

Sintaxis expresiones regulares

- Una cadena de caracteres corresponde a sí misma.
'abc' encaja con 'abc' (en ese orden y en minúsculas).
- El carácter '.' representa cualquier carácter (alfanumérico, símbolo, etc.).
- [conjunto] encaja con cualquier carácter que aparezca en el conjunto (sólo uno).
[abc] corresponde a 'a', 'b' o 'c'. Puedo especificar rangos, por ejemplo, todas las letras minúsculas y mayúsculas: [a-zA-Z].
- [^conjunto] encaja con cualquier carácter que **no aparezca en el conjunto**.
[^abc] encaja con cualquier carácter individual que **no sea** 'a', 'b' o 'c'.

Sintaxis expresiones regulares

- `^` señala el principio de línea.
- `$` señala el final de línea.
- Si se concatenan dos regex, e_1 y e_2 , en una sola regex $e_1 e_2$, entonces se corresponde con una cadena si una parte de esa cadena se corresponde con e_1 y otra parte **contigua** encaja con e_2 .
 - Ejemplo: `[a-d]z` se corresponde con `'az'` y también con `'bz'`, `'cz'` o `'dz'`.

Sintaxis expresiones regulares

- `exp*` indica si aparece **cero o más veces** la regex que va delante.
- `exp+` indica si aparece **una o más veces** la regex que va delante.
- Ejemplos:
 - `a*` corresponde con "", 'a', 'aa', 'aaa', etc.
 - `ba+` corresponde con 'baa', 'baaa', etc.
 - Pero `ba+` no encaja con 'bb' (la a debe aparecer *al menos una vez*).
 - `ba*` encaja con 'bb' (el carácter a no tiene por qué aparecer).

Sintaxis expresiones regulares

- `exp?` indica si aparece **cero o una vez** la regex que va delante. Se usa para las partes opcionales del patrón.
- `(exp)` agrupa expresiones regulares.
- Ejemplos:
 - `az?` corresponde con `'a'`, `'az'`, `'av'`, etc.
 - `(ab)+` corresponde con `'ab'`, `'abab'`, etc. (deben aparecer **ambas** letras).

Sintaxis expresiones regulares

- `exp | exp` indica un or lógico, es decir, encaja con alguna de las regex separadas por el símbolo `|`.
- `\` carácter de escape, hace que el símbolo pierda su significado especial.
- Ejemplos:
 - `aass|booo` corresponde con 'aass' o 'booo'.
 - `a*` corresponde con 'a*', 'ho1a*', etc.

Comando egrep

- Propósito: Filtra líneas usando expresiones regulares.
- Sintaxis: `egrep [options] [regex] [files]`
- Opción `-v`
Hace lo opuesto: devuelve las líneas que no corresponden con la regex.
- Opción `-n`
Devuelve el número de línea en la que hay una cadena que encaja con la regex.
- Opción `-e`
Utiliza el siguiente argumento como una expresión. Si se usa varias veces, busca todos los patrones indicados. Se puede emplear para proteger una regex que empiece por `'-'`.
- Opción `-q`
Silencioso, no devuelve nada (pero nos da status de salida).
- Opción `-i`
Insensible a mayúsculas/minúsculas.

Comandos para sed?

- q sale del programa.
- d borra la línea.
- p imprime la línea (ejecutar con -n).
- r lee e inserta un fichero.
- s sustituye (**la más utilizada**).

Comandos para sed?

- Direcciones.
 - número → actúa sobre la línea cuyo número se indica.
 - /regex/ → devuelve líneas que corresponden con la regex.
 - \$ → representa la última línea.
- Uso de intervalos.
 - número,número → solo mira en ese intervalo de líneas.
 - número,\$ → procesa desde la línea número hasta la última.
 - número,/regex/ → procesa desde la línea número **hasta la primera línea que encaje con regex.**

Ejemplos con sed?

- `sed -E '3,6d'` → borra las líneas de la 3 a la 6.
- `sed -E -n '/BEGIN|begin/,/END|end/p'` → imprime las líneas entre las que coinciden con esas regex.
- `sed -E '3q'` → imprime las tres primera líneas.
- `sed -E -n '13,$p'` → imprime desde la línea 13 a la última.
- `sed -E '/[Hh]o!a/d'` → borra las líneas que contengan Ho!a u ho!a.

Sustituciones

- `sed -E 's/regex/sustitución/'` → busca la primera coincidencia con el patrón `regex` y la sustituye por la cadena `sustitución`.
- `sed -E 's/regex/sustitución/g'` → busca todas las subcadenas de la línea que encajan con la `regex` y las sustituye por la cadena `sustitución`.
- `sed -E 's/(regex)regex(regex).../ \1 sustitución\2/g'` → Las subcadenas que corresponden con las agrupaciones (entre paréntesis, en orden de apertura) se usan en la cadena de sustitución. Se llaman *backreferences* y su numeración es automática.
- `sed -E 's/regex/"&"/` → El carácter `'&'` en la cadena de sustitución indica la parte completa de la entrada que coincidió con el patrón de búsqueda de `regex`. En este ejemplo, se pone entre comillas la primera cadena que coincida con `regex`.

Ejemplos de sustitución con sed?

- `sed -E 's/[0-9]/X/'` → el primer dígito de la línea se sustituye por una 'X'.
- `sed -E 's/[0-9]/X/g'` → todos los dígitos de la línea se sustituyen por una 'X'.
- `sed -E 's/^[A-Za-z]+[]+([0-9]+)/NOMBRE:\1 NOTA:\2/g'` → añade NOMBRE: y NOTA: delante de los nombres y notas (ojo, no funciona con acentos, guiones, nombres con espacios. . .).
- `sed -E -n '/^(..)(..)\3\2\1$/p'` /usr/share/dict/words → busca palíndromos de 6 letras en el fichero /usr/share/dict/words.
- `echo 'James Bond' | sed -E 's/(.*) (.*)/My name is \2, \1 \2./'`

Imprimir

- `print`
Sentencia que imprime los operandos. Si e separan con ' , ' entonces inserta un espacio.
Al final imprime un salto de línea.
- `printf()`
Función que imprime permitiendo controlar el formato de la salida, de manera análoga a la función del mismo nombre de la biblioteca `libc` para C:

```
$ ls -l | awk '{ printf("Size: %08d KBytes\n", $5) }'
```

Variables

- `$0` es la línea que está procesando.
- `$1`, `$2` ... son el primer, segundo... campo de la línea, respectivamente.
- NR Número de línea (**N**úmero de **R**egistro) que se está procesando.
- Ejemplo para imprimir la tercera y segunda columna de un archivo CSV:

```
$ cat a.txt|awk -F, '{printf("%d%d", $3, $2)}'
```

Variables

- **FILENAME** nombre del archivo que se está procesando. Si es `stdin`, tiene el valor `'-'`.
- **NF** número de campos del registro que está procesando.
- **var=contenido** podemos declarar variables dentro del programa. Con el modificador `-v` se pueden pasar variables al programa.

```
$ ls -l | awk '  
{  
    size=$5 ; printf("Size: %08d KBytes\n", size)  
}',
```

Programas

- patrón { programa }

Procesa sólo las líneas que corresponden a un patrón. El patrón puede ser:

- Una expresión regular.
- Una expresión de relación, en la que se comparan valores y se evalúa la expresión.

Ejemplo con patrón regex

```
$ ls -l | awk '/[Dd]esktop/{ print $1 }'
```

```
$ ls -l | awk '$1 ~ /[Dd]esktop/ { print $1 }'
```

Ejemplo con patrón expresión de relación

```
$ ls -l | awk ' NR >= 5 && NR <= 10 { print $1 }'
```

Programas

- Marcas de inicio y finalización.

```
BEGIN{  
    comandos-awk  
}  
  
/patron/{  
    accion  
}  
  
END{  
    comandos-awk  
}
```

- next pasa a la siguiente regla.

Arrays asociativos

- En AWK, los arrays contienen múltiples pares clave/valor, es decir, son *asociativos*.
- Los índices del array no necesitan ser un conjunto de números consecutivos. Pueden ser cadenas de caracteres o números. Tampoco hay que especificar el tamaño del array.
- Sintaxis.

```
nombreamay[string]=valor
```

- `arrayname` es el nombre del array.
- `string` es el índice del array.
- `valor` es cualquier valor asignado al elemento del array.

Operadores unarios

- `+variable` → retorna la propia variable.
- `-variable` → cambia el signo del valor de la variable.
- `++variable` → pre-incremento automático (primero incremento y luego accedo al valor).
- `--variable` → pre-decremento automático (primero decremento y luego accedo al valor).
- `variable++` → post-incremento automático (primero accedo al valor y luego incremento).
- `variable--` → post-decremento automático (primero accedo al valor y luego decremento).

Operadores aritméticos

- `numero1 + numero2` → Suma.
- `numero1 - numero2` → Resta.
- `numero1 * numero2` → Multiplicación.
- `numero1 / numero2` → División.
- `numero1 % numero2` → Módulo (resto) de la división de `numero1` entre `numero2`.

Operadores con cadenas

- `cadena1 cadena2` → concatena los valores de `cadena1` y `cadena2`.

```
BEGIN {  
    string1="Audio";  
    string2="Video";  
    numberstring="100";  
    string3=string1 string2;  
    print "La cadena concatenada es:" string3;  
    numberstring=numberstring+1;  
    print "Cadena a número:" numberstring;  
}
```

Operadores de asignación

- `=` → asignación.
- `+=` → suma y asignación.
- `-=` → resta y asignación.
- `*=` → producto y asignación.
- `/=` → división y asignación.
- `%=` → módulo de división y asignación.

Operadores de comparación

- `>` → mayor que.
- `>=` → mayor o igual que.
- `<` → menor que.
- `<=` → menor o igual que.
- `==` → igual que.
- `!=` → distinto que.
- `&&` → *and* lógico (da *true* si los dos operandos son *true*).
- `||` → *or* lógico (da *true* si alguno de los dos operandos es *true*).

Operadores con regex

- `~ regex` → si hay coincidencia parcial con la regex.
- `!~ regex` → si no coincide con la regex.

Arrays asociativos

- Para acceder a todos los elementos del array se usa un bucle for sobre los índices del array.
- Si hago varias acciones en cada iteración, deben ir rodeadas por { }.

```
for (var in nombrearray)
    acciones
```

- Para borrar elementos del array:

```
delete nombrearray[index]  # Borra un elemento
# Borrar todos los elementos del array
for (var in nombrearray)
    delete nombrearray[var]
```

Arrays asociativos

- Para AWK los índices siempre son cadenas de caracteres, incluso aunque hayamos usado números en la inicialización.

```
BEGIN {  
    item[101]="HD Camcorder";  
    item[102]="Refrigerator";  
    item[103]="MP3 Player";  
    item[104]="Tennis Racket";  
    item[105]="Laser Printer";  
    item[1001]="Tennis Ball";  
    item[55]="Laptop";  
    item["na"]="Not Available";  
    print item["101"];  
    print item[102];  
    print item["103"];  
    print item[104];  
    print item["105"];  
    print item[1001];  
    print item[55];  
    print item["na"];  
}
```