Chapter 2

Aplicaciones de un sólo uso

Supongan que como ingeniero a cargo del sistema han decidido instalar un servidor de ftp anónimo, donde han colocado información pública sobre la empresa en la que trabajan para que sea consultable desde el mundo entero vía Internet.

Su jefe descubre que usted ha perdido una semana de su tiempo en eso y no está contento. Usted le explica que esto es una buena idea puesto que desde todo el mundo ahora obtienen información sobre su empresa. Su jefe lo mira incrédulo y le dice "¿y cuanta gente ha accesado el sistema?"

Usted recuerda que el servidor de ftp tiene una bitácora de acceso, un archivo donde escribe una línea por cada usuario que se conecta con los siguientes datos:

```
fecha, hora, maquina, archivo, bytes, tiempo
```

El nombre de la máquina que se conectó es `maquina`, `archivo` es el nombre del archivo que copió, `bytes` es el número de caracteres que se transmitieron y `tiempo` es la duración de la transferencia en segundos. Todos van separados por comas.

Entonces, usted recuerda sus clases de Pascal y decide que es trivial hacer un programa que cuente las líneas del archivo:

```
program cuenta(input, output)

var lines : integer;
var line  : string;
```

```
lines := 0;
while not eof do
  begin
    readln(line);
    lines := lines + 1;
  end;

writeln(lines);
end.
```

El resultado es que 200 máquinas han accesado el servidor. El jefe comienza a entusiasmarse, y pregunta "bien, y ¿cuales son esas máquinas?".

Usted decide que modificar el programa anterior no es tan difícil y hay que imprimir todas las máquinas del archivo:

```
program cuenta(input, output)

var lines : integer;
var line  : string;
var maquina : string;

lines := 0;
while not eof do
  begin
    readln(line);
    {  buscar en la linea la coma numero 2, tomar el nombre de la
       maquina y copiarlo al string llamado maquina.
       este codigo no es trivial }
    writeln(maquina);
    lines := lines + 1;
  end;

writeln(lines);
end.
```

Con nuestro programa generamos un listado después de algunos esfuerzos, con 200 nombres impresos para mostrarle al jefe. Al mirarlo, se da cuenta que hay máquinas del extranjero, y muchas aparecen varias veces, entonces exclama: "está muy bien, entrégame ahora un listado ordenado por frecuencia de acceso, en que cada máquina aparezca una sola vez con el número total de accesos que hizo. Entrégame otro en que aparezcan los archivos que se llevaron, ordenados por frecuencia también."

A esa altura usted mira el programa en Pascal y no parece tan fácil de modificar, hacer una tabla de frecuencias, ordenar, parece demasiado trabajo a pesar que todo es conocido.

En realidad este tipo de requerimientos es usual y cotidiano, y no tiene sentido estar haciendo programas que implementan las mismas cosas conocidas como métodos de ordenamiento, separación de palabras en la entrada, etc. Tal como el ingeniero que diseña una central eléctrica, no se requiere que él adem'as construya los generadores. Lo que se requiere es un conjunto de herramientas que sean fácilmente conectables las unas con las otras para crear soluciones globales. Una herramienta sería un ordenamiento (*sort*), otra un separador de palabras, etc.

En ambientes de desarrollo es imperioso contar con un conjunto de herramientas, que permitan desarrollar soluciones instantáneas a preguntas como éstas. Las herramientas básicas conectables se llaman filtros, y básicamente procesan su entrada de datos y generan una salida aplicando alguna transformación básica (por ejemplo contar las líneas de un archivo se puede hacer con el comando `wc -l`).

En Unix existen varios ejemplos de este tipo que estudiaremos a continuación.

2.1 Filtros

Supongan que tienen un archivo de texto y quieren obtener las frecuencias de las palabras que aparecen en él (esto es una versión simplificada del problema anterior). Un programa que hace esto puede escribirse usando un separador de palabras, luego buscando la palabra en una tabla y llevando un contador asociado. Al final, ordenamos la tabla según frecuencia de aparición y la imprimimos. Usando filtros podemos hacerlo también.

Tenemos un filtro que separa las palabras de la entrada y escribe en su salida una palabra por línea. Luego tenemos un programa que ordena lexicográficamente y uno que elimina duplicados:

- `words`

  Separa palabras de la entrada. Este filtro como tal no existe en Unix, pero se puede usar otro filtro llamado `tr` para producir el mismo efecto con argumentos un poco más complejos (ver `man tr`)[1].

- `sort`

[1] Una primera aproximación de `words` es: `tr ' ,.;"`
`t' '`
`n'`

Ordena de menor a mayor lexicográficamente la entrada. Si se le da como argumento la opción `-n` hacer un sort numérico. La opción `-r` hace un ordenamiento de mayor a menor.

- `uniq`

  Elimina las líneas consecutivas iguales, dejando una sola de cada una. Con la opción `-c` acompaña cada línea con un contador que dice cuantas líneas iguales habían en la entrada.

  Ejemplo: Si el archivo `in` es:

  ```
  hola
  hola
  chao
  y nada
  mas
  mas
  ```

  el comando:

  ```
  % uniq -c <in
  ```

  genera:

  ```
  2 hola
  1 chao
  1 y nada
  2 mas
  ```

Nuestro problema entonces podemos resolverlo usando redirección de entrada y salida en Unix, y archivos auxiliares. Los datos vienen en el archivo `in` y los resultados estarán en `out`.

```
% words <in >aux         # en aux queda una palabra por linea
% sort <aux >aux2        # ordenamos todas las palabras
% uniq -c <aux2 >aux3    # contamos frecuencia de aparicion
% sort -rn <aux3 >out    # ordenamos por frecuencia
```

Esta solución está correcta, pero es incómodo estar usando archivos auxiliares cuando esos datos intermedios no son útiles después.

En Unix, la salida de un programa puede conectarse con la entrada de otro programa, usando *pipes*. Un *pipe* es una tubería de conexión donde todos los datos de salida de un filtro son entregados como entrada a otro. Ver Figura 2.1. Ejemplo:

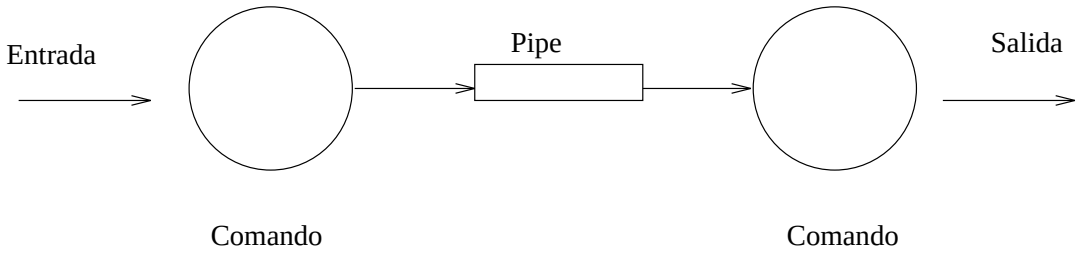

Figure 2.1: Comandos usando un *pipe*

```
% words <in | sort
```

Esto obtiene las palabras del archivo `in` y las ordena.

Como en un Lego, los filtros pueden seguir conectándose unos con otros, y el programa completo anterior puede escribirse en una sóla línea:

```
% words < in | sort | uniq -c | sort -rn > out
```

Nuevos requerimientos de filtraje son fácilmente agregados. Por ejemplo, si queremos sólo las diez palabras más frecuentes, usamos el filtro `head` que tiene una opción con el número de líneas que deseamos dejar:

```
% words < in | sort | uniq -c | sort -rn | head -10 >out
```

Si queremos enviarle un mail a jorge con este resumen:

```
% words < in | sort | uniq -c | sort -rn | head -10 | mail -s "Top 10" jorge
```

Cuando alguno de estos filtros no existe, es un trabajo menor hacer un programa que cumpla esa función y agregarlo a la serie.

Cuando la serie de filtros es un comando útil, se puede escribir en un archivo y luego ejecutarlo. Un archivo de comandos se conoce también como un *shell script*. Dentro de estos archivos, podemos aplicar algo de control de flujo con primitivas como `while` o `if`. En el ejemplo, podemos obtener las palabras más frecuentes de todos los archivos de un directorio:

```
#!/bin/csh

foreach i (*)
    words <$i | sort | uniq -c | sort -rn | head -10 | mail -s "Top $i" jorge
end
```

En el ejemplo, introducimos una variable `i` que en cada iteración contiene el nombre de un archivo (los nombres se obtienen con el `*`). Para obtener el contenido de la variable, se antepone el símbolo $ (peso).

Si estas líneas las guardamos en un archivo llamado `top`, y luego ejecutamos el comando:

```
% chmod +x top
```

Entonces podemos ejecutar directamente el archivo, dando el comando `top`. Los archivos de comandos también pueden recibir argumentos, que vienen en las variables predefinidas `$1, $2, ....` Por ejemplo podemos usar un argumento que diga en qué directorio hay que ejecutar el comando `top`:

```
#!/bin/csh

cd $1
foreach i (*)
    words <$i | sort | uniq -c | sort -rn | head -10 | mail -s "Top $i" jorge
end
```

Esto se puede ejecutar ahora como:

```
% top archivos
```

y entregará las frecuencias de las palabras en el directorio `archivos`.

2.2 Expresiones Regulares

Usando shell scripts y filtros obtenemos dos grandes fuerzas: control de flujo y herramientas de software fácilmente conectables. Sin embargo hay cosas que todavía no podemos resolver. Por ejemplo, el problema original de la lista de máquinas del archivo de salida requiere obtener ciertas partes de la línea, ignorando otras. En algunos casos, la entrada es muy estructurada, pero en otros no es así, y requerimos de sistemas de búsqueda de secuencias más elaborados.

Para eso, es muy complejo escribir un programa que analice la entrada buscando secuencias, y es útil tener la habilidad de especificar lo que se busca sin programar. Para esto usaremos una nueva funcionalidad: Expresiones Regulares.

Una expresión regular se utiliza en un *patrón* de búsqueda. En el caso de los filtros, buscamos líneas que calcen con el patrón. Por ejemplo, queremos buscar las líneas que contienen la palabra `end`. El patrón de búsqueda es: `end`. Estos patrones los utilizan varios filtros y particularmente el editor de texto (`vi`, `ed`, etc). Un patrón que contiene una expresión regular permite buscar líneas que contengan secuencias más complejas, y mediante símbolos especiales provee varias funcionalidades:

- `[...]`

  Esto representa un patrón que calza con un carácter, que debe estar dentro del grupo encerrado entre `[` y `]`. Ejemplo: `[123456790]` calza con un dígito. Para hacer más corta la notación, también se puede especificar un rango: `[0-9]` es equivalente a lo anterior.

  Si el grupo de caracteres especificado comienza con `^` (not) entonces el patrón calza con todos salvo con los caracteres especificados. Ejemplo: `[^ ]` busca el primer carácter diferente de blanco.

- `.`

  El símbolo "punto" calza con cualquier carácter. Es el comodín.

- `*`

  Implica 0 o más ocurrencias del símbolo anterior. Por ejemplo `a*` calza con ninguna "a", una o varias "a". El patrón que calza con cualquier secuencia es: `.*`

- `+`

  Implica 1 o más ocurrencias del símbolo anterior. Es igual a `*` salvo que no acepta el caso de ninguna ocurrencia.

- `^`

  Si es el primer carácter del patrón, calza con el comienzo de la línea. Ejemplo: `^From`, calza con las líneas que comienzan por `From` (sin blancos y ningún carácter antes).

- `$`

  Si es el último carácter del patrón, calza con el fin de la línea. Ejemplo: `bytes$` calza con todas las líneas que terminan con la secuencia `bytes`.

- `\`

  Cuando queremos buscar textualmente un símbolo especial en la secuencia (un *, [, etc) lo precedemos por un *backslash* (\). Por ejemplo `[A-Z]\*` busca una letra mayúscula seguida de un asterisco textual.

2.2.1 `egrep`

El filtro más famoso que utiliza expresiones regulares es `egrep`. Aunque posee más capacidades que las descritas, podemos basarnos en ellas para construir filtros bastante poderosos. `egrep` recibe como argumento una expresión regular, y lee toda su entrada escribiendo en su salida sólo las líneas que calzan con la expresión.

Si queremos obtener todas las líneas de nuestra bitácora de ftp que corresponden al mes de agosto, y el formato de la fecha es:
Sun Aug 21 17:26:51 CST 1994

Podemos hacer `egrep '^... Aug'` (los apóstrofos (') son necesarios para evitar que el shell interprete los caracteres especiales de la expresión).

En un ejemplo anterior, vimos que podíamos obtener frecuencias de palabras en un texto con:

```
% words < in | sort | uniq -c | sort -rn > out
```

Ahora podemos decidir que sólo nos interesan las palabras con algún carácter alfabético (excluímos así los números y símbolos):

```
% words <in | egrep '[A-Za-z]' | sort | uniq -c | sort -rn > out
```

El comando `egrep` acepta una opción `-v` que invierte el patrón y sólo selecciona las líneas que no calzan con el patrón. Si ahora sólo nos interesan las palabras con *todos* sus caracteres alfabéticos:

```
% words <in | egrep -v '[^A-Za-z]' | sort | uniq -c | sort -rn > out
```

Puede verse que `egrep '[A-Za-z]'` no es equivalente a `egrep -v '[^A-Za-z]'` (a pesar de la doble negación, ¿porqué?). En el segundo caso, patrón calza con las palabras que no contienen letras, y con la opción `-v` pido que me muestre las demás, es decir las que contienen sólo letras (no hay que confundir el patrón `[^A-Za-z]` – que busca caracteres no alfabéticos – con el patrón `^[A-Za-z]` – que busca caracteres alfabéticos al comienzo de la línea).

Por ejemplo, si queremos eliminar algunas palabras del archivo original (en este caso no queremos contar la palabra '`de`'):

```
% words <in | egrep -v '^de$' | sort | uniq -c | sort -rn > out
```

Colocar `^` y `$` es necesario para evitar borrar las palabras que contienen /tt de como subsecuencia.

2.2.2 El editor

Dentro del editor `vi`, tenemos expresiones regulares para hacer varias cosas:

- Búsquedas

  Dentro del editor puedo buscar una expresión regular en el texto editado:

  ```
  /[A-Za-z]/
  ```

Posiciona el cursor sobre la primera letra alfabética del archivo.

- Sustituciones

  El comando sustituir acepta expresiones regulares que serán cambiadas por otra secuencia:

  ```
  s/[ ,.;]+/ /
  ```

  cambia una serie de separadores en un sólo blanco. Normalmente el reemplazo se aplica sobre la primera aparición del patrón en la línea. Si queremos que se aplique sobre todas las apariciones, se coloca una `g` (de global) al final:

  ```
  s/[ ,.;]+/ /g
  ```

  Cambia todas las secuencias de separadores por un blanco.

  La sustitución siempre se aplica sobre la secuencia más larga posible que calce con el patrón.

  En la expresión de reemplazo no va una expresión regular. Sin embargo, se permiten algunos caracteres especiales: `&` reemplaza la secuencia que calzó con el patrón.

  ```
  s/[0-9]+/Numero: &/
  ```

  Antepone a un número cualquiera la palabra `Numero:`.

  También se permite usar sub-expresiones dentro de la expresión regular. Éstas van encerradas entre `\(` y `\)`. En el lado de la expresión de reemplazo se referencia la sub-secuencia que calzó con ellas como `\1`, `\2`, etc. Si queremos invertir dos palabras separadas por un blanco:

  ```
  s/\(.*\) \(.*\)/\2 \1/
  ```

- Comandos Globales

  Otro uso importante en el editor es la posibilidad de aplicar un comando en todas las líneas que calzan con un patrón dado. Esto son comandos globales:

  ```
  g/^From/s/ +/ /g
  ```

Este comando reemplaza una secuencia de blancos por uno solo en todas las líneas que comienzan por `From`. También podemos aplicar cualquier comando como borrar:

```
g/^From/d
```

Borra todas las líneas que comienzan por `From`. Esto permite comandos increíblemente poderosos, al usar comandos como borrar y mover líneas. El orden de ejecución es importante para determinar el resultado: primero se marcan todas las líneas que calzan con la secuencia, luego se aplica el comando asociado una por una (de menor a mayor).

En el editor existe el comando mover, que recibe un rango de líneas y una línea de destino:

```
5,10m0
```

Esto mueve las líneas 5 a 10 al comienzo del archivo (línea 0). También se puede usar la línea . (punto) que es la línea actual (donde estamos en el editor). También se permite usar expresiones simples, como `.+1`, que es la línea siguiente a la actual. Cuando usamos comandos globales, la línea punto permite referenciar la línea donde estamos aplicando el comando. ¿Qué hacen los comandos siguientes?

```
g/^From$/.m0
g/.*/.m0
g/.*/.+1d
g/.*/.+2d
1,$m0
```

El último comando no hace nada parecido al segundo. ¿Porqué?

- El editor como filtro: sed

  El editor de texto puede verse como un filtro muy poderoso, que actúa sobre su entrada en base a un grupo de comandos, generando en su salida el archivo modificado. Existe una versión especial del editor para utilizarse de esta forma y es el comando `sed`.

  A diferencia del editor, lo normal es que los comandos del editor que recibe `sed` son ejecutados sobre *todas* las líneas de la entrada.

  `sed` recibe los comandos como argumentos o dentro de otro archivo. Cuando es un sólo comando es preferible ponerlo como argumento, pero cuando son muchos la notación es muy pesada y es mejor usar un archivo de comandos. Por ejemplo, si queremos eliminar todas las líneas que comienzan por `From`:

```
% sed '/^From/d' <in >out
```

Podemos aplicar comandos de sustitución:

```
% sed 's/[ ,.;:]+/ /g'
```

Obviamente podemos incorporarlo en una serie de pipes[2]:

```
% sed 's/,/\
/g' < in | wc -l
```

A esta altura somos capaces de hacer aplicaciones que procesan archivos de texto bastante poderosas, por ejemplo el listador de nuestra bitácora de accesos `ftp` anónimo requiere sólo quedarse con los nombres de máquinas y eliminar el resto de la línea. Si la bitácora está en el archivo `ftp.log`, con el formato habitual:

```
fecha, hora, maquina, archivo, bytes, tiempo
```

El comando es:

```
% sed 's/[^,]*,[^,]*,\([^,]*\),.*/\1/' <ftp.log
```

Propuesto: obtener las frecuencias de acceso de las máquinas ordenadas de más frecuente a menos.

2.3 Lenguajes de procesamiento de patrones

En base a archivos de comandos, filtros y expresiones regulares hemos construido una base bastante sólida para desarrollar aplicaciones de un sólo uso. Sin embargo, hay complicaciones e inconsistencias de un comando a otro que hacen difícil la tarea sin necesidad. Por ejemplo algunos comandos aceptan `+` en las expresiones regulares y otros sólo aceptan `*` (la funcionalidad no se pierde, sólo la notación es más breve).

Por ello, se han propuesto nuevos lenguajes de programación que se basan en integrar todas estas ideas en un ambiente de programación. Los ejemplos más conocidos son `awk` y últimamente `perl`.

[2] Para colocar un fin de línea en una secuencia de comandos al shell, debemos precederlo por backslash y luego dar return.

2.3.1 awk

Este lenguaje es bastante completo, pero sólo veremos sus funcionalidades básicas a modo de ejemplo.

La idea es tener un lenguaje de programación con procesamiento de patrones y texto integrado. `awk` está pensado para procesar una entrada que sigue cierto patrón, ojalá una serie de 'campos' separados por un separador conocido (como nuestra bitácora de ftp que está separada por comas). La entrada es un serie de líneas y el programa será aplicado a cada una de ellas.

El caso más simple es obtener los campos separados:

```
% awk 'BEGIN{FS=","}{print $3}' < ftp.log
```

Este ejemplo define el separador de campos como una coma, y luego imprime el tercer campo para cada línea (es equivalente al último ejemplo de `sed` visto). Un grupo de instrucciones precedido por `BEGIN` se ejecuta una sóla vez, el grupo principal se ejecuta una vez por línea y finalmente se puede agregar un grupo precedido por `END` que se ejecuta al finalizar.

Por ejemplo un programa que cuenta el total de bytes transferidos (campo número 5):

```
% awk 'BEGIN{FS=","}{suma = suma + $5}END{print suma}' < ftp.log
```

Las variables no se declaran y sus tipos se determinan según el contexto en que se usan y nacen automáticamente inicializadas. Por ejemplo `suma` es un entero que comienza valiendo cero.

La entrada se analiza para evitar errores innecesarios por ejemplo si el quinto campo no es un entero, o no hay cinco campos en algunas líneas `$5` toma el valor cero y no da un error.

Como es un lenguaje de programación, ahora podemos hacer aritmética, por ejemplo, además del total de de bytes, imprimir también el promedio por conexión:

```
% awk 'BEGIN{FS=","}{suma = suma + $5; n = n + 1} \
       END{print "Total es ", suma; print "Promedio es ", suma/n}'
```

Dentro del lenguaje se pueden invocar comandos como en un shell y redireccionar la salida y entrada hacia o desde expresiones:

```
% awk 'BEGIN{FS=","}{print $3 | "sort"}' < ftp.log
```

También existen arreglos en awk, los sub-índices son siempre strings, lo que permite tener arreglos "asociativos", puesto que uno asocia valores a strings cualesquiera. Para recorrer un arreglo se usa una construcción

especial (puesto que ahora los sub-índices no son un rango predefinido). Por ejemplo, podemos hacer una tabla con los accesos realizados por cada máquina en nuestra bitácora:

```
# Este es el archivo freq.awk
BEGIN{FS=","}

{
    tab_freq[$3] += 1
}

END {
    for (i in tab_freq)
 print tab_freq[i], i | "sort -rn"
}
```

Para ejecutar un archivo con un programa `awk` damos el comando:

```
% awk -f freq.awk < ftp.log
```

También existen alternativas de ejecución según expresiones regulares,basta preceder un trozo de programa (encerrado entre llaves) por un patrón de búsqueda (de hecho, `BEGIN` y `END` son patrones especiales que calzan con el comienzo y final del archivo). Sólo se ejecutan para cada línea los segmentos que calzan con el patrón:

```
# Este es el archivo freq2.awk
# Solo cuento los accesos de los hosts del dominio dcc.uchile.cl
BEGIN{FS=","}

/.*\.dcc\.uchile\.cl/ {
    tab_freq[$3] += 1
}

END {
    for (i in tab_freq)
         print tab_freq[i], i | "sort -rn"
}
```

Para invertir un patrón se le puede negar con un signo de exclamación (!).

```
# Este es el archivo freq3.awk
# No cuento los accesos de los hosts del dominio dcc.uchile.cl
```

```
BEGIN{FS=","}

! /.*\.dcc\.uchile\.cl/ {
    tab_freq[$3] += 1
}

END {
    for (i in tab_freq)
         print tab_freq[i], i | "sort -rn"
}
```

Las expresiones van siendo evaluadas una por una y todas las que calzan con la línea de entrada son evaluadas. Por ejemplo podemos contar el mismo total y además para el dominio local:

```
# Este es el archivo freq4.awk
# cuento los accesos de los hosts del dominio dcc.uchile.cl
# y del resto de los hosts
BEGIN{FS=","}

/.*\.dcc\.uchile\.cl/ {
    tab_dcc_freq[$3] += 1
}

{
    tab_freq[$3] += 1
}

END {
    print "Accesos Totales:"
    for (i in tab_freq)
         print tab_freq[i], i | "sort -rn"
    close("sort -rn")

    print "Accesos Locales:"
    for (i in tab_dcc_freq)
print tab_dcc_freq[i], i | "sort -rn"
}
```

El `close` es necesario para terminar un `sort` y comenzar otro. Si no se coloca, el sort se efectuará sobre ambas tablas concatenadas que no es el resultado que se busca.

2.3.2 perl

A pesar que awk permite mezclar expresiones regulares con un lenguaje de programación, no es fácil manipular patrones dentro de expresiones (como es el \(\) y \1 de vi). Como respuesta a la mayoría de las falencias de awk se creó un nuevo lenguaje hace pocos años, llamado perl.

Perl es un lenguaje diseñado para ejecutarse en un archivo, no en la línea de comandos como awk. Igual como un archivo de comandos, el archivo puede ser ejecutable, comenzando por la secuencia:

```
#!/usr/local/bin/perl
```

El lenguaje es mucho más flexible y general, pero también con una sintáxis más oscura. Veremos los ejemplos más triviales solamente.

Las principales diferencias con awk son: no hay un ciclo implícito, las variables ahora siempre comienzan con $ y no existe la separación en campos automática.

Existe un ciclo genérico que se usa casi siempre:

```
while(<>) {
...
}
```

Esto procesa la entrada estándar, o los archivos dados en argumento en forma automática. El ciclo entra una vez por línea, y la línea de entrada se recibe en una variable predefinida $_.

Para separar los campos, usamos condiciones con expresiones regulares. Esto nos permite procesar archivos mucho menos estructurados que nuestro ftp.log. Dentro de una expresión regular podemos encerrar entre paréntesis trozos de expresión y luego los referenciamos como $1, $2, ...

Por ejemplo, supongamos que la bitácora de ftp fuera con este formato (orientado a un humano, no a un programa):

```
11:33:35, host=araucaria size=231919 file=/pub/Pc/Sound/jp100.zip
```

Procesarla en awk se hace difícil puesto que no se puede usar un separador de campos trivial. En perl esto resulta muy simple, usando un if con expresión regular:

```
#!/usr/local/bin/perl

while(<>)
{
    if( /^([0-9]+:[0-9]+:[0-9]+), host=([^ ]+) size=([0-9]+) file=(.*)$/ ) {
        $date = $1;
```

```
        $host = $2;
        $size = $3;
        $file = $4;
        print "ftp from $host, file $file, $size bytes\n";
    }
}
```

Una vez que copiamos el valor de los campos a variables, estas son globales y puedo usarlas después. Por ejemplo, la tabla de frecuencia de los hosts puedo hacerla ahora con el archivo `freq.pl`:

```
#!/usr/local/bin/perl

while(<>)
{
    if( /.*host=([^ ]+).*/ ) {
        $freq_tab{$1} = $freq_tab{$1} + 1;
    }
}
foreach $key (keys %freq_tab) {
    print $freq_tab{$key}, $key;
}
```

El manejo de la entrada y salida es más flexible también. Por ejemplo el comando anterior funciona con su entrada estándar, con un archivo pasado como parámetro e incluso con una lista de archivos (que se procesan como si hubiesen sido concatenados).

```
% freq.pl <ftp.log
% freq.pl ftp.log
% freq.pl ftp1.log ftp2.log
```

Las salidas se pueden manipular asignando un nombre ellas, por ejemplo si queremos la tabla de frecuencia anterior ordenada de mayor a menor:

```
#!/usr/local/bin/perl

while(<>)
{
    if( /.*host=([^ ]+).*/ ) {
        $freq_tab{$1} = $freq_tab{$1} + 1;
    }
}
```

```
open(OUT, "| sort -rn");
# con select hacemos que print vaya hacia alla
select(OUT);
foreach $key (keys %freq_tab) {
    print $freq_tab{$key}, $key;
}
close(OUT);
```

También podemos manipular variables dentro de expresiones regulares, aunque la ejecución es más lenta (perl tiende a ser sorprendentemente rápido). La mayoría de los comandos básicos de Unix están disponibles como funciones de perl, llegando incluso a poderse hacer servidores de red en perl. Desgraciadamente, tanta flexibilidad le dan una sintáxis muy dura y difícil de entener.

La extensión y eficiencia del lenguaje le hacen un candidato ideal para hacer programas de un sólo uso, integrando los complejos scripts de shell, sed y awk en un solo ambiente.

Como ejemplo final, este es un procesador de la bitácora modificada de ftp, escrito en perl y realizando cuanta estadística se puede sobre los datos:

```
#!/usr/local/bin/perl
# El formato de las lineas es:
# Sat Sep 10 11:33:35 1994 host=araucaria size=231919 file=/pub/Pc/Sound/jp100.zip

while(<>)
{
if( $first_time == 0 && /^(... ... .. [0-9]+:[0-9]+:[0-9]+ ....)/ ) {
print "Anonymous ftp xferlog from $1\n";
$first_time = 1;
}

if( /^(... ... .. [0-9]+:[0-9]+:[0-9]+ ....) host=([^ ]+) size=([0-9]+)
file=([^ ]+)) {
$date = $1;
$host = $2;
$size = $3;
$file = $4;
$cnt++;
$tot_size += $size;
$to_host_sz{$host} += $size;
$to_host_n{$host}++;
$top_file{$file}++;
}
```

```
}

print "to $date\n\n";
print "Total bytes transfered: $tot_size, files: $cnt\n\n";
printf("Avrg rate: %.2f Bytes/s\n\n", $tot_size/$tot_time);

print "Size \t NFiles \t Host";

open(OUT, "|sort -rn");
select(OUT);
foreach $key (keys %to_host_sz) {
    print "$to_host_sz{$key} \t $to_host_n{$key} \t $key\n";
}
close(OUT);


print "\n\nCount\tFile Name\n";

open(OUT, "|sort -rn");
select(OUT);
foreach $key (keys %top_file) {
    print "$top_file{$key} \t $key\n";
}
close(OUT);
```