Indice

1. ¿Qué son y para qué sirven las utilerías de Unix? 1
2. Repaso de tuberías y redireccionamientos 1
3. Filtros para extracción y presentación de información 2
    - 3.1. grep 2
        - 3.1.1. Expresiones Regulares 2
        - 3.1.2. fgrep 3
        - 3.1.3. egrep 4
    - 3.2. sort 4
    - 3.3. cut 6
    - 3.4. uniq 7
    - 3.5. tr 7
    - 3.6. head 8
    - 3.7. tail 8
4. Cálculos y conteos 9
    - 4.1. expr 9
    - 4.2. wc 10
5. Comparación de archivos 10
    - 5.1. diff 10
    - 5.2. Variantes de diff: bdiff, sdiff, diff3 11
        - 5.2.1. bdiff 11
        - 5.2.2. sdiff 11
        - 5.2.3. diff3 12
    - 5.3. comm 13
    - 5.4. cmp 13
6. sed 13
7. awk 18
    - 7.1. Historia 18
    - 7.2. Tutorial 18
        - 7.2.1. El primer programa en awk 18
        - 7.2.2. La estructura de un programa en awk 19
        - 7.2.3. Cómo se ejecuta un programa en awk 20
        - 7.2.4. Errores 20
        - 7.2.5. Salida de datos sin formato 20
        - 7.2.6. Salida de datos con formato 22
        - 7.2.7. Selección de datos 23
        - 7.2.8. Realización de operaciones 24
        - 7.2.9. Control de flujo 26
        - 7.2.10. Arreglos 27
        - 7.2.11. Programas sencillos pero útiles 27
    - 7.3. Estructura del lenguaje 29
        - 7.3.1. Sumario de patrones 29

7.3.2. Operadores de comparación 29
7.3.3. Operadores de coincidencia de cadenas 30
7.3.4. Expresiones regulares 30
7.3.5. Secuencias de escape (caracteres especiales) 30
7.3.6. Sumario de expresiones regulares 31
7.3.7. Resúmen de acciones 31
7.3.8. Variables automáticas 32
7.3.9. Funciones aritméticas incluidas 32
7.3.10. Funciones de cadena incluidas 33
7.3.11. Operadores en orden de precedencia creciente 34
7.3.12. Enunciados de control de flujo 35
7.3.13. Definición de funciones del usuario 35
7.3.14. Enunciados de salida 36
7.3.15. Caracteres de formato de printf 36
7.3.16. Ejemplos de especificaciones de printf 37
7.3.17. Funciones de entrada de datos 37
7.3.18. Interpretación de argumentos de la línea de comandos 37

Notas del curso de Utilerías de Unix

Por Diego Martín Zamboni

1. ¿Qué son y para qué sirven las utilerías de Unix?

El sistema operativo Unix, desde su concepción, tiene la filosofía de la especialización: está compuesto por muchos pequeños programas, cada uno de los cuales realiza una función muy específica. De hecho, son pocos los comandos de Unix que realizan más de una función en particular. Unix provee los medios necesarios para integrar distintos comandos, aprovechando sus capacidades conjuntas para llevar a cabo tareas más complejas de una manera relativamente sencilla.

Existen algunos comandos que son muy utilizados por ser los más "útiles" para la obtención de resultados rápidamente. En este texto trataremos de dar una visión muy general de los comandos más populares entre los usuarios de Unix.

Cabe hacer notar que describiré cada comando de la manera más general posible. Quizás algunos de ellos presenten ciertos cambios dependiendo de la versión de Unix en que se estén ejecutando. Para mayor información sobre cualquiera de los comandos que serán mencionados, recomiendo referirse a la página de manual correspondiente (ya sea en los manuales impresos o en línea, con el comando **`man` *`comando`***).

Estas notas suponen que el que las consulta tiene un conocimiento básico del sistema operativo Unix y de la utilización de los comandos fundamentales.

2. Repaso de tuberías y redireccionamientos

Algunos conceptos muy importantes en Unix y que son fundamentales para la utilización de la gran mayoría de las utilerías que se verán a continuación son los de *filtros*, *tuberías* y *redireccionamientos*.

Muchos programas en Unix están diseñados como filtros, es decir, aceptan los datos de un dispositivo conocido como *entrada estándar*(que normalmente es el teclado) y envían sus resultados a otro dispositivo conocido como *salida estándar* (que normalmente es la pantalla). Un programa que opera de esta manera puede ser utilizado como filtro, de la manera que se explica a continuación.

Un *redireccionamiento* es una característica de Unix que nos permite cambiar el dispositivo asociado ya sea a la salida estándar o a la entrada estándar de un programa. Por ejemplo, si se redirecciona la salida estándar de un comando hacia un archivo, lo que el programa imprima no aparecerá en la pantalla, sino que se escribirá al archivo que le indicamos. El símbolo para indicar un redireccionamiento de salida estándar en Unix es el mayor-que (>). Por ejemplo:

`ls -l > listado` ⇐ Manda el directorio a un archivo llamado `listado`

De manera similar, si redireccionamos la entrada estándar de un comando hacia un archivo, dicho comando tomará los datos que necesite de ese archivo en vez de esperar su introducción desde el teclado. El símbolo que se utiliza en Unix para indicar un redireccionamiento de entrada estándar es el menor-que (<). Por ejemplo:

`cat < datos` ⇐ Toma el contenido del archivo `datos` y lo imprime en la pantalla.

Ambos redireccionamientos pueden combinarse en un solo comando. Por ejemplo:

`cat < datos > datos2` ⇐ Toma el contenido del archivo `datos` y lo envía a un archivo llamado `datos2`

Una *tubería* (pipe en inglés) sirve para redireccionar la salida estándar de un comando y automáticamente convertirla en la entrada estándar de otro. De esta manera es posible "ligar" comandos, de tal manera que los resultados producidos por uno de ellos automáticamente pasan a otro para sufrir algun otro procesamiento. El símbolo utilizado en Unix para indicar una tubería es la barra vertical (|). Por ejemplo:

`ls -la | more` ⇐ Convierte el listado del directorio en la entrada estándar para el comando `more`

3. Filtros para extracción y presentación de información

En esta sección examinaremos algunos de los filtros más utilizados en Unix para obtención, arreglo y presentación de datos.

3.1. grep

grep es un comando con el que probablemente muchos ya estamos familiarizados. Su nombre significa Global Regular Expression Print, y su principal aplicación es la localización de cadenas en archivos de texto.

Lo que hace grep es tomar las líneas que le llegan de la entrada estándar, busca en ellas el patrón o patrones especificados e imprime las líneas que lo(s) contienen en la salida estándar.

Su sintáxis es la siguiente:

```
grep [opciones] patron [archivo ...]
```

Si se especifica uno o más archivo de entrada, el patrón se busca en las líneas de cada uno de los archivos especificados. Si no se especifica archivo, le búsqueda se realiza sobre las líneas recibidas de la entrada estándar. En caso de que se especifique más de un archivo, junto con cada línea en la que se encuentre la cadena buscada se imprimirá el archivo en el que se encuentra.

El patrón de búsqueda debe ponerse entre comillas en caso de que contenga espacios o caracteres con significado especial para el shell, como $, (,), etc. Es importante recalcar que el grep trabaja por líneas, es decir, no encontrará un patrón dado si ese patrón comienza en una línea y termina en la siguiente.

Las principales opciones de grep son:

-i Ignora la distinción entre mayúsculas y minúsculas.

-v En vez de imprimir las líneas que contienen el patrón, imprime aquellas que no lo contienen.

-c No imprime las líneas encontradas, solamente dice cuántas líneas contienen el patrón especificado.

-n Junto a cada línea encontrada imprime el número de dicha línea dentro del archivo.

-l Cuando se especifican múltiples archivos para la búsqueda, solamente imprime los nombres de los archivos que contienen la cadena buscada.

grep es muy útil para encontrar en los archivos cadenas fijas. Sin embargo, además de permitirnos especificar la cadena de texto exacta a buscar, grep nos permite especificar la búsqueda de patrones que tienen ciertas semejanzas o contienen partes comunes, aunque no sean exactamente iguales. A esta clase de patrones se les conoce como **expresiones regulares**.

3.1.1. Expresiones Regulares

La sintáxis de las expresiones regulares utiliza un conjunto de caracteres con significado especial para las búsquedas. A estos caracteres se les llama *metacaracteres*, o *wild card* en inglés. Los principales metacaracteres utilizados en expresiones regulares son:

El circunflejo (^): Este caracter sirve para hacer referencia al principio de la línea. Por ejemplo, la cadena `^hola` coincidirá con cualquier línea que comience con la palabra "hola", pero no la encontrará si aparece en cualquier otra parte de la línea.

El signo de pesos ($): Funciona de manera similar al circunflejo, pero hace referencia al final de la línea. Por ejemplo, `adios$` hace referencia a cualquier línea que termine con la palabra "adios".

El punto (.): Este caracter sirve como comodín, es decir, puede estar en lugar de cualquier otro caracter. Por ejemplo, `co.a` coincidiría con "cosa", "cota", "coma", "cola", etc.

El asterisco (*): Significa "cero o más ocurrencias del caracter anterior". Por ejemplo, `car*o` coincide con "cao", "caro", "carro", "carrro", etc. El asterisco se puede combinar con el punto; de esta manera, `.*` significa "cualquier secuencia de caracteres, incluyendo ninguno".

Los corchetes ([]): Sirven para indicar los caracteres específicos que se quiere que sean aceptados en una posición específica dentro de la cadena. Así, `[aeiou]` coincide con cualquiera de las vocales. Se pueden especificar rangos de caracteres dentro de los corchetes así como concatenar más de un rango y/o caracteres, de esta manera, `[a-z]` aceptaría cualquier letra minúscula del alfabeto, `[a-zA-Z]` encuentra cualquier letra minúscula o mayúscula, y `[a-z123]` aceptaría cualquier letra minúscula y cualquiera de los dígitos 1, 2 o 3.

Los corchetes con circunflejo ([^]): Tienen el mismo significado que los corchetes, pero coinciden con cualquier caracter **excepto** los que están entre los corchetes. Así, por ejemplo, `[^a-zA-Z0-9]` encontraría cualquier caracter que no sea una letra o un dígito.

En ocasiones, necesitaremos localizar dentro de un archivo alguna cadena que contenga o consista de algún metacaracter; en este caso no queremos que grep lo interprete como expresión regular, sino como un caracter literal. Para hacer esto, basta anteponer al caracter deseado una diagonal invertida (\). ·Este caracter se conoce como *caracter de escape*, y le indica al grep que el caracter que sigue debe ser tomado literalmente, sin ninguna clase de interpretación especial.

Cuando queremos utilizar el grep con expresiones regulares, es muy recomendable encerrar el patrón de búsqueda entre comillas. Esto se debe a que muchos metacaracteres de expresiones regulares también tienen significado especial para el shell, lo cual puede causar confusiones.

Ejemplos de grep:

`grep Casa lista`	Imprime todas las líneas del archivo **lista** que contengan en cualquier parte la cadena Casa.
`grep -v -i "^Hola todos"`	Encuentra en todos los archivos del directorio actual las líneas que no comiencen con la cadena "hola todos", sin importar si está en mayúsculas o minúsculas.
`grep "ch.*se" *.dat`	Imprime todas las líneas de los archivos cuyos nombres terminen con **.dat** que contengan los caracteres **ch**, después cualquier número de caracteres (incluyendo ninguno) y después los caracteres **se**. Por ejemplo: chinese, cheese, ch12345ABCse, etc.

Existen dos variantes de grep, **fgrep** y **egrep**, que funcionan básicamente de la misma manera pero tienen algunas características distintas.

3.1.2. fgrep

grep permite hacer una búsqueda de un texto fijo o expresión regular, pero solo se puede especificar un único patrón en una orden grep. **fgrep** permite buscar múltiples objetivos, que se pueden especificar en la línea de comandos o en un archivo. A cambio de esto, fgrep no permite la utilización de expresiones regulares, pero gracias a ellos las búsquedas son mucho más rápidas.

Para especificar las cadenas de búsqueda en la línea de comandos, se utiliza la siguiente sintáxis:

```
fgrep [opciones] "cadena1
cadena2
...
cadenaN" [archivo ...]
```

Obsérvese que las comillas se inician antes de la primera cadena y no se cierran sino hasta después de la última, y las cadenas están separadas entre sí por cambios de línea. El shell, al notar que no se han cerrado las comillas al final de la primera línea, nos proporciona un prompt distinto (>) que

indica que podemos seguir tecleando, y solo tratará de interpretar el comando cuando hayamos cerrado las comillas.

Otra manera de especificar las cadenas a buscar es guardarlas en un archivo, en una línea separada cada una. El fgrep acepta una nueva opción (aparte de todas las que acepta el grep) que es:

-f *archivo* Toma del archivo mencionado las cadenas de búsqueda.

3.1.3. egrep

Este comando incluye lo mejor de grep y fgrep, y añade aún más cosas. Como fgrep, permite buscar múltiples objetivos y extraerlos de un archivo. Como grep, permite la búsqueda de expresiones regulares, y añade algunos metacaracteres adicionales para realizar búsquedas extendidas.

Aparte de los metacaracteres normales del grep, egrep dispone de los siguientes:

El signo más (+): Funciona de manera similar al asterisco, pero indica una o más repeticiones del caracter anterior, en vez de cero o más como el asterisco.

El signo de interrogación (?): Funciona también de manera similar al asterisco, pero indica cero o una repeticiones del caracter anterior.

Para especificar varios patrones de búsqueda, se pueden colocar en líneas separadas, como en el fgrep, pero también se pueden separar con una barra vertical (|). De manera similar al fgrep, la opción -f permite especificar un archivo que contiene los patrones de búsqueda.

3.1.4. Ejemplos

`grep Hola lista`	Imprime todas las líneas del archivo *lista* que contengan la cadena Hola en cualquier posición.
`grep "[Hh]ola" *.txt`	Imprime las líneas de todos los archivos del directorio actual cuyo nombre termine con .txt, que contengan la cadena hola u Hola.
`grep -v "co.a" texto.doc`	Imprime todas las líneas del archivo `texto.doc` que no contengan una cadena que comience con `co`, después cualquier caracter, y al final una `a`.
`fgrep "exito↵ logro↵ triunfo" curriculum.txt`	Imprime todas las líneas del archivo `curriculum.txt` que contengan la cadena `exito`, `logro` o `triunfo` (↵ representa oprimir RETURN).
`egrep "file[01]?[0-9]" lista`	Imprime las líneas del archivo `lista` que contengan una cadena desde `file0` o `file00` hasta `file19`.
`egrep -vf Nombres alumnos`	Imprime las líneas del archivo `alumnos` que no contengan ninguno de los nombres contenidos en el archivo `Nombres`.
`grep -l "^[·\t]*A quien corresponda" *`	Imprime los nombres de los archivos que contengan la cadena `A quien corresponda` al principio de una línea, posiblemente precedida de espacios o tabuladores.

3.2. sort

Este comando nos permite ordenar el contenido de un archivo en de acuerdo a un criterio alfabético o numérico, ascendente o descendente, y de acuerdo a cualquier campo de la línea, que nosotros especifiquemos. Los datos ordenados son impresos en la salida estándar.

La sintáxis del comando sort es la siguiente:

```
sort [opciones] [+pos1 [-pos2]] [archivo ...]
```

Al igual que el grep, si no se especifican archivos de entrada, sort toma los datos de la entrada estándar. Si se especifica más de un archivo de entrada, se ordenan juntas todas las líneas de todos los archivos y se escriben juntas; no se hace el ordenamiento archivo por archivo.

Las principales opciones son:

-b Ignora los espacios y tabuladores sobrantes antes de la cadena a ordenar a la hora de hacer el ordenado.

-d Hace un ordenamiento "de diccionario", esto es, toma en cuenta solamente letras, números y espacios en blanco.

-f Toma mayúsculas y minúsculas como iguales al hacer el ordenamiento; normalmente se toman diferente, pues sus códigos ASCII son distintos.

-n Realiza el ordenamiento por orden numérico en vez de ASCII. Implica automáticamente la opción -b.

-r Realiza un ordenamiento descendiente en vez de ascendiente.

-t*x* Especifica *x* como el separador de campos.

-c Antes de comenzar el ordenamiento verifica si el archivo ya se encuentra ordenado; en caso de que así sea, termina sin realizar ninguna acción.

-m Si tenemos dos archivos ya ordenados como entrada, esta opción simplemente los mezcla, sin hacer ningún ordenado previo.

-u Después de hacer el ordenamiento elimina las líneas repetidas.

-o *archivo* Guarda la salida producida en el archivo especificado. Esta opción tiene una ventaja sobre redireccionar la salida estándar, que es que se puede especificar el mismo archivo de entrada, de manera que podemos ordenar un archivo dejando el resultado en el mismo archivo, sin necesidad de crear archivos intermedios.

+pos1 [-pos2] Estos parámetros sirven para especificar la llave que se va a utilizar para hacer el ordenamiento. Por default el ordenamiento se hace tomando en cuenta toda la línea, desde el primer caracter. Con estos parámetros se especifica que el ordenamiento se hará comenzando en el caracter pos1 y terminando en pos2. Si no se especifica pos2 se toma hasta el final de la línea. Tanto pos1 como pos2 tienen la forma **m.n**, donde m es el número de campo y n es un desplazamiento en caracteres dentro de ese campo. Sin embargo, m.n tiene distinto significado dependiendo de si se está utilizando en pos1 o en pos2:

- En pos1, m.n significa *el n-ésimo caracter a partir del primer caracter del campo (m+1)*. En español, se puede entender como "brincarse m campos, y a partir del primer caracter del campo en el que quedemos, brincar n caracteres".
- En pos2, m.n significa *el n-ésimo caracter a partir del último caracter del campo m*.

El ordenamiento se hará tomando en cuenta todos los caracteres entre pos1 y pos2, incluyendo los caracteres que se encuentran en esas posiciones. En ambos casos, si se omite `.n`, se asume que n vale cero.

3.2.1. Ejemplos

`sort /etc/passwd`

Ordena el archivo /etc/passwd con los parámetros de default, es decir, comenzando desde el primer caracter y haciendo un ordenamiento ascendente. Podríamos decir que esto hace un ordenamiento por nombre de cuenta, que es el primer campo de cada línea, pero este ordenamiento no sería totalmente correcto, porque estamos tomando en cuenta también el resto de los caracteres, incluyendo los separadores de campos.

El archivo ya ordenado se imprime a la pantalla.

`sort -t: -f +0 -1 /etc/passwd > sorted_password`	Ordena el archivo /etc/passwd tomando en cuenta únicamente el primer campo (el nombre de la cuenta) y sin hacer distinción entre mayúsculas y minúsculas, dejando el resultado en el archivo `sorted_password`. Obsérvese que tanto en +0 como en -1 se omite el elemento .n, por lo que el ordenamiento se hace en base a la cadena contenida entre el primer caracter del campo #1 y el último caracter de ese mismo campo.
`grep "/bin/sh$" /etc/passwd | sort -nr -t: +1 -2`	En este ejemplo se utiliza una tubería para combinar el `grep` con el `sort`. Primero se extraen todas las líneas de /etc/passwd que corresponden a usuarios que tienen el Bourne Shell como shell de default (el shell asignado es el último elemento de cada línea de /etc/passwd), y esas líneas de pasan a través de una tubería al `sort`, que las ordena en base al identificador de usuario, es decir, el segundo campo de cada línea. El ordenamiento se hace en forma descendente.
`sort -df > lista_ordenada`	Recibe datos del teclado (la entrada estándar), ordena las líneas introducidas sin tomar en cuenta mayúsculas y minúsculas y considerando solamente dígitos, letras y espacios en blanco (no se toman en cuenta signos de puntuación ni de otra especie), y se dejan los datos ordenados en el archivo `lista_ordenada`.

3.3. cut

El comando `grep` y sus variantes, vistos anteriormente, nos permiten seleccionar ciertos renglones de un archivo, de acuerdo a un criterio que nosotros establecemos. Sin embargo, nos vemos obligados a trabajar con renglones completos, que es el máximo nivel de selección que nos proporciona `grep`. En ocasiones no necesitamos toda la información presente en un renglón, sino solamente algunos elementos. Para ello existe el comando `cut`, que nos permite seleccionar partes de un archivo en sentido horizontal (el `grep` hace selecciones en sentido vertical), ya sea por caracteres o por campos.

Las dos sintáxis generales del comando `cut` son las siguientes:

`cut -c`*`lista`* [*`archivo ...`*]
`cut -f`*`lista`* [-d*`char`*] [-s] [*`archivo ...`*]

Al igual que los comandos anteriores, si se omiten los archivos de entrada, se toma la misma de la entrada estándar. Toda la salida se envía a la salida estándar.

Las opciones son:

lista — Una lista de enteros separados por comas (en orden numérico creciente), pudiendo especificarse rangos utilizando el guión (-). Por ejemplo: **1,4,7**; **1-3,8** (del 1 al 3 y el 8); **-5,10** (abreviación de **1-5,10**), **3-** (del 3 al último).

-c*lista* — La lista especifica posiciones de caracteres dentro de la línea. Por ejemplo, **-c1-72** imprimiría los primeros 72 caracteres de cada línea.

-f*lista* — La lista especifica números de campos separados por un cierto delimitador (ver la opción -d). Por ejemplo, **-f1,7** imprime el primero y séptimo campos de cada línea; **-f1-3,8** imprime los primeros 3 campos y el octavo. Cuando se seleccionan varios campos, se imprimen separados por el mismo delimitador seleccionado.

-d*char* — Selecciona el caracter especificado como separador de campos. Cuando el caracter utilizado tenga algún significado especial para el shell o sea un espacio es necesario encerrarlo entre apóstrofes. El separador por default es el tabulador.

-s Cuando se está haciendo una selección de campos, indica que las líneas que no contengan ningún caracter delimitador no sean impresas. Normalmente, dichas líneas son impresas en su totalidad, sin ningún cambio.

3.3.1. Ejemplos

```
cut -d: -f1,5 /etc/passwd
```
Lista las cuentas existentes en el sistema junto con el nombre real de sus dueños.

```
who am i | cut -f1 -d' '
```
Imprime el nombre de la cuenta en la que estoy trabajando.

```
grep "/bin/csh$" /etc/passwd |
sort -t: -rn +2 -3 | cut -d:
-f1,3,5
```
Selecciona los usuarios que tengan asignado como shell el C Shell, ordena dichos usuarios en orden descendiente de UID (identificador de usuario) e imprime el login, UID y nombre real de cada uno de ellos.

3.4. uniq

Este comando sirve para identificar líneas que estén repetidas en un archivo de texto. Para que funcione correctamente, las líneas iguales tienen que estar juntas, es decir, es necesario primero ordenar el archivo utilizando el comando `sort`. La sintáxis general de `uniq` es la siguiente:

`uniq [opciones] [`*`arch_entrada`* `[`*`arch_salida`*`]]`

Si se utiliza sin opciones, `uniq` solo imprime las líneas que no estén repetidas y la primera ocurrencia de las que lo estén. Si no se especifica archivo de entrada, los datos se toman de la entrada estándar, y si no se especifica archivo de salida, los resultados se mandan a la salida estándar. Cuando se especifican ambos archivos, forzosamente tienen que ser archivos distintos.

Las opciones principales son las siguientes:

-u Imprime solamente las líneas que no están repetidas en el archivo original.

-d Imprime una sola copia de las líneas repetidas (el modo normal de operación de `uniq` es la combinación de -u y -d).

-c Imprime, antes de cada línea, el número de veces que aparece.

-*n* Se ignoran los primeros *n* campos al hacer la comparación. Un campo se define como una cadena de caracteres "no blancos" (tabuladores o espacios) separados por tabuladores y/o espacios de sus vecinos. No hay manera de cambiar el delimitador de campos.

+*n* Se ignoran los primeros *n* caracteres al hacer la comparación. Se puede combinar esta opción con la anterior, y en ese caso primero se brincan los campos especificados y después los caracteres.

3.4.1. Ejemplos

```
cut -d: -f1-6 /etc/passwd | sort
-t: +5 | tr " :" ".\011" | uniq -5
```
Primero elimina el último campo del /etc/passwd, después ordena en base al último campo que queda (el directorio home de cada usuario), a continuación reemplaza los espacios por puntos y los dos puntos por tabuladores (código 11 octal), y finalmente reporta uno de cada grupo de usuarios que tengan asignado el mismo directorio home.

3.5. tr

Este comando sirve para convertir ciertos caracteres en otros, es decir, copia la entrada estándar a la salida estándar haciendo las sustituciones que le indiquemos. Es importante hacer notar que `tr`

siempre toma los datos de la entrada estándar, no hay manera de indicarle directamente un archivo, por lo que siempre tenemos que utilizar redireccionamientos o tuberías para proporcionarle la entrada.

Su sintáxis es la siguiente:

```
tr [opciones] [cadena1 [cadena2]]
```

La acción consiste en reemplazar los caracteres que se encuentren en *cadena1* por el caracter que esté en la posición correspondiente de *cadena2*. Las opciones que se pueden utilizar son:

-c Reemplaza por los caracteres de *cadena2* los caracteres que **no** se encuentren en *cadena1*.

-d Borra los caracteres que se encuentren en *cadena1*. En este caso, no hace falta poner *cadena2*.

-s Comprime todos los caracteres que se encuentren en *cadena2* y que queden repetidos en forma contigua a la salida a un solo caracter.

En las cadenas es posible utilizar ciertas abreviaturas para indicar rangos de caracteres y repeticiones de caracteres:

[a-z] Representa todos los caracteres de la **a** a la **z**, inclusive.

[a*n] Representa *n* repeticiones de **a**. Si el primer dígito de *n* es **0**, se considera *n* como un número octal, de lo contrario se interpreta como decimal. Si *n* vale cero o es omitido, significa "tantos como sean necesarios hasta alcanzar la longitud de *cadena1*".

También puede utilizarse el caracter \ seguido de un número octal de 1, 2 o 3 dígitos para representar el caracter con el código ASCII correspondiente. Si *cadena1* es más corta que *cadena2*, los caracteres de *cadena1* que no alcancen correspondencia son pasados sin alteración.

3.5.1. Ejemplos

```
tr "[a-z]" "[A-Z]" < arch1 > arch2
```

Convierte todas las letras minúsculas de `arch1` en mayúsculas, dejando el resultado en `arch2.`

```
tr -cs "[A-Z][a-z]" "[\012*]" <
arch1 > arch2
```

Convierte todos los caracteres que **no** sean letras mayúsculas o minúsculas en cambios de línea (código 12 octal), y suprime los cambios de línea que queden repetidos.

3.6. head

Este comando sirve para mostrar las primeras líneas de un archivo. Si se ejecuta el comando `head archivo`, por default obtendremos las 10 primeras líneas de dicho archivo. La única opción con que se cuenta es ***-n***, lo que provoca que se muestren las *n* primeras líneas del archivo. Por ejemplo: `head -20 /etc/hosts` nos muestra las 20 primeras líneas del archivo /etc/hosts.

3.7. tail

Funciona de manera muy similar al comando `head`, pero nos muestra las líneas al final del archivo en vez de las del principio. Su sintáxis es un poco más compleja, siendo la siguiente:

```
tail [±[n][unidad][[-]f] [archivo ...]
```

El significado de las opciones es el siguiente:

-n Se imprimen las *n* últimas líneas del archivo.

+n Se imprime el archivo a partir de la línea *n.*

unidad Es la unidad en la que se va a realizar el conteo, y por default son líneas. Esta opción consta de una letra, y puede ser **b** para bloques (cuyo tamaño depende del sistema, pero en la Cray son de 4096 bytes cada uno), **c** para caracteres o **l** para líneas (el default).

[-]f Imprime lo que se le haya especificado, pero al llegar al final del archivo, en vez de terminar el programa, se queda esperando y va imprimiendo cualquier cosa que le vaya siendo añadida al archivo. Esta opción es útil para monitorear el crecimiento de un archivo que está siendo escrito por algún otro proceso. El guión no tiene ningún significado especial, y solamente es necesario ponerlo cuando la **f** no se está colocando a continuación de alguna otra opción.

4. Cálculos y conteos

4.1. expr

Este comando nos sirve para evaluar expresiones aritméticas o de cadena desde el shell. Su sintáxis general es:

```
expr argumentos
```

Donde los *argumentos* son evaluados y el resultado es escrito a la salida estándar. Los caracteres que tengan un significado especial para el shell tienen que estar precedidos por el caracter de escape (\). Solo se pueden manejar números enteros o cadenas. Internamente, todos los números son manejados con enteros de 64 bits.

La lista de operadores válidos son los siguientes, en orden de precedencia. Los operadores que tienen la misma precedencia se listan entre llaves {}.

expr1 | *expr2*	Si *expr1* no vale 0 ni cadena nula, regresa dicha expresión. En caso contrario, regresa *expr2*. Ambas expresiones pueden estar formadas por cualquier lista válida de operadores y expresiones.
expr1 \& *expr2*	Regresa *expr1* si *expr2* no es cero ni nulo, de lo contrario regresa 0.
expr1 {=,\>,\>=,\<,\<=,!=} *expr2*	Regresa 1 si la comparación indicada es verdadera, y 0 en caso de que no lo sea.
expr1 {+,-} *expr2*	Regresa el resultado de la suma o resta indicada.
expr1 {*,/,%} *expr2*	Regresa el resultado de la multiplicación, división o residual indicada.
expr1:*expr2*	*expr2* tiene que ser una expresión regular, y este operador regresa el número de caracteres de *expr1* que coinciden con dicho patrón. Las expresiones regulares siguen las reglas que ya hemos visto, a excepción de que siempre se asume que tienen el ^ al principio, de manera que la comparación se hace desde el principio de la cadena. Por lo tanto, ^ no es un caracter especial. La única sintáxis especial en la expresión de búsqueda puede ser de la forma \(...\), lo cual regresaría los 3 primeros caracteres de *expr1*.

4.1.1. Ejemplos

El shell de Unix no tiene una manera directa de hacer operaciones matemáticas. Por ejemplo, si en un guión ejecutamos

```
a=3+4
```

y después hacemos un `echo $a`, lo que obtenemos es

```
3+4
```

Si lo que deseamos es sumar 3 más 4, tenemos que ejecutar

```
a=`expr 3 + 4`
```

Nótese que estamos utilizando los apóstrofes invertidos, que para el shell significan "lo que produzca como salida el comando que está encerrado entre ellas".

Así, si en un guión de shell estamos llevando alguna especie de contador en la variable `c`, para incrementarla tenemos que hacer:

```
c=`expr $c + 1`
```

Para encontrar el número de caracteres en una cadena contenida en la variable `cad`, podemos ejecutar:

```
expr $cad : '.*'
```

4.2. wc

A pesar de que tiene un parecido con cierto acrónimo de significado universal, el nombre de este comando significa "Word Count", y sirve para contar los caracteres, palabras y líneas de un archivo, dejando el resultado en la salida estándar. Su sintáxis es la siguiente:

```
wc [opciones] [archivo ...]
```

Si no se especifican opciones, se proporciona la cuenta tanto de líneas, palabras y caracteres del archivo especificado. Si no se especifican archivos, se toma la entrada estándar, y si se proporcionan varios archivos, se realiza la cuenta de cada uno de ellos por separado.

Las opciones válidas son las siguientes:

-c Cuenta el número de caracteres del archivo.

-l Cuenta las líneas.

-w Cuenta las palabras. Se considera como palabra una cadena de caracteres delimitada por espacios, tabuladores o cambios de línea.

4.2.1. Ejemplos

`who | wc -l`	Proporciona el número de usuarios que están trabajando en el sistema.

5. Comparación de archivos

Los comandos de esta sección nos sirven para realizar comparaciones entre archivos y/o directorios, en cuanto a su contenido.

5.1. diff

Este comando recibe como argumentos dos archivos, y nos dice qué cambios hay que realizar en el primero para convertirlo en el segundo. Su sintáxis general es:

```
diff [opciones] arch1 arch2
```

Obsérvese que los archivos no son opcionales, aunque en cualquiera de las dos posiciones se puede utilizar el guión (-) para indicar la entrada estándar. Los resultados siempre son depositados en la salida estándar.

La salida normal de este comando contiene líneas de las siguientes formas:

*n1***a***n3,n4* Significa que después de la línea *n1* del primer archivo se tienen que insertar las líneas de la *n3* a la *n4* del segundo archivo. En los siguientes renglones a esta instrucción vendrán las líneas de texto que se deben insertar, es decir, una copia de las líneas *n3* a *n4* del segundo archivo.

*n1,n2***d***n3* Significa que se tienen que borrar las líneas *n1* a *n2* del primer archivo. *n3* no se utiliza, y siempre es igual a *n1*-1. A continuación de esta instrucción viene una copia de los renglones a borrar.

*n1,n2***c***n3,n4* Significa que los renglones entre *n1* y *n2* del primer archivo se tienen que cambiar por los renglones entre *n3* y *n4* del segundo, viniendo a continuación los renglones originales, después una pequeña línea de guiones (----) y finalmente el texto que reemplazará al contenido original.

Las opciones para el `diff` son las siguientes:

-b Ocasiona que se ignore en la comparación el espacio en blanco (espacios y tabuladores) sobrantes al final de la línea, y que todas las cadenas de espacio en blanco se consideren iguales independientemente de su longitud. Utilizando esta opción, la línea "Hola todos" sería considerada igual a la línea "Hola todos ".

-e Produce como salida no las líneas descritas anteriormente, sino los comandos para el editor **ed** necesarios para convertir `arch1` en `arch2`.

-f Produce las instrucciones en formato **ed** de manera similar a la opción **-e**, pero en orden inverso, es decir, comenzando por la última y llegando a la primera.

-h Hace una comparación "rápida". Solo funciona cuando los cambios son pequeños y bien diferenciados, pero a cambio funciona con archivos de longitud ilimitada. No se pueden utilizar **-e** ni **-f** en conjunción con esta opción.

5.1.1. Ejemplos

Para convertir un archivo en otro automáticamente, se puede ejecutar la siguiente secuencia de comandos:

```
$(diff -e arch1 arch2; echo "w"; echo "q") | ed arch1
```

Con esto, se generan las instrucciones **ed** necesarias para la conversión, al final se añaden los comandos `w` y `q`, que significan Write (escribir el archivo al disco) y Quit (salir del programa) respectivamente. Todas estas instrucciones son pasadas a través de una tubería al **ed**, indicándole que opere sobre el archivo `arch1`. **ed** toma sus comandos de la entrada estándar, por lo que ejecutará lo que le pasamos a través de la tubería.

5.2. Variantes de diff: bdiff, sdiff, diff3

5.2.1. bdiff

Sirve para comparar archivos que son muy grandes para `diff`. Su sintáxis es:

```
bdiff arch1 arch2 [n] [-s]
```

La forma en la que opera es: ignora las líneas que sean comunes al principio de cada archivo, parte el resto de cada archivo en segmentos de *n* líneas, y ejecuta `diff` sobre los segmentos correspondientes. Las opciones son las siguientes:

n Tamaño de los segmentos en los que se van a partir los archivos. Por default es 3,500.

-s Especifica que no se imprimirá ningún mensaje por parte de `bdiff`, aunque no suprime posibles mensajes generados por `diff`.

A pesar de que los segmentos se procesan por separado, se ajustan los números de línea en la salida para que aparezca como que todo el archivo fue procesado junto. Debido a la segmentación, `bdiff` no siempre encuentra el mínimo conjunto de cambios posible.

5.2.2. sdiff

Este comando utiliza `diff`, pero interpreta los resultados para producir un listado de ambos archivos lado a lado, separados por un espacio. Dicho espacio está en blanco si las líneas son iguales en ambos archivos, contiene un < si la línea solo existe en el primer archivo, un > si la línea solo existe en el segundo, y un | si la línea existe en ambos pero son diferentes.

La sintáxis del `sdiff` es:

```
sdiff [opciones] arch1 arch2
```

Y las opciones válidas son:

-w *n* Especifica *n* como el ancho de las líneas a producir. Por default es 130 caracteres.

-l Solo imprime el lado izquierdo en las líneas que son idénticas.

-s No imprime las líneas que son idénticas.

-o *archivo* Especifica *archivo* como el nombre de un tercer archivo que será utilizado como el depositario de una mezcla controlada de `arch1` y `arch2`. Las líneas idénticas son copiadas a *archivo*. Los conjuntos de diferencias producidos por `diff` son impresas en la pantalla, donde un conjunto de diferencias se define como un grupo de diferencias contiguas que comparten el mismo separador de columnas. Después de imprimir cada conjunto de diferencias, aparece un prompt **%** y el programa espera por uno de los comandos siguientes, que deben ser tecleados por el usuario:

l -Añade la columna izquierda a *archivo*.

r -Añade la columna derecha a *archivo*.

s -Activa el modo silencioso: no imprime las líneas idénticas.

v -Desactiva el modo silencioso.

e l -Invoca al **ed** sobre la columna izquierda del conjunto de diferencias.

e r -Invoca al **ed** sobre la columna derecha del conjunto de diferencias.

e b -Invoca al **ed** sobre la concatenación de la columna izquierda con la derecha (las líneas de la izquierda, después las de la derecha).

e -Invoca al **ed** sobre un archivo vacío.

q -Termina el programa.

Cuando se teclea un comando que invoca al editor, saliendo del editor el archivo resultante es añadido a *archivo*.

5.2.3. diff3

Este comando permite hacer una comparación simultánea entre tres archivos distintos. Su sintáxis es:

`diff3 [-e|x|3]` *`arch1 arch2 arch3`*

Sin opciones, `diff3` imprime los rangos de los archivos que difieren, identificándolos con uno de los siguientes códigos:

= = = = Los tres archivos son diferentes.

= = = =1 *arch1* es diferente.

= = = =2 *arch2* es diferente.

= = = =3 *arch3* es diferente.

Los cambios que se deben realizar se indican de la siguiente manera:

f:*n1***a** El texto que sigue se debe insertar después de la línea *n1* del archivo *f*, donde *f*=1,2 o 3.

f:*n1*,*n2***c** El texto que sigue se debe poner en lugar de las líneas de la *n1* a la *n2* del archivo *f*.

Solo se puede tener una de las tres opciones a la vez, y sus significados son:

-e Escribe los comandos **ed** necesaiors para incorporar en *arch1* todos los cambios encontrados entre *arch2* y *arch3*, es decir, los cambios marcados con ==== y ====3.

-x Produce los comandos **ed** para incorporar en *arch1* los bloques diferentes en los tres archivos, es decir, los marcados con ====.

-3 Produce los comandos necesarios para incorporar en *arch1* los cambios marcados con ====3.

5.3. comm

Otra manera de visualizar diferencias y similitudes entre archivos es con el comando `comm`. Su sintáxis es:

```
comm [-123] arch1 arch2
```

Sin opciones, `comm` produce una salida en tres columnas: la primera contiene las líneas que solamente se encuentran en `arch1`, la segunda las que solo se encuentran en `arch2` y la tercera las líneas comunes a ambos archivos. Las opciones sirven para suprimir la primera, segunda o tercera columna, y se puede especificar más de un número a la vez. Por ejemplo, `comm -12` solamente imprime las líneas comunes a ambos archivos. `comm -123` no imprime nada.

5.4. cmp

Es el comando más sencillo y "tonto" de comparación de archivos, pues simplemente hace una comparación byte por byte de ambos archivos y reporta las diferencias. Su sintáxis es:

```
cmp [opciones] arch1 arch2
```

Sin especificar opciones, `cmp` hace la comparación byte por byte. Si los archivos son idénticos, no imprime nada. Si se encuentran diferencias, se imprime el número de línea y número de byte de la diferencia. Si el primer archivo es un subconjunto inicial del segundo (es decir, que todo el primer archivo está contenido al principio del segundo, pero este último es más largo), se reporta como tal. El código de salida es 0 cuando los archivos son idénticos, 1 cuando existieron diferencias y 2 cuando hubo algún problema, como un archivo inaccesible o inexistente.

Las opciones son:

-l Para cada diferencia, imprime el byte en que ocurrió (en decimal) y los bytes en cuestión (en octal).

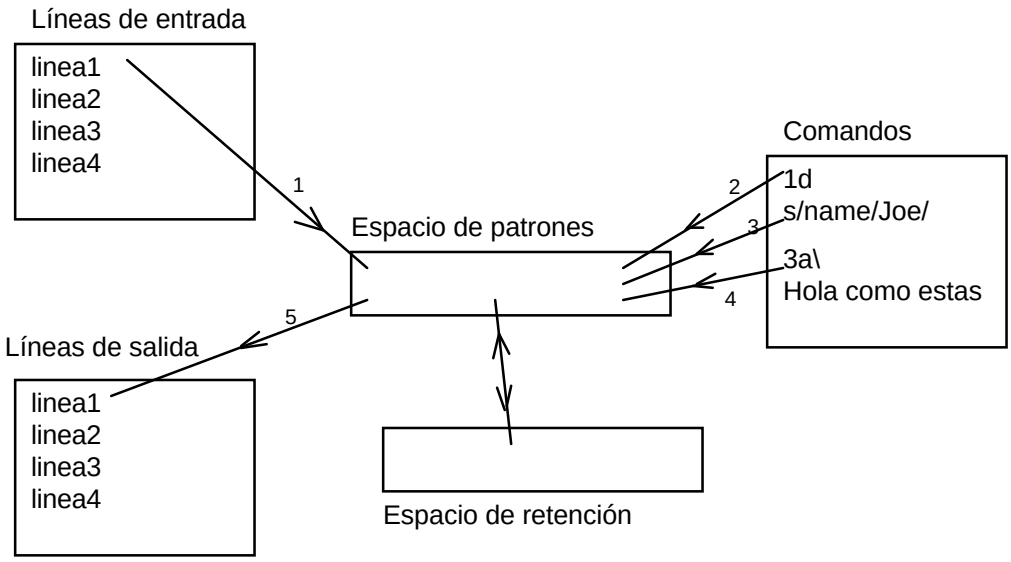
-s No imprime los mensajes correspondientes a las diferencias, solamente regresa los códigos apropiados.

-w Hace la comparación entre palabras de 64 bits en vez de entre bytes.

6. sed

`ed` es un editor muy poderoso (aunque un tanto hostil con el usuario), pero tiene un par de desventajas. La primera es que `ed` necesita cargar todo el archivo en la memoria simultáneamente, por lo que no sirve para archivos extremadamente largos. La segunda es que, aunque es posible, no es muy fácil especificarle a `ed` una secuencia de comandos que deben ejecutarse automáticamente. Para ello existe `sed`, que significa *Stream EDitor*, es decir, editor de flujo. Los comandos de `sed` son similares a los de `ed`, pero con una diferencia principal: `sed` toma su entrada de un archivo, lo edita, pero no modifica el contenido del archivo original, sino que deja el resultado en la salida estándar. `sed` también puede tomar los datos de entrada de la entrada estándar, por lo que cae dentro de nuestra clasificación de filtros.

Debido a que es un editor de flujo, `sed` no es un editor interactivo, sino que todos los comandos a ejecutar son proporcionados desde el principio. En la siguiente figura podemos apreciar cómo funciona `sed`.



Los números sobre las flechas indican el orden en que se van ejecutando, y la dirección de la flecha indica el sentido en el que fluye la información. La explicación es la siguiente:

Todos los comandos de `sed` son compilados antes de comenzar la ejecución. Una vez compilados, se extrae una línea de la entrada y se copia en un espacio reservado llamado *espacio de patrones*. Sobre esta línea se aplican todos los comandos que le correspondan. Finalmente, la línea, posiblemente modificada, es extraida del espacio de patrones y colocada en la salida estándar, al tiempo que se lee la siguiente línea de la entrada y se coloca en el espacio de patrones, para repetir el proceso hasta que las líneas de entrada se agoten. Algunos comandos pueden hacer uso del *espacio de retención* (hold space), que es una zona de almacenamiento temporal donde se pueden guardar algunas cosas para ser usadas posteriormente.

La sintáxis del `sed` es la siguiente:

```
sed [-n] [-e guion] [-f archivo] [archivo ...]
```

Las opciones son las siguientes:

-n Suprime la impresión de los patrones resultantes a la salida estándar, permitiendo de esta manera imprimir solamente lo que se desea, mediante los comandos de `sed` que tienen tal fin.

-e *guion* Especifica entre apóstrofes los comandos a ejecutar. Si solamente hay una secuencia de comandos y no se especifica la opción **-f**, la **-e** se puede omitir, y poner directamente la cadena de comandos.

-f *archivo* Lee de *archivo* los comandos a ejecutar.

Si no se especifican archivos de entrada, `sed` toma su entrada de la entrada estándar.

Las instrucciones de `sed`, ya sea en la línea de comandos o en un archivo, existen una por línea, y cada una tiene la siguiente estructura:

[*dirección1* [,*dirección2*]] *función* [*argumentos*]

Lo cual significa aplicar la función especificada sobre todas las líneas que caigan dentro de la dirección o direcciones indicadas.

Como ya se dijo, `sed` copia cada línea de la entrada al espacio de patrones, le aplica todas las funciones cuyas direcciones seleccionen dicha línea, y al final de los comandos copia lo que quede en el espacio de patrones a la salida estándar, a menos que se esté utilizando la opción **-n**.

Una dirección puede tomar una de tres formas:

1. Un número decimal que indica directamente el número de la línea a la que se hace referencia.
2. El signo $, que hace referencia a la última línea del archivo de entrada, sea cual sea su número.

3. Una *dirección de contexto*, es decir, una expresión regular de la forma */regexp/*que seleccionará aquellas líneas que contengan la expresión *regexp*. Las expresiones regulares siguen la misma estructura que ya ha sido vista.

Cabe hacer las siguiente observaciones sobre las direcciones:

- La cadena *\?regexp?*, donde *?* es cualquier caracter, es idéntica a */regexp/*. Tomando esto en cuenta, en **\xabc\xdefx** la segunda x lleva una \ antes, por lo que no se considera un caracter especial, de manera que la expresión regular indicada es **abcxdef**.
- La secuencia **\n** coincide con un cambio de línea, de manera que se pueden localizar cosas que estén sobre más de una línea del archivo de entrada. En este caso, todas las líneas involucradas serían leídas simultáneamente en el espacio de patrones, incluyendo los códigos de cambio de línea.
- El punto (.) coincide con cualquier caracter excepto con el cambio de línea que siempre se encuentra al final del espacio de patrones.
- Si no se especifican direcciónes, se aplica la función sobre todas las líneas de entrada.
- Si se especifica solo una dirección, se seleccionan todas las líneas que sean seleccionadas por dicha dirección.
- Si se especifican dos direcciones se selecciona el rango inclusivo desde la primera línea que coincide con la primera dirección hasta la primera línea después de esa que coincida con la segunda dirección. Después de encontrar la segunda dirección, se repite la búsqueda de la primera dirección, de manera que se puede actuar sobre varios bloques de texto que cumplan con ciertas condiciones en los extremos.
- Se puede especificar una función que actúe sobre todas las líneas que **no** sean seleccionadas por las direcciones especificadas anteponiendo un signo de admiración a la función deseada.

A continuación se detallan los principales comandos de `sed`. Entre paréntesis se indica el máximo número de direcciones que puede llevar cada función. En general, se puede utilizar la \ para continuar una función en la siguiente línea, así como para dividir algún argumento de *texto* en más de una línea.

(1)	**a** *texto*	Coloca *texto* en la salida estándar después de colocar en ella el contenido del espacio de patrones, pero antes de leer la siguiente línea de entrada.
(2)	**b** *etiqueta*	Brinca al comando **:** (ver más abajo) que tenga la etiqueta mencionada. Si no se especifica *etiqueta*, brinca al final del guión.
(2)	**c** *texto*	Borra el espacio de patrones actual (es decir, no pone nada en la salida estándar). Si tiene cero o una dirección o estamos al final de un rango de dos direcciones, se coloca *texto* en la salida estándar. Se inicia el siguiente ciclo, es decir, se pasa directamente a la lectura de la siguiente línea de entrada.
(2)	**d**	Borra el espacio de patrones (no pone nada en la salida estándar) y se inicia el siguiente ciclo.
(2)	**D**	Se borra el espacio de patrones hasta el primer cambio de línea y se inicia el siguiente ciclo.
(2)	**g**	Reemplaza el contenido del espacio de patrones con el contenido del espacio de retención.
(2)	**G**	Añade el contenido del espacio de retención al final del espacio de patrones.
(2)	**h**	Reemplaza el contenido del espacio de retención con el contenido del espacio de patrones.
(2)	**H**	Añade el contenido del espacio de patrones al final del espacio de retención.
(1)	**i** *texto*	Coloca *texto* en la salida estándar antes de colocar el contenido del espacio de patrones..

(2)	**l**	Imprime el contenido del espacio de patrones a la salida estándar, representando los caracteres no imprimibles con su código ASCII y segmentando las líneas demasiado largas.
(2)	**n**	Copia el espacio de patrones a la salida estándar y lee la siguiente línea de entrada.
(2)	**N**	Añade la siguiente línea de entrada al final del espacio de patrones, dejando un signo de cambio de línea entre ellas. Se incrementa el contador de línea actual.
(2)	**p**	Copia el espacio de patrones a la salida estándar sin hacer ninguna clase de interpretación como la que hace **l**.
(2)	**P**	Imprime en la salida estándar el contenido del espacio de patrones hasta el primer signo de cambio de línea.
(1)	**q**	Termina la ejecución de los comandos, sin iniciar un nuevo ciclo.
(2)	**r** *rfile*	Copia el contenido del archivo especificado a la salida estándar después de colocar la línea actual, pero antes de leer la siguiente. *rfile* tiene que estar separado de la **r** por exactamente un espacio.
(2)	**s**/*regexp*/ *reemplazo*/ *banderas*	Encuentra la primera subcadena del espacio de patrones que coincida con *regexp* y la sustituye por *reemplazo*. Se puede utilizar cualquier caracter en vez de /. Las banderas pueden ser: • **n** - Donde n=1 a 512, sustituye solo la n-ésima ocurrencia de *regexp* dentro de la línea. • **g** - Realizar un reemplazo global, es decir, sustituir **todas** las ocurrencias que no estén encimadas de *regexp* por *reemplazo*, en vez de hacerlo solamente para la primera ocurrencia. • **p** - Imprime a la salida estándar el espacio de patrones si es que se hizo alguna sustitución. • **w** *wfile* - Añade el espacio de patrones al archivo *wfile* si es que se hizo alguna sustitución.
(2)	**t** *etiqueta*	Brinca a la etiqueta especificada si se hizo alguna sustitución desde que se leyó la última línea de entrada o se ejecutó el último **t**. Si no se pone *etiqueta*, el salto es al final de la secuencia de comandos.
(2)	**w** *wfile*	Añade el espacio de patrones al final del archivo *wfile*. *wfile* tiene que estar separado de la **w** por exactamente un espacio, y no puede haber en toda la secuencia de comandos más de 10 *wfile*s distintos, incluyendo aquellos que aparezcan en los comandos **s**.
(2)	**x**	Intercambia el contenido del espacio de patrones y el espacio de retención.
(2)	**y**/*cad1*/*cad2*/	Transformación. Reemplaza todas las ocurrencias de los caracteres en *cad1* por los caracteres correspondientes de *cad2*, de manera similar al comando `tr` visto anteriormente. Las cadenas tienen que ser forzosamente de la misma longitud.
(2)	**!***función*	Se aplica la función o grupo de funciones especificado a las línea que **no** sean seleccionadas por la dirección o direcciones especificadas.
(0)	**:***etiqueta*	Sirve de destino para las funciones **t** y **b**.
(1)	**=**	Imprime en la salida estándar el número de línea actual.
(2)	**{**	Agrupa varias funciones (hasta el correspondiente }) para que sean ejecutadas bajo el control de una sola dirección o rango de direcciones.
(0)		Un comando vacío es ignorado (o sea, se pueden dejar líneas en blanco para hacer más legible un programa).

(0) # Si el # aparece como el primer caracter de la primera línea de un archivo de guión, toda la línea es considerada como un comentario. La única excepción es cuando el caracter inmediatamente posterior al # es una **n**. En ese caso, se activa la opción **-n** del `sed`, y se ignora el resto de la línea.

6.1. Ejemplos de sed

Supongamos que nuestro archivo de entrada (llamado `poema`) es:

```
Las rosas son rojas
Las violetas son azules
El azucar es dulce
Asi son todos.
```

A continuación se muestra el resultado de aplicar dos comandos `sed` distintos sobre este archivo:

```
$ sed 's/son/eran/' poema
Las rosas eran rojas
Las violetas eran azules
El azucar es dulce
Asi eran todos.
$ sed '/violetas/,4s/son/eran/' poema
Las rosas son rojas
Las violetas eran azules
El azucar es dulce
Asi eran todos.
```

En este último ejemplo, estamos combinando una dirección de contexto (/violetas/, que selecciona las líneas que contengan la cadena "violetas") con una dirección numérica (4), y sobre ese rango de líneas estamos aplicando la sustitución.

Otra aplicación muy común de `sed` es para borrar líneas en blanco de un archivo. Supongamos que tenemos el archivo `doble`:

```
Este archivo

esta

a doble espacio
```

Podemos eliminar los renglones extras con el siguiente comando:

```
$ sed '/^$/d' doble
```

En el cual estamos utilizando una dirección de contexto para seleccionar todas las líneas que no contengan nada entre el principio y el final de la línea. Ahora bien, si queremos eliminar no solamente las líneas que no contengan nada, sino también aquellas que solo contengan espacio en blanco (espacio y tabuladores) pues para efectos prácticos (o al menos visuales) esas líneas también están vacías, podemos utilizar el siguiente comando:

```
$ sed '/^[ \t]*$/d' doble
```

Con lo cual se seleccionan todas las líneas que contengan solamente espacios y/o tabuladores en cualquier número, incluyendo ninguno. Si en vez de borrar las líneas vacías queremos crear un nuevo archivo llamado `sencillo` que contenga solamente las líneas no vacías, podemos cambiar el comando a:

```
$ sed -n '/^[ \t]*$/!w sencillo' doble
```

Con lo cual estamos diciendo que se escriban al archivo `sencillo` todas las líneas que **no cumplan** (ojo con el signo de admiración) con la condición especificada, que sigue siendo la misma. Se

pone la opción **-n** para que no se produzca salida a la pantalla, sino que solamente se cree el archivo `sencillo` en el disco.

En todos los ejemplos que hemos visto, el resultado se produce en la pantalla. Si lo queremos guardar en un archivo, debemos redireccionar la salida del `sed` al archivo deseado, o incluir dentro el guión las instrucciones **w** necesarias para crear el archivo con la información requerida.

7. awk

`awk` es el comando más complejo de UNICOS (y de Unix en general), sobre todo porque no es un comando que realice una sola tarea, sino que es un lenguaje de programación hecho y derecho, pero con facilidades para descomposición de líneas y reconocimiento de patrones (expresiones regulares), proporcionando también capacidad de realizar cálculos aritméticos, y todo ello con una estructura sumamente similar al lenguaje C.

7.1. Historia

`awk` fue diseñado e implementado originalmente por Alfred Aho, Peter Weinberger y Brian Kernighan en 1977. El nombre del lenguaje, al contrario de los demás comandos de Unix, no viene de algún acrónimo de lo que hace el comando, sino que está formado por las iniciales de los tres apellidos de sus autores. Se realizó como parte de un experimento para ver cómo las herramientas `grep` y `sed` podrían ser generalizadas para tratar con números al igual que con texto. A pesar de que fue ideado para escribir programas muy cortos, sus facilidades y características pronto atrajeron a usuarios que comenzaron a escribir programas considerablemente más largos. Esto ocasionó la necesidad de nuevas cosas que no se encontraban en el lenguaje original, de manera que se publicó una nueva versión grandemente mejorada en 1985. La gúia definitiva a este lenguaje es el libro de Aho, Weinberger y Kernighan, *The Awk Programming Language*, publicado por Addison-Wesley. Este libro es la mejor referencia de Awk que se puede encontrar.

Una de las principales mejoras en esta nueva versión era la posibilidad para los usuarios de definir sus propias funciones, además de incluir muchas otras funciones intrínsecas y otras características adicionales que hicieron más fácil la escritura de programas complejos.

En muchos sistemas todavía se cuenta con la versión anterior del `awk`, mientras que la nueva versión se llama `nawk` para hacer la diferenciación. En otros sistemas el comando `awk` invoca directamente a la nueva versión del lenguaje. Desgraciadamente, el `awk` de UNICOS es un híbrido entre la nueva y la vieja versión, pues incluye algunas de las mejoras pero le faltan cosas sumamente importantes, como la posibilidad de funciones definidas por el usuario. En UNICOS no existe el `nawk`.

En esta sección se describirá la versión nueva de awk, haciendo notar las partes que no puedan funcionar en UNICOS debido a que se trata de cosas presentes solamente en la nueva versión de awk.

7.2. Tutorial

No hay mejor manera de aprender un lenguaje de programación que comenzar a programar en él lo más pronto posible. Esta sección, por lo tanto, es un tutorial en el cual comenzaremos a ver de inmediato programas en awk. El contenido de esta sección fue extraido casi íntegro del primer capítulo del libro de Aho, Weinberger y Kernighan.

7.2.1. El primer programa en awk

Muchos programas útiles en awk no son más de una o dos líneas. Supongamos que tenemos un archivo llamado `emp.data` que contiene el nombre, pago en nuevos pesos por hora, y el número de horas trabajadas por una serie de empleados, conteniendo un empleado por línea, de la siguiente manera:

```
Beth        4.00        0
Dan         3.75        0
Kathy       4.00        10
Mark        5.00        20
```

```
Mary        5.50        22
Susie       4.25        18
```

Ahora queremos imprimir el nombre y el pago total (pago/hora * horas trabajadas) para todos los que trabajaron. Esta clase de trabajo es justamente para lo que está diseñado awk, de manera que es sumamente fácil de lograr con el siguiente comando:

```
awk '$3 > 0 { print $1,$2*$3 }' emp.data
```

Con lo cual obtendríamos el siguiente resultado:

```
Kathy 40
Mark 100
Mary 121
Susie 76.5
```

Este comando le dice al sistema que ejecute el awk, utilizando el programa que se encuentra entre los apóstrofes, tomando los datos de entrada del archivo `emp.data`. Lo que está entre los apóstrofes es el programa completo de awk. Consiste de un solo *enunciado patrón-acción*. El patrón, `$3>0`, coincide con todas las líneas cuya tercera columna, o campo, es mayor que cero, y la acción

```
{ print $1,$2*$3 }
```

imprime el primer campo yel producto del segundo y tercer campos para cada línea que cumple con el patrón.

Si queremos imprimir los nombres de los empleados que no han trabajado, podemos utilizar el comando

```
awk '$3 == 0 { print $1 }' emp.data
```

Aquí el patrón, `$3==0`, selecciona todas las líneas cuyo tercer campo es igual a cero, y la acción

```
{ print $1 }
```

imprime el primer campo de dichas líneas. De manera que al ejecutar el comando anterior, el resultado obtenido es:

```
Beth
Dan
```

7.2.2. La estructura de un programa en awk

Detengámonos un momento a pensar qué es lo que está pasando. En los comandos anteriores, las partes entre apóstrofes son programas escritos en el lenguaje awk. Todos los programas awk son una secuencia de uno o más enunciados patrón-acción de la forma siguiente:

patrón { *acción* }

patrón { *acción* }

...

La operación básica de awk consiste en leer una secuencia de líneas de entrada una tras otra, buscando líneas que *coincidan* con alguno de los patrones del programa. El significado preciso de la palabra "coincidir" depende del patrón en cuestión. Por ejemplo, para patrones como `$3>0`, "coincidir" significa "la condición se cumple."

Cada línea de entrada es comparada contra el patrón en turno. Para cada patrón que coincide, la acción correspondiente, que puede incluir múltiples instrucciones, es ejecutada. Entonces la siguiente línea es leída y la comparación recomienza desde el principio. Esto continúa hasta que toda la entrada ha sido procesada.

Tanto el patrón como la acción (pero no ambos) puede ser omitido. Si un patrón no tiene acción, como el siguiente:

```
$3==0
```

entonces cada línea que coincida con el patrón (en este caso, cada línea para la cual la condición sea verdadera) será impresa en su totalidad. El programita anterior, aplicado sobre `emp.data`, imprimiría las dos líneas del archivo en las cuales el tercer campo es cero:

```
Beth        4.00        0
Dan         3.75        0
```

Si se tiene una acción sin patrón, como la siguiente:

```
{ print $1 }
```

entonces la acción, que en este caso es imprimir el primer campo de la línea, se ejecutará para todas las líneas de entrada.

Como tanto los patrones como las acciones son opcionales, las acciones van siempre encerradas entre llave para distinguirlas de los patrones.

7.2.3. Cómo se ejecuta un programa en awk

Hay muchas maneras de ejecutar un programa en awk. La primera es la que ya hemos visto, con un comando de la forma

```
awk 'programa' archivo ...
```

con lo cual se ejecuta el *programa* sobre cada uno de los archivos especificados, en secuencia.

También se pueden omitir los archivos de entrada y teclear solamente

```
awk 'programa'
```

con lo cual awk aplicará el *programa* a cualquier cosa que le llegue de la entrada estándar, ya sea del teclado o de algún redireccionamiento o tubería. Esto convierte a awk en un filtro, de acuerdo con lo que hemos visto anteriormente. También nos facilita experimentar con awk: podemos escribir nuestros programas, aplicarlos sobre la entrada estándar y darle datos desde el teclado, a ver cómo se comporta el programa.

Es importante notar que los programas están encerrados entre apóstrofes para proteger a los caracteres especiales como $ de ser interpretados por el shell. También permite que los programas sean de más de una línea (recordemos que, si el shell detecta que no se cerró una comilla o un apóstrofe, nos proporciona un prompt de continuación donde podemos seguir tecleando cosas que pertenezcan al mismo comando, y que serán interpretadas hasta que cerremos el caracter de apertura).

Esto que hemos visto es conveniente cuando el programa es corto. Sin embargo, a medida que el programa crece, es más conveniente ponerlo en un archivo separado, digamos `prog`, y utilizar el comando

```
awk -f prog [archivo ...]
```

La opción `-f` le indica al awk que lea el programa del archivo especificado.

7.2.4. Errores

Cuando cometamos algún error en un programa en awk, el awk nos dará un mensaje. Dependiendo del sistema en particular de que se trate, el mensaje será más o menos informativo. Si se trata de la versión vieja del awk (como en UNICOS) solamente nos dirá qué clase de error se obtuvo (normalmente serán errores de sintáxis) y a partir de qué línea se abortó el programa. Las nuevas versiones de awk (`nawk` y el `awk` de nocdos) imprimen también el contexto del error, es decir, algunos caracteres antes y después de donde fue detectado el error, para que nos sea más fácil encontrarlo y corregirlo.

7.2.5. Salida de datos sin formato

A partir de este momento, ya no mostraremos toda la línea de comandos completa que hay que ejecutar, sino solamente el programa del que se esté hablando. Los programas pueden ser ejecutados de cualquiera de las maneras que fueron mencionadas en la sección **Cómo se ejecuta un programa en awk**.

Existen dos tipos de datos en awk: números y cadenas de caracteres. El archivo `emp.data` es un típico ejemplo de esta clase de datos: una mezcla de palabras y números separados por espacios y/o tabuladores.

Awk lee su entrada una línea a la vez y parte cada línea en campos, donde, por default, un campo es una secuencia de caracteres que no contiene espacios ni tabuladores. El primer campo de la línea que acaba de ser leída es llamado `$1`, el segundo `$2`, etc. La línea entera es llamada `$0`. El número de campos puede varias de línea a línea.

IMPRESIÓN DE LA LÍNEA COMPLETA

Si una acción no tiene patrón, la acción es realizada para todas las líneas de entrada. El enunciado `print` por si solo imprime la línea de entrada actual, de manera que el programa

```
{ print }
```

imprime todas las líneas en la salida estándar. Como `$0` hace referencia a toda la línea,

```
{ print $0 }
```

hace exactamente lo mismo que el anterior.

IMPRESIÓN DE CIERTOS CAMPOS

Más de un elemento puede ser impreso en la misma línea de salida con un solo enunciado `print`. El siguiente programa imprime el primero y tercer campo de cada línea:

```
{ print $1, $3 }
```

Y con el archivo `emp.data` como entrada, la salida producida es:

```
Beth 0
Dan 0
Kathy 10
Mark 20
Mary 22
Susie 18
```

Las expresiones separadas por comas en el enunciado `print` son, por default, separadas por un solo espacio en blanco cuando son impresas. Cada línea producida por `print` termina siempre en un cambio de línea. Ambos separadores (de campos y de líneas), sin embargo, pueden ser cambiados, como veremos posteriormente.

LA VARIABLE NF

No solamente se pueden poner números fijos después del `$` al hacer referencia a una variable de campo, sino cualquier expresión; en este caso la expresión será evaluada y su valor numérico utilizado como número de campo. Awk cuenta el número de campos en la línea de entrada actual y almacena la cuenta en una variable automática llamada `NF`. Por lo tanto, el programa

```
{ print NF, $1, $NF }
```

imprime, para cada línea de entrada, el número de campos, el primer campo y el último.

REALIZACIÓN DE CÁLCULOS

Se pueden hacer cálculos matemáticos con los valores de los campos e incluir los resultados en lo que se imprime. El programa

```
{ print $1, $2*$3 }
```

que fue el primer programa visto, es un ejemplo claro de ello.

LA VARIABLE NR

Awk cuenta con otra variable automática, llamada `NR`, que siempre contiene el número de líneas leídas hasta el momento (incluyendo la actual). El siguiente programa utiliza `NR` y `$0` para numerar las líneas de un archivo:

```
{ print NR,$0 }
```

Aplicándolo sobre `emp.data`, la salida es:

```
1 Beth          4.00        0
2 Dan           3.75        0
3 Kathy         4.00        10
4 Mark          5.00        20
5 Mary          5.50        22
6 Susie         4.25        18
```

INCLUSIÓN DE TEXTO EN LAS IMPRESIONES

El comando `print` acepta texto entremezclado con los campos y valores. Por ejemplo, el programa

```
{ print "El pago total para",$1,"es",$2*$3 }
```

produce como resultado

```
El pago total para Beth es 0
El pago total para Dan es 0
El pago total para Kathy es 40
El pago total para Mark es 100
El pago total para Mary es 121
El pago total para Susie es 76.5
```

Obsérvese que no se pusieron espacios después del `para` ni antes o después del `es` en el programa, ya que el `print` automáticamente separa los elementos a imprimir con espacios.

7.2.6. Salida de datos con formato

El comando `print` sirve para imprimir cosas de forma rápida y sencilla. Para tener un control preciso sobre el formato de lo que imprimimos, podemos utilizar el comando `printf`.

EL COMANDO PRINTF

El comando `printf` tiene la siguiente forma:

```
printf(formato, valor1, valor2, ..., valorN)
```

y funciona exactamente igual que el comando printf del lenguaje C (los paréntesis rodeando los argumentos son opcionales). Es decir, *formato* es una cadena que contiene: 1) Texto literal a imprimir; 2) Especificadores de formato utilizando el caracter especial %. La primera especificación de formato encontrada indica cómo y dónde se imprimirá *valor1*, la segunda se refiere a *valor2*, etc. Por lo tanto, debe haber tantos especificadores de formato como valores a imprimir.

El siguiente programa hace lo mismo que el último de la sección anterior, pero utilizando `printf`:

```
{ printf("El pago total para %s es $%.2f\n",$1,$2*$3) }
```

La cadena de formato del `printf` contiene dos especificadores que comienzan con `%`. La primera, `%s`, indica la impresión del primer valor, `$1`, como una cadena de caracteres; la segunda, `%.2f`, indica la impresión del segundo valor, `$2*$3` como un número con dos dígitos después del punto decimal. Todo lo demás en la cadena de formato, incluyendo el signo $, es impreso tal cual; el `\n` al final de la cadena tiene el mismo significado que hemos visto siempre: cambio de línea.

Utilizando el programa anterior, la salida es:

```
El pago total para Beth es $0.00
El pago total para Dan es $0.00
El pago total para Kathy es $40.00
El pago total para Mark es $100.00
El pago total para Mary es $121.00
El pago total para Susie es $76.50
```

El `printf` no produce espacios ni cambios de línea automáticamente; nosotros debemos indicarlos explícitamente.

INTERACCIÓN CON OTROS COMANDOS DE UNIX

Supongamos que queremos imprimir toda la información para cada empleado, junto con su pago total, ordenado en orden creciente de pago. Lo más fácil de hacer es utilizar awk para añadir el pago total al principio de cada línea y pasar el resultado a través del comando `sort` para que haga el ordenamiento, es decir:

```
awk '{printf("%6.2f %s\n",$2*$3,$0)}' emp.data | sort
```

7.2.7. Selección de datos

Los patrones de awk son útiles para seleccionar de la entrada las líneas que nos interesan para procesarlas de alguna manera. Como un patrón sin una acción imprime todas las líneas que coincidan con el patrón, muchos programas de awk consisten solamente de un patrón con el fin de seleccionar ciertas líneas.

SELECCIÓN POR COMPARACIÓN

El patrón puede ser una condición de comparación, como en el siguiente programa, que imprime todos los empleados que ganan $5 o más por hora:

```
$2>=5
```

SELECCIÓN POR CÁLCULO

También se pueden incluir cálculos en los patrones y en las compraciones, como el siguiente:

```
$2*$3>=50 {printf("$%.2f para %s\n",$2*$3,$1)}
```

que imprime el pago para aquellos empleados cuyo pago total sea más de $50.

SELECCIÓN POR CONTEXTO

Aparte de comparaciones numéricas, los patrones también pueden contener comparaciones sobre palabras o frases o cualquier cadena de texto. El siguiente programa

```
$1=="Susie"
```

imprime todas las líneas cuyo primer campo es `Susie`. El operador `==` verifica la igualdad de dos elementos. También se puede buscar por líneas que contengan cualquier conjunto de letras, palabras o frases utilizando expresiones regulares encerradas entre diagonales (/). El siguiente programa imprime todas las líneas que contengan `Susie` en cualquier parte:

```
/Susie/
```

COMBINACIÓN DE PATRONES

Los patrones que hemos mencionado pueden ser combinados utilizando paréntesis y los operadores lógidos && (AND), `||` (OR) y `!` (NOT). El programa

```
$2>=4 || $3>=20
```

imprime todas las líneas cuyo segundo campo sea al menos 4 *o* el tercero sea al menos 20, lo cual produce:

```
Beth      4.00  0
Kathy     4.00  10
Mark      5.00  20
Mary      5.50  22
Susie     4.25  18
```

VALIDACIÓN DE DATOS

En los datos "del mundo real" siempre hay errores. Awk es una herramienta excelente para verificar que los datos tengan valores razonables y en el formato correcto, tarea a la cual se le llama comunmente *validación de datos.*

La validación de datos normalmente es negativa, es decir, en vez de imprimir las líneas con datos correctos, imprimimos las línea sospechosas. El siguiente programa utiliza patrones de comparación para aplicar 5 pruebas de factibilidad a cada línea de `emp.data`:

```
NF !=3      { print $0,"no tiene 3 campos"}
```

```
$2<3.35     { print $0,"el pago está por debajo del mínimo"}
$2>10       { print $0,"el pago excede N$10 por hora"}
$3<0        { print $0,"trabajo horas negativas"}
$3>60       { print $0,"trabajo demasiadas horas"}
```

Si no hay errores, no se produce ninguna salida.

BEGIN Y END

El patrón especial `BEGIN` coincide antes de que se lea la primera línea de entrada, y `END` coincide después de que la última línea ha sido procesada. Este programa utiliza `BEGIN` para imprimir un encabezado para las columnas del archivo:

```
BEGIN {print "Nombre  Pago/Hr   Horas";print ""}
{ print }
```

y produce la siguiente salida:

```
Nombre  Pago/Hr   Horas

Beth      4.00        0
Dan       3.75        0
Kathy     4.00        10
Mark      5.00        20
Mary      5.50        22
Susie     4.25        18
```

En este ejemplo observamos otra característica de awk, y es que podemos poner más de un enunciado en una sola línea si los separamos con punto y coma (;). Nótese que `print ""` imprime una línea en blanco, a diferencia de `print` solo, que imprime toda la línea.

7.2.8. Realización de operaciones

Ya hemos visto programas en los que se realizan operaciones sencillas y se utilizan las variables `NF` y `NR`. En esta sección veremos algunos otros ejemplos, utilizando también variables definidas por nosotros dentro del programa. En awk, las variables no necesitan ser declaradas.

CONTEOS

Este programa usa la variable `emp` para contar los empleados que han trabajado más de 15 horas:

```
$3>15 { emp=emp+1 }
END   { print emp,"empleados trabajaron mas de 15 horas" }
```

y, aplicándolo sobre `emp.data`, produce la siguiente salida:

```
3 empleados trabajaron mas de 15 horas
```

Para cada línea en que el tercer campo excede 15, el valor anterior de `emp` es incrementado en 1. Las variables en awk se inicializan automáticamente con cero, así que no necesitamos inicialización alguna para `emp`.

SUMAS Y PROMEDIOS

Para contar el número total de empleados podemos utilizar la variable automática `NR`. Su valor al final del archivo es el número total de líneas leídas:

```
END { print NR,"empleados" }
```

El siguiente programa utiliza `NR` para calcular el pago promedio:

```
      {pago=pago+$2*$3}
END   {print NR,"empleados"
       print "El pago total es",pago
       print "El pago promedio es",pago/NR
      }
```

La primera acción se aplica sobre todas las líneas del archivo, de manera que acumula el pago total para todos los empleados. La acción del `END` imprime:

```
6 empleados
El pago total es 337.5
El pago promedio es 56.25
```

MANEJO DE TEXTOS

Una de las ventajas de awk es que maneja textos tan facilmente como números. Las variables en awk no tienen un tipo definido, de manera que pueden contener números o cadenas de texto indistintamente, sin necesidad de declaraciones de ninguna clase. Por ejemplo, el programa

```
$2>maxpago  {maxpago=$2;maxemp=$1}
END   {print "Pago mas alto:",maxpago,"para",maxemp}
```

Y produce la salida

```
Pago mas alto: 5.50 para Mary
```

CONCATENACIÓN DE CADENAS

La concatenación es el proceso de unir dos o más cadenas para formar una sola, y en awk se indica simplemente poniendo una cadena a continuación de la otra. El programa

```
      {nombres=nombres $1 " "}
END   {print nombres}
```

Junta todos los nombres de los empleados en una sola cadena, añadiendo cada nombre y un espacio en blanco al valor anterior de la variable `nombres`. Al final, el valor de `nombres` es impreso por la acción `END`, produciendo

```
Beth Dan Kathy Mark Mary Susie
```

Así como los números son automáticamente inicializados en cero, las cadenas se inician como una cadena nula ("")

IMPRESIÓN DE LA ÚLTIMA LÍNEA

Es importante tener en cuenta que los valores `$0`, `$1`, etc. pierden su valor después de que se procesa cada línea, y por tanto no lo conservan al llegar a la acción `END`. Por lo tanto, para imprimir la última línea de un archivo, tenemos que hacer lo siguiente:

```
      { ultima=$0 }
END   { print ultima }
```

FUNCIONES INTERNAS

Ya hemos visto que awk mantiene ciertas variables automáticas que llevan valores importantes como el número de campos y el conteo de líneas. De la misma manera, existen funcionas ya integradas en awk que nos permiten obtener valores frecuentemente utilizados, tanto numéricos como de cadenas. Algunas de estas funciones sirven para obtener raíces cuadradas, logaritmos, números aleatorios y cosas así. Una de las funciones de texto existentes es `length`, que regresa el número de caracteres que tiene una cadena. Por ejemplo, el siguiente programa imprime la longitud de cada uno de los nombres:

```
{ print $1,length($1) }
```

CONTEO DE LINEAS, PALABRAS Y CARACTERES

El siguiente programa utiliza `length`, `NR` y `NF` para contar el número de líneas, palabras y caracteres en la entrada. Por conveniencia, se considera cada campo como una palabra.

```
      { nc=nc+length($0)+1
        nw=nw+NF
      }
END   { print NR,"lineas,",nw,"palabras,",nc,"caracteres" }
```

Nótese que se añade 1 a la cuenta de caracteres en cada línea para considerar el caracter de cambio de línea.

7.2.9. Control de flujo

Todo lenguaje de programación que se precie de serlo debe incluir maneras de controlar el flujo de ejecución de los programas, y awk no es la excepción. Los enunciados explicados a continuación solo pueden ser utilizados en acciones.

IF-ELSE

Es exactamente igual que el If-Else en lenguaje C. El siguiente ejemplo imprime el pago total y promedio de los empleados que reciben más de N$6/hora, utilizando `if` para evitar la división por cero.

```
$2>6  {n=n+1;pago=pago+$2*$3}
END   {if(n>0)
            print n,"empleados, pago total es",pago,
                  "pago promedio es",pago/n
       else
            print "Ningun empleado recibe mas de N$6/hr."
      }
```

Aquí podemos observar otra cosa: en awk, como en C, un enunciado puede comenzar en un renglón y terminar en otro si el último caracter de la línea es algún separador, como una coma (,).

WHILE

Este enunciado también funciona igual que en C, y sirve para hacer ciclos de terminación condicionada. El siguiente ejemplo muestra como el valor de una cantidad de dinero invertida a una cierta tasa de interés crece con el número de años, utilizando la fórmula $valor=cantidad(1+tasa)^{años}$.

```
# interes1.awk: Obtencion de interes compuesto
#  Entrada: cantidad tasa años
#  Salida:  Valor compuesto al final de cada año.
{ i=1
  while(i<=$3) {
      printf("\t%.2f\n",$1*(1+$2)^i)
      i=i+1
  }
}
```

Nota: El awk de UNICOS no soporta el operador ^ para la exponenciación, de manera que este programa tendría que ser reescrito utilizando algún otro algoritmo. Para probarlo, hacerlo en alguna estación de trabajo utilizando nawk o el equivalente.

Aquí observamos la forma de introducir comentarios en un programa en awk, utilizando el signo #. Todo lo que está de ahí hasta el final de la línea se considera comentario.

Utilizando este programa, podemos encontrar como $1000 crecen con 6% de interés compuesto anual a lo largo de 5 años:

```
$ awk -f interes1.awk
1000 .06 5
      1060.00
      1123.60
      1191.02
      1262.48
      1338.23
```

FOR

Este enunciado nos permite construir ciclos de la misma manera que funciona el for en lenguaje C. Así, el programa anterior podría se escrito como

```
# interes2.awk: Obtencion de interes compuesto
#  Entrada: cantidad tasa años
```

```
#  Salida:  Valor compuesto al final de cada año.
{ for(i=1;i<=$3;i=i+1)
      printf "\t%.2f\n",$1*(1+$2)^i
}
```

Al igual que en C, si el cuerpo del ciclo contiene una sola línea, no es necesario encerrarlo en llaves.

7.2.10. Arreglos

Awk nos permite utilizar arreglos para almacenar grupos de valores relacionados. Al igual que las variables, los arreglos no tienen que ser dimensionados ni declarados, sino que se van creando nuevos elementos a medida que son necesitados. El siguiente ejemplo imprime las líneas del archivo de entrada en orden inverso al que fueron leídas:

```
      { line[NR]=$0 }
END   { for(i=NR;i>0;i--) print line[i] }
```

Y podemos observar que, al igual que en C, contamos con los operadores `++` y `--` para incrementar o decrementar una variable.

Una característica muy peculiar y que puede ser muy útil de awk es que los índices de los arreglos no tienen que ser forzosamente números enteros, sino que también pueden ser cadenas. El siguiente programa imprime el pago por hora para cada uno de los empleados:

```
      { emp[NR]=$1;pago[$1]=$2 }
END   { for(i=1;i<NR;i++)
            print emp[i],pago[emp[i]]
      }
```

En la primera acción, se van creando dos arreglos: `emp` para contener los nombres de los empleados con un índice numérico, y `pago` para mantener el pago de cada empleado con un índice alfanumérico que es el nombre del empleado. En la acción final, accedemos a ambos arreglos, haciendo referencia a `pago` a través de su elemento correspondiente de `emp`.

7.2.11. Programas sencillos pero útiles

A continuación mostramos algunos programas que son cuando mucho de una o dos líneas, pero que muestran la gran cantidad de cosas que se pueden hacer en awk con muy poco código.

1. Imprimir el total de líneas de entrada:
   ```
   END { print NR }
   ```
2. Imprimir la décima línea de la entrada:
   ```
   NR==10
   ```
3. Imprimir el último campo de cada línea:
   ```
   { print $NF }
   ```
4. Imprimir todas las líneas con más de 4 campos.
   ```
   NF>4
   ```
5. Imprimir el número de líneas que contienen `Beth`:
   ```
   /Beth/      { n++ }
   END         { print n }
   ```
6. Imprimir todas las líneas con más de 80 caracteres:
   ```
   length($0)>80
   ```
7. Imprimir los dos primeros campos de cada línea en orden inverso:
   ```
   { print $2, $1 }
   ```
8. Intercambiar los dos primeros campos de la línea e imprimirla:
   ```
   { temp=$1;$1=$2;$2=temp;print }
   ```
9. Imprimir cada línea con el primer campo reemplazado por el número de línea:
   ```
   { $1=NR; print }
   ```

10. Borrar el segundo campo de cada línea:

```
{ $2=""; print }
```

11. Imprimir la suma de todos los campos de cada línea:

```
{suma=0
 for(i=1;i<=NF;i++) suma=suma+$i
 print suma
}
```

7.3. Estructura del lenguaje

Hemos terminado el tutorial básico de awk. Lo que viene a continuación son únicamente tablas de referencia de patrones, acciones, operadores, expresiones regulares, etc. Estas tablas contienen todo lo que es válido para la nueva versión de awk. Lo que está marcado con un ⊗ es aquello que no está disponible en el awk de UNICOS en la Cray Y-MP4/432 que posee la UNAM.

7.3.1. Sumario de patrones

Sumario de patrones

1. `BEGIN { ` *enunciados* `}`
   Solo se ejecutan los *enunciados* antes de leer la primera línea de entrada.
2. `END { ` *enunciados* `}`
   Solo se ejecutan los *enunciados* después de leer y procesar la última línea de entrada.
3. *expresión* `{ ` *enunciados* `}`
   Los *enunciados* se ejecutan sobre cada línea donde la *expresión* sea verdadera, es decir, no-cero o no-nulo.
4. `/`*regexp*`/ { ` *enunciados* `}`
   Los *enunciados* se ejecutan en cada línea que contenga una cadena que coincida con la expresión regular *regexp*.
5. *patrón_compuesto* `{ ` *enunciados*`}`
   Los *enunciados* se ejecutan cuando el *patrón_compuesto* es verdadero. Este último está formado por patrones sencillos unidos por operadores `&&`, `||` y `!` (AND, OR y NOT).
6. *patrón1*`,`*patrón2* `{ ` *enunciados* `}`
   Un patrón de rango coincide con cada línea desde una que coincida con *patrón1* hasta la siguiente línea que coincida con *patrón2*, inclusive; los *enunciados* son ejecutados sobre cada línea que coincida.

Notas:

a. Los patrones `BEGIN` y `END` no pueden ser combinados con otros patrones.

b. Un patrón de rango no puede ser parte de otro patrón.

c. `BEGIN` y `END` son los únicos patrones que forzosamente requieren una acción.

7.3.2. Operadores de comparación

Operador	Significado
<	menor que
<=	menor o igual que
==	igual que
!=	diferente de
>=	mayor o igual que
>	mayor que
~	coincide con (ver tabla 7.3.3)
!~	no coincide con

7.3.3. Operadores de coincidencia de cadenas

Operadores de Coincidencia de Cadenas

1. */regexp/*
   Coincide cuando la línea actual contiene una subcadena que coincide con *regexp.*
2. *expresión* `~ /regexp/`
   Coincide cuando el valor de cadena de *expresión* contiene una subcadena que coincide con *regexp*.
3. *expresión* `!~ /regexp/`
   Coincide cuando el valor de cadena de *expresión* no contiene una subcadena que coincide con *regexp*.

7.3.4. Expresiones regulares

Expresiones Regulares

1. Los metacaracteres de expresiones regulares en awk son:
   `\ ^ $ . [ ] | ( ) * + ?`
2. Una expresión regular básica es algo de lo siguiente:
   un no-metacaracter, como `A`, que coincide con sí mismo.
   una secuencia de escape que coincide con algún símbolo especial. Ej.: `\t` coincide con un tabulador.
   un metacaracter "escapado", como `\*`, que coincide con el metacaracter literalmente (no lo interpreta).
   `^`, que coincide con el principio de una cadena.
   `$`, que coincide con el final de una cadena.
   `.`, que coincide con cualquier caracter.
   una clase de caracteres: `[ABC]` coincide con el caracter `A`, `B` o `C`.
   las clases de caracteres pueden incluir abreviaciones: `[A-Za-z]` coincide con cualquier letra.
   una clase de caracteres complementada: `[^0-9]` coincide con cualquier caracter que no sea un dígito.
3. Los siguientes operadores combinan expresiones regulares básicas en expresiones mayores:
   alternación: `A|B` coincide con `A` o con `B`.
   concatenación: `AB` coincide con `A` seguida de `B`.
   cerradura: `A*` coincide con cero o más `A`'s.
   cerradura positiva: A+ coincide con una o más `A`'s.
   cero o uno: `A?` coincide con cero o una `A`.
   paréntesis: `(`*r*`)` coincide con la misma cadena que *r*.

7.3.5. Secuencias de escape (caracteres especiales)

Secuencia	Significado
`\b`	Retroceso de caracter (backspace)
`\f`	Cambio de página (formfeed)
`\n`	Cambio de línea (linefeed)
`\r`	Retorno de carro (sin cambio de línea)
`\t`	Tabulador
`\`*ddd*	Código ASCII *ddd*, donde *ddd* es un número octal de 1 a 3 dígitos entre 0 y 7
`\`*c*	Cualquier otro caracter *c* literalmente (p. ej.: `\\` para diagonal invertida, `\"` para ", etc.)

7.3.6. Sumario de expresiones regulares

Expresión	Coincide con
c	El no-metacaracter *c*
\c	Una secuencia de escape o el caracter literal *c*.
^	Principio de la cadena
$	Final de la cadena
.	Cualquier caracter
[c_1c_2...]	Cualquier caracter de c_1c_2...
[^c_1c_2...]	Cualquier caracter que no esté en c_1c_2...
[c_1-c_2]	Cualquier caracter en el rango que comienza con c_1 y termina con c_2
[^c_1-c_2]	Cualquier caracter que no está en el rango dec_1 a c_2
r_1|r_2	Cualquier cadena que coincida con r_1 o con r_2, donde r_1 y r_2 pueden ser cualquier otra expresión regular básica o compuesta.
(r_1)(r_2)	Cualquier cadena *xy* donde *x* coincide con r_1 y *y* coincide con r_2
(*r*)*	Cero o más cadenas consecutivas que coincidan con *r*
(*r*)+	Una o más cadenas consecutivas que coincidan con *r*
(*r*)?	Cero o una cadena que coincida con *r*
(*r*)	Cualquier cadena que coincida con *r* (no se necesitan paréntesis alrededor de las expresiones regulares básicas)

7.3.7. Resúmen de acciones

Acciones

Los enunciados de las acciones pueden incluir:

expresiones, con constantes, variables, asignaciones, llamadas a funciones, etc.

`print` *lista de expresiones*

`printf` [(]*formato,lista de expresiones*[)]

`if (`*expresión*`)` *enunciado*

`if (`*expresión*`)` *enunciado* `else` *enunciado*

`while (`*expresión*`)` *enunciado*

`for (`*expresión inicial*`;`*expresión de terminación*`;`*expresión de cambio*`)` *enunciado*

`for (`*variable* `in` *arreglo*`)` *enunciado*

`do` *enunciado* `while (`*expresión*`)`

`break`

`continue`

`next`

`exit`

`exit` *expresión*

`{` *enunciados* `}`

7.3.8. Variables automáticas

Variable	Significado	Default
ARGC	número de argumento en la lína de comandos	-
ARGV	arreglo de argumentos de la línea de comandos (ARGV[0..(ARGC-1)])	-
FILENAME	nombre del archivo de entrada	-
FNR	número de registro en el archivo actual	-
FS	controla el separador de campos de entrada	" "
NF	número de campos en el registro actual	-
NR	número de registros leídos hasta ahora (en total)	-
OFMT	formato de salida para números	"%.6g"
OFS	separador de salida de campos	" "
ORS	separador de salida de registros	"\n"
RLENGTH	longitud de la cadena encontrada por la función match	-
RS	controla el separador de entrada de registros	"\n"
RSTART	inicio de la cadena encontrada por la función match	-
SUBSEP	separador de elementos de un arreglo	"\034"

7.3.9. Funciones aritméticas incluidas

Función	Valor regresado
⊗atan2(*y*,*x*)	arcotangente de *y*/*x* en el rango de -π a π
⊗cos(*x*)	coseno de *x*, *x* en radianes
exp(*x*)	e^x
int(*x*)	Parte entera de *x* (decimales truncados)
log(*x*)	logaritmo natural (base e) de *x*
⊗rand()	númeo aleatorio *r* tal que $0 \leq r < 1$
⊗sin(*x*)	seno de *x*, *x* en radianes
sqrt(*x*)	raíz cuadrada de *x*
⊗srand(*x*)	*x* es la nueva semilla para rand()

Algunas implementaciones particulares de awk y nawk añaden funciones adicionales; verificar la página del manual correspondiente.

7.3.10. Funciones de cadena incluidas

Función	
⊗`gsub(`*r*`,`*s*`)`	sustituye *r* con *s* globalmente en `$0`. Regresa el número de sustituciones realizadas.
⊗`gsub(`*r*`,`*s*`,`*t*`)`	sustituye *r* con *s* gloablmente en *t*. Regresa el número de sustituciones realizadas.
⊗`index(`*s*`,`*t*`)`	regresa la primera posición de la cadena *t* en *s*, o 0 si *t* no se encuentra.
`length(`*s*`)`	regresa la longitud de *t*.
⊗`match(`*s*`,`*r*`)`	verifica si *s* contiene alguna subcadena que coincida con *r*; regresa la posición o 0 si no fue encontrada; cambia el valor de `RSTART` y `RLENGTH.`
`split(`*s*`,`*a*`)`	parte *s* en campos y los almacena en el arreglo *a*. La partición se hace de acuerdo al separador `FS.`
`split(`*s*`,`*a*`,`*fs*`)`	parte *s* en campos y los almacena en el arreglo *a*. La partición se hace de acuerdo al separador *fs*`.`
`sprintf(`*formato*`,` *lista*`)`	Regresa la cadena que hubiera sido impresa si los mismo parámetros se le hubieran pasado a `printf`.
⊗`sub(`*r*`,`*s*`)`	pone *s* en lugar de la primera subcadena más larga de `$0` que coincida con *r*; regresa el número de sustituciones hechas.
⊗`sub(`*r*`,`*s*`,`*t*`)`	pone *s* en lugar de la primera subcadena más larga de *t* que coincida con *r*; regresa el número de sustituciones hechas.
`substr(`*s*`,`*p*`)`	regresa los caracteres de *s* de la posición *p* en adelante.
`substr(`*s*`,`*p*`,`*n*`)`	regresa una subcadena de *s* de *n* caracteres comenzando en la posición *p*.

7.3.11. Operadores en orden de precedencia creciente

Operación	Operadores	Ejemplo	Significado del ejemplo
asignación	`= += -= *= /= %= ⊗^=`	`x*=2`	`x=x*2`
condicional	`?:`	`x?y:z`	si `x`, regresa `y`, si no, `z`
OR lógico	`||`	`x||y`	regresa 1 si `x` o `y` es verdadero, 0 si no.
AND lógico	`&&`	`x&&y`	regresa 1 si `x` y `y` son verdaderos, 0 si no.
membresía de arreglo	`in`	`i in a`	1 si `a[i]` existe, 0 si no.
coincidencia	`~ !~`	`$1 ~ /x/`	1 si el primer campo contiene una `x`, 0 si no.
relacionales	`< <= == != >= >`	`x==y`	1 si `x` y `y` son iguales, 0 si no.
concatenación		`"a" "bc"`	`"abc"`; no hay operador de concatenación explícito.
suma, resta	`+ -`	`x+y`	suma de `x` más `y`.
multiplicación, división, módulo	`* / %`	`x%y`	residuo de `x/y`.
más y menos unarios	`+ -`	`-x`	el valor negado de `x`.
NO lógico	`!`	`!$1`	1 si `$1` es cero o nulo, 0 si no.
⊗exponenciación	`^`	`x^y`	x^y
incremento, decremento	`++ --`	`++x, x++`	Sumar 1 a `x`, antes y después de utilizar el valor (en caso de que esté dentro de una expresión).
campo	`$`	`$i+1`	el valor del i-ésimo campo más 1.
agrupamiento	`( )`	`($i)++`	sumar 1 al i-ésimo campo. Si fuera `$i++`, se sumaría 1 al (i+1)-ésimo campo.

7.3.12. Enunciados de control de flujo

Enunciados de control de flujo

`{` *enunciados* `}`

agrupamiento de enunciados

`if (`*expresión*`)` *enunciado*

si *expresión* es verdadero, ejecuta *enunciado*

`if (`*expresión*`)` *enunciado1* `else` *enunciado2*

si *expresión* es verdadero, ejecuta *enunciado1*, si no, ejecuta *enunciado2*

`while (`*expresión*`)` *enunciado*

ejecuta *enunciado* mientras *expresión* sea verdadero

`for (`*expresión inicial*`;`*expresión de terminación*`;`*expresión de cambio*`)` *enunciado*

equivalente a *expresión_inicial*`;while(`*expresión_de_terminación*`)` `{`*enunciado*`;`*expresión_de_cambio*`}`

`for (`*variable* `in` *arreglo*`)` *enunciado*

ejecuta *enunciado* con *variable* tomando el valor de cada uno de los índices de *arreglo*

`do` *enunciado* `while (`*expresión*`)`

ejecuta *enunciado*; si *expresión* es verdadero, repite

`break`

salir inmediatamente del ciclo `while`, `for` o `do` más interno en el que se esté

`continue`

comenzar la siguiente iteración del ciclo `while`, `for` o `do` más interno en el que se esté

`next`

comienza la siguiente iteración del ciclo principal (leer la siguiente línea de entrada)

`exit`

`exit` *expresión*

ir inmediatamente a la acción `END`. Si se está en la acción `END`, terminar el programa inmediatamente. Regresa *expresión* como el código de salida del programa

7.3.13. Definición de funciones del usuario

En las versiones nuevas de awk, el usuario tiene la posibilidad de definir sus propias funciones. Una función se define con un enunciado de la forma

```
function nombre(lista-parámetros) {
        enunciados
}
```

Estas definiciones tienen que estar fuera de cualquier acción, y pueden estar en cualquier lugar del programa.

En la definición de una función, los ca mbios de línea son opcionales después de la primera llave y antes de la última. La lista de parámetros es una secuencia de nombres separados por comas; dentro del cuerpo de la función estas variables hacen referencia a los argumentos con los que fue llamada la función.

El cuerpo de la función puede contener un enunciado `return` que regresa el control y opcionalmente un valor al lugar de donde fue llamada la función. La forma es:

`return` [*expresión*]

Ejemplo:

```
function max(m,n) {
      return m>n?m:n
}
```

Dependiendo de la versión de awk de que se trate, puede o no ser posible la definición de funciones recursivas.

Todos los parámetros son pasados por valor, a excepción de los arreglos, que son pasados por referencia.

Todas las variables declaradas en la lista de parámetros son locales a la función, pero todas las demás variables son globales.

7.3.14. Enunciados de salida

Enunciados de salida de datos

`print`

Imprime `$0` en la salida estándar

`print` *expresión*`,` *expresión*`,...`

Imprime las expresiones separadas por `OFS`, terminadas por `ORS`

`print` *expresión, expresión ,...* `>` *archivo*

Imprime al archivo *archivo* en vez de a la salida estándar

`print` *expresión, expresión ,...* `>>` *archivo*

Añade al archivo *archivo* en vez de sobreescribir lo que ya estaba

`print` *expresión, expresión ,...* `|` *comando*

Imprime a la entrada estándar de *comando*.

`printf(`*formato*`,` *expresión*`,` *expresión*`,...)`

`printf(`*formato*`,` *expresión*`,` *expresión*`,...)` `>` *archivo*

`printf(`*formato*`,` *expresión*`,` *expresión*`,...)` `>>` *archivo*

`printf(`*formato*`,` *expresión*`,` *expresión*`,...)` `|` *comando*

Es similar al `print` pero imprime los datos con formato

`close(`*archivo*`), close(`*comando*`)`

cierra las comunicaciones entre `print` y el *archivo* o *comando* especificado

`system(`*comando*`)`

ejecuta *comando*; regresa el valor del código de salida del *comando*.

Los argumentos del **`printf`** no necesitan estar entre paréntesis. Pero si una expresión en la lista de argumentos contiene algún operador relacional, ya sea dicha expresión o la lista de argumentos tiene que estar entre paréntesis para evitar la interpretación del operador como un redireccionamiento o tubería.

7.3.15. Caracteres de formato de `printf`

Caracter	Imprime la expresión como
`c`	Caracter ASCII
`d`	Entero decimal
`e`	Número de punto flotante en notación científica
`f`	Número de punto flotante
`g`	`e` o `f`, lo que sea más corto.
`o`	Número octal sin signo
`s`	Cadena de caracteres
`x`	Número hexadecimal sin signo
`%`	Imprime un %, no se consume ningún argumento.

7.3.16. Ejemplos de especificaciones de `printf`

formato	`$1`	`printf(`*formato*`,$1)`
%c	97	a
%d	97.5	97
%5d	97.5	97
%e	97.5	9.750000e+01
%f	97.5	97.500000
%7.2f	97.5	97.50
%g	97.5	97.5
%.6g	97.5	97.5
%o	97	141
%06o	97	000141
%x	97	61
|%s|	January	|January|
|%10s|	January	| January|
|%-10s|	January	|January |
|%.3s|	January	|Jan|
|%10.3s|	January	| Jan|
|%-10.3s|	January	|Jan |
%%	January	%

7.3.17. Funciones de entrada de datos

Expresión	Cambia el valor de
`getline`	`$0, NF, NR, FNR`
`getline` *var*	*var*, `NR, FNR`
`getline <` *archivo*	`$0, NF`
`getline` *var* `<` *archivo*	*var*
comando `| getline`	`$0, NF`
comando `| getline` *var*	*var*

7.3.18. Interpretación de argumentos de la línea de comandos

Los argumentos de la línea de comandos están disponibles para el programa en awk por medio de el arreglo automático `ARGV`. El valor de la variable automática `ARGC` es uno más que el número de argumentos. Por ejemplo, con la línea de comandos

```
awk -f prog a v=1 b
```

`ARGC` tiene el valor 4, `ARGV[0]` contiene `awk`, `ARGV[1]` contiene `a`, `ARGV[2]` contiene `v=1` y `ARGV[3]` contiene `b`. `ARGC` es uno más que el número de argumentos porque `awk`, el nombre del comando, es contado como el argumento cero, al igual que en C. Ninguna de las opciones del comando awk (-F, -f) se incluye en los argumentos.

Nótese que incluso el nombre del archivo de entrada (no el del programa) es considerado parte de los argumentos, de manera que podemos alterarlo en la sección de `BEGIN` para que el programa actúe sobre algún archivo diferente.