echo "  - $arg"
done

exit 0
```

PARTE VII: LECTURA DE ENTRADA: read

11. COMANDO READ

`read` lee una línea de la entrada estándar y la guarda en una variable.

```
read variable
```

Ejemplo: `prueba_read.sh`

```
#!/bin/sh

echo "¿Cuál es tu nombre?"
read nombre
echo "¡Hola, $nombre!"

exit 0
```

Read en bucle (leer fichero línea a línea):

```
while read linea
do
    echo "Línea leída: $linea"
done < fichero.txt
```

Ejemplo completo: `prueba_read_bucle.sh`

```
#!/bin/sh

# Crear fichero de prueba
cat > /tmp/datos.txt << 'EOF'
línea uno
línea dos
línea tres
EOF

echo "Leyendo /tmp/datos.txt:"
while read linea
do
    echo "  -> $linea"
done < /tmp/datos.txt

rm /tmp/datos.txt
exit 0
```

Cuidado: Si usas `read` dentro de un bucle con redirección, ten cuidado con la entrada de otros comandos en el bucle.

PARTE VIII: VARIABLE IFS (Internal Field Separator)

12. IFS: SEPARADOR DE CAMPOS

`IFS` define qué caracteres se usan como **separadores de campos**. Por defecto: espacio, tabulador, salto de línea.

Ejemplo: `prueba_ifs.sh`

```
#!/bin/sh

datos="uno:dos:tres:cuatro"

echo "Con IFS por defecto (espacio):"
for palabra in $datos
do
    echo "  - $palabra"
done

echo ""
echo "Con IFS = ':':"
IFS=:
for palabra in $datos
do
    echo "  - $palabra"
done

exit 0
```

Cuidado: Cambiar `IFS` afecta a todo el script. **Usa subshell** para cambios locales:

```
#!/bin/sh

datos="uno:dos:tres"

# Cambio LOCAL en subshell
(
    IFS=:
    for palabra in $datos
    do
        echo "  - $palabra"
    done
)

# IFS original aquí
```

PARTE IX: FUNCIONES

13. DEFINICIÓN Y USO DE FUNCIONES

Las funciones agrupan código reutilizable. Los parámetros se acceden como `$1`, `$2`, etc.

```
nombre_funcion() {
    # comandos
    # Acceso a parámetros: $1, $2, $#, etc.
}

# Ejecutar función
nombre_funcion arg1 arg2
```

Ejemplo: `prueba_funciones.sh`

```
#!/bin/sh

# Definir función
saludar() {
    nombre=$1
    edad=$2
    echo "¡Hola, $nombre! Tienes $edad años."
}

# Llamar función
saludar "Juan" "25"
saludar "María" "30"

exit 0
```

Funciones con variables locales (shadowing):

```
#!/bin/sh

variable_global="GLOBAL"

mi_funcion() {
    variable_global="LOCAL"  # Oculta la global dentro de la función
    echo "Dentro de la función: $variable_global"
}

echo "Antes: $variable_global"
mi_funcion
echo "Después: $variable_global"  # ¡Cambio PERSISTE!
```

PARTE X: OPERACIONES ARITMÉTICAS

14. CÁLCULOS CON NÚMEROS

```
resultado=$((expresión))
```

Ejemplo: `prueba_aritmetica.sh`

```
#!/bin/sh

a=10
b=3

suma=$((a + b))
resta=$((a - b))
producto=$((a * b))
division=$((a / b))
modulo=$((a % b))

echo "a = $a, b = $b"
echo "Suma: $suma"
echo "Resta: $resta"
echo "Producto: $producto"
echo "División: $division"
echo "Módulo: $modulo"

# Incrementar variable
contador=0
contador=$((contador + 1))
echo "contador = $contador"

exit 0
```

PARTE XI: EXPRESIONES REGULARES

15. CONCEPTOS DE EXPRESIONES REGULARES

Una **expresión regular** describe un **lenguaje formal** (conjunto de cadenas).

Caracteres especiales:

Carácter	Significado
`.`	Cualquier carácter
`[abc]`	Uno de a, b o c

Carácter	Significado
`[a-z]`	Rango: de a a z
`[^abc]`	Cualquiera EXCEPTO a, b, c
`^`	Principio de línea
`$`	Final de línea
`*`	Cero o más veces el anterior
`+`	Una o más veces el anterior
`?`	Cero o una vez el anterior
`|`	Alternativa (o)
`( )`	Agrupación
`\`	Escape (carácter literal)

Ejemplos:

```
a.c       → Encaja: aac, abc, aXc (cualquier carácter en el medio)
[0-9]+    → Encaja: 1, 123, 9999 (uno o más dígitos)
[a-z]*    → Encaja: "", a, abc, xyz (cero o más minúsculas)
^inicio   → Encaja si la línea COMIENZA con "inicio"
fin$      → Encaja si la línea TERMINA con "fin"
(ab)+     → Encaja: ab, abab, abababab
hola|adiós→ Encaja: hola o adiós
```

PARTE XII: GREP Y EGREP

16. COMANDO EGREP: BÚSQUEDA CON EXPRESIONES REGULARES

`egrep` es un filtro que escribe las líneas que encajan con una expresión regular.

Sintaxis:

```
egrep [opciones] 'expresión_regular' [fichero...]
```

Opciones útiles:

Opción	Significado
`-v`	Invertir: líneas que no encajan
`-n`	Mostrar número de línea
`-q`	Silencioso (solo status de salida)
`-i`	Ignorar mayúsculas/minúsculas
`-c`	Contar líneas que encajan

Ejemplos de uso:

```
# Líneas con un número de 2 dígitos
egrep '[0-9][0-9]' datos.txt

# Comprobar si hay alguna línea que empiece por ERROR
if egrep -q '^ERROR' log.txt
then
    echo "Hay errores"
fi

# Filtrar líneas que NO contengan la palabra tmp
egrep -v 'tmp' /etc/fstab
```

Ejemplo completo: `prueba_egrep.sh`

```
#!/bin/sh

# Crear fichero de prueba
cat > /tmp/test.txt << 'EOF'
Juan 25
María 30
Pedro 22
Ana 28
EOF

echo "=== Líneas con números de 2 dígitos ==="
egrep '[0-9]{2}' /tmp/test.txt

echo ""
echo "=== Nombres que empiezan con J o M ==="
egrep '^[JM]' /tmp/test.txt

echo ""
echo "=== Contar líneas ==="
egrep -c '' /tmp/test.txt

rm /tmp/test.txt
exit 0
```

PARTE XIII: SED - STREAM EDITOR

17. COMANDO SED: EDITOR DE FLUJOS

`sed` es un **editor de flujos**: aplica comandos a cada línea de entrada.

Sintaxis básica:

```
sed 'COMANDOS' fichero
```

Comandos importantes:

Comando	Significado
`s/viejo/nuevo/`	Sustituir la primera coincidencia
`s/viejo/nuevo/g`	Sustituir todas las coincidencias
`d`	Borrar la línea
`p`	Imprimir la línea (útil con `-n`)
`q`	Salir

Selección de líneas (direcciones):

```
sed '3d' fichero              # borrar línea 3
sed '1,5d' fichero            # borrar líneas 1 a 5
sed '/ERROR/d' fichero        # borrar las que contienen ERROR
sed '/^#/d' fichero           # borrar comentarios que empiezan por #
sed '5,10s/foo/bar/g' fichero # solo líneas 5–10
```

Ejemplos de uso:

```
# Cambiar 'mundo' por 'universo'
sed 's/mundo/universo/' mensaje.txt

# Cambiar todas las apariciones
sed 's/mundo/universo/g' mensaje.txt

# Eliminar líneas en blanco
sed '/^$/d' mensaje.txt
```

Ejemplo completo: `prueba_sed.sh`

```
#!/bin/sh

# Crear fichero de prueba
cat > /tmp/datos.txt << 'EOF'
hola mundo
adiós mundo
hola amigos
mundo mundo
EOF

echo "=== Original ==="
cat /tmp/datos.txt

echo ""
echo "=== Reemplazar 'mundo' por 'universo' (primera ocurrencia por línea) ==="
sed 's/mundo/universo/' /tmp/datos.txt

echo ""
echo "=== Reemplazar 'mundo' por 'universo' (todas en cada línea) ==="
sed 's/mundo/universo/g' /tmp/datos.txt

echo ""
echo "=== Borrar líneas que contienen 'adiós' ==="
sed '/adiós/d' /tmp/datos.txt

rm /tmp/datos.txt
exit 0
```

Sustituciones con backreferences (referencias hacia atrás):

Con `sed -E` (expresiones regulares extendidas) y agrupaciones `(...)`, puedes referir lo que encajó:

```
echo "Juan 25" | sed -E 's/([A-Za-z]+)[ ]+([0-9]+)/Nombre: \1, Edad: \2/'
# → Nombre: Juan, Edad: 25
```

PARTE XIV: AWK - PROCESADOR DE TEXTO POR CAMPOS

18. COMANDO AWK: LENGUAJE DE PROCESAMIENTO

`awk` es un lenguaje completo pensado para procesar texto por columnas.

Estructura general:

```
awk 'patrón { acciones }' fichero
```

Variables predefinidas:

Variable	Significado
`$0`	Línea completa
`$1, $2, ...`	Campos de la línea
`NF`	Número de campos
`NR`	Número de línea (registro actual)
`FS`	Separador de campos (por defecto: espacios/tabuladores)

Separador de campos:

Por defecto, espacios y tabuladores. Se puede cambiar con `-F` :

```
awk -F: '{print $1}' /etc/passwd    # separador ':'
awk -F, '{print $2}' datos.csv      # separador ','
```

Ejemplos básicos:

```
# Imprimir primera columna
awk '{ print $1 }' datos.txt

# Usar separador coma (CSV) y sacar columna 3 y 2
awk -F, '{ printf("%s\t%s\n", $3, $2) }' datos.csv

# Mostrar nombre y tamaño de ficheros
ls -l | awk '{ printf("Size:%08d bytes\tFichero:%s\n", $5, $9) }'
```

Filtrar por patrón:

```
# Líneas que contienen ERROR
awk '/ERROR/ { print $0 }' log.txt

# Solo líneas 5 a 10
awk 'NR >= 5 && NR <= 10 { print $0 }' fichero

# Solo las que tengan más de 3 campos
awk 'NF > 3 { print $0 }' fichero
```

Inicialización y finalización:

```
awk '
BEGIN { suma = 0 }
      { suma += $1 }
END   { printf("Suma: %d\n", suma) }
' numeros.txt
```

Arrays asociativos (contar repeticiones):

```
awk '
{ dups[$0]++ }
END {
    for (linea in dups) {
        print dups[linea], linea
    }
}' datos.txt
```

PARTE XV: RECORRER ÁRBOLES DE DIRECTORIOS

19. COMANDO DU: USO DE DISCO

`du` informa del uso de disco. Con `-a` muestra todos los ficheros y directorios por debajo de una ruta.

```
du -a .           # listar tamaños de todo el árbol bajo el directorio actual
du -ah /var/log   # lo mismo, con tamaños "humanos"
```

Combinado con otros filtros:

```
# Ver los 10 elementos más "pesados"
du -a . | sort -n | tail -n 10
```

20. COMANDO FIND: BÚSQUEDA Y ACCIONES

`find` recorre un árbol aplicando pruebas y acciones.

Patrón típico:

```
find RUTA [pruebas] [acciones]
```

Ejemplos:

```
# Todos los ficheros bajo .
find . -type f

# Ficheros .c
find . -type f -name '*.c'

# Directorios vacíos
find . -type d -empty

# Ficheros modificados en los últimos 7 días
find . -type f -mtime -7
```

Con `-exec` para ejecutar comandos:

```
# Borrar todos los .o
find . -type f -name '*.o' -exec rm -f {} \;

# Cambiar permisos a todos los scripts .sh
find . -type f -name '*.sh' -exec chmod u+x {} \;
```

PARTE XVI: FILTROS Y HERRAMIENTAS DE TEXTO

21. SORT: ORDENAR LÍNEAS

```
sort fichero                # Orden alfanumérico
sort -n fichero             # Orden numérico
sort -k2 -t: fichero        # Ordenar por campo 2 con separador ':'
sort -u fichero             # Ordenar y eliminar duplicados
```

22. UNIQ: ELIMINAR DUPLICADOS

```
uniq fichero                # Eliminar duplicados contiguos
uniq -c fichero             # Contar duplicados
uniq -d fichero             # Solo las líneas repetidas
```

Normalmente se combina con `sort`:

```
sort palabras.txt | uniq -c | sort -nr
```

23. TAIL Y HEAD: PRIMERAS Y ÚLTIMAS LÍNEAS

```
head -n 5 fichero            # Primeras 5 líneas
tail -n 10 fichero           # Últimas 10 líneas
tail -f /var/log/syslog      # Seguir un log en tiempo real
```

24. CUT Y PASTE: CAMPOS Y COLUMNAS

```
cut -d: -f1,3 /etc/passwd      # campos 1 y 3, separador ':'
paste nombres.txt notas.txt    # pegar columnas de dos ficheros
```

25. TR: TRADUCCIÓN DE CARACTERES

`tr` traduce o elimina caracteres:

```
tr 'viejo' 'nuevo'   # Traducir cada carácter
tr -d 'caracteres'   # Borrar caracteres
tr 'a-z' 'A-Z'       # Convertir a mayúsculas
```

Ejemplo: `prueba_tr.sh`

```
#!/bin/sh

texto="Hola Mundo 123"

echo "Original: $texto"
echo "Mayúsculas: $(echo "$texto" | tr 'a-z' 'A-Z')"
echo "Minúsculas: $(echo "$texto" | tr 'A-Z' 'a-z')"
echo "Sin dígitos: $(echo "$texto" | tr -d '0-9')"
echo "Espacios por '_': $(echo "$texto" | tr ' ' '_')"

exit 0
```

26. JOIN: INNER JOIN RELACIONAL

`join` hace un inner join entre dos ficheros ordenados por la clave.

Requisitos:

- Ambos ficheros deben estar ordenados por la columna usada como clave.
- Por defecto, usa la primera columna, con separadores de espacio/tabulador.

Ejemplo:

```
echo 'a bla
b ble
c blo' > a.txt

echo 'a ta
b te
c to' > b.txt

join a.txt b.txt
# a bla ta
# b ble te
# c blo to
```

Si no están ordenados:

```
sort a.txt -o a.txt
sort b.txt -o b.txt
join a.txt b.txt
```

Opciones útiles:

- `-1 N` : columna de clave en el primer fichero
- `-2 N` : columna de clave en el segundo
- `-t SEP` : separador de campos

PARTE XVII: REDIRECCIONES Y PIPES

27. REDIRECCIONES

```
comando > fichero       # Salida estándar a fichero (trunca)
comando >> fichero      # Salida estándar al final (append)
comando 2> fichero      # Salida de error a fichero
comando 2>&1            # Salida de error a salida estándar
comando < fichero       # Entrada desde fichero
```

28. PIPES (TUBERÍAS)

```
comando1 | comando2     # Salida de comando1 → Entrada de comando2
comando1 | comando2 | comando3  # Encadenar comandos
```

Ejemplo: `prueba_pipes.sh`

```
#!/bin/sh

echo "=== Pipeline simple ==="
echo -e "manzana\nplátano\ncereza" | sort

echo ""
echo "=== Pipeline complejo ==="
echo -e "3\n1\n4\n1\n5" | sort -n | uniq -c | sort -rn

echo ""
echo "=== Redireccionamiento ==="
echo "Error de prueba" 2> /tmp/error.txt
echo "Contenido de error.txt:"
cat /tmp/error.txt
rm /tmp/error.txt

exit 0
```

29. XARGS: CONSTRUCCIÓN DE COMANDOS

`xargs` construye y ejecuta comandos a partir de la entrada estándar.

Sintaxis típica:

```
comando_que_lista | xargs otro_comando
```

Ejemplos:

```
# Ver en una columna los ficheros a, b, c
echo a b c | xargs ls -1

# Borrar todos los .o bajo el árbol
find . -type f -name '*.o' -print | xargs rm -f

# Buscar una cadena en todos los .c
find . -type f -name '*.c' -print | xargs egrep 'main\('
```

Con nombres con espacios (variante "segura" -print0 / -0):

```
find . -type f -name '*.c' -print0 | xargs -0 egrep 'main\('
```

RESUMEN RÁPIDO DE CONSTRUCCIONES

```
# Parámetros
$0, $1, $2, ..., $#, "$@", "$*"

# Control de flujo
if [ -f fichero ]; then ... fi
case $var in patrón) ... ;; esac

# Bucles
for var in lista; do ... done
while comando; do ... done

# Sustitución
$(comando)
`comando`  # Forma antigua

# Aritméticas
$((a + b))

# Variables especiales test
test -f fichero
test -d directorio
test "$a" = "$b"
test $a -eq $b

# Funciones
nombre() { ... }

# Salida estándar y error
> fichero, >> fichero, 2> fichero, 2>&1

# Pipes
cmd1 | cmd2 | cmd3

# Expresiones regulares con egrep
egrep 'patrón' fichero

# Sed
sed 's/viejo/nuevo/g' fichero
sed '/patrón/d' fichero
```

```
# Awk
awk '{ print $1 }' fichero
awk -F: '{ print $3 }' fichero

# Filtros
sort, uniq, tail, head, tr, grep, sed, awk, cut, paste, join

# Utilitarios
du, find, xargs
```