PROGRAMACIÓN INTEGRATIVA

SEGUNDA SESIÓN PRÁCTICA

EJERCICIO RESUELTO

1. Usar el script del paso anterior para encontrar todos los ficheros dentro del directorio `$HOME` cuyo nombre cumpla las siguientes condiciones:

   a) Contener únicamente letras.

   b) No contener ningún número.

   c) Contener exactamente tres vocales, acentuadas o no.

   Conceptos básicos relacionados:

   - Expresiones regulares (ver Sección 2.3.1 de los apuntes).
   - Ejecución de flujo condicional (lazos e `if`s, Sección 2.4 de los apuntes).

   Comandos relacionados: (ver también las páginas de `man` para más información)

   - `basename`: comando que recibe como parámetro una ruta y elimina el árbol de directorios y, opcionalmente, el sufijo de la misma.
   - `xargs`: construye y ejecuta comandos a partir de entrada estándar.

   Esta semana nos centraremos en el estudio de las expresiones regulares, utilizando como apoyo el script desarrollado la semana pasada para buscar nombres de ficheros que cumplan un determinado patrón.

   a) Deseamos encontrar todos los ficheros cuyo nombre contenga únicamente letras. Podemos pensar que "letras" son todos aquellos caracteres entre `'a'` y `'z'`, tanto mayúsculas como minúsculas. Según esta idea, podríamos buscar los ficheros cuyo nombre contiene únicamente letras de la siguiente manera:

   ```
   [user@host:/usr]$ ej1.sh ".*\/[a-zA-Z]*$"
   ./include/X11/bitmaps/mailfullmsk
   ./include/X11/bitmaps/scales
   ./include/X11/bitmaps/Excl
   ./include/X11/bitmaps/mensetmanus
   ./include/X11/bitmaps/black
                                          ⋮
   ```

   En esta primera versión, estamos utilizando la expresión entre corchetes `'[a-zA-Z]'` para denotar "un caracter que puede ser cualquiera comprendido entre la `'a'` y la `'z'` minúsculas y la `'A'` y la `'Z'` mayúsculas". Más adelante veremos las limitaciones de este esquema. Lo más llamativo de esta expresión regular es que hemos tenido que cambiar la manera en la que expresamos que el patrón buscado debe aparecer tras la última `'/'` de la ruta. Si hubiésemos empleado la misma aproximación que en el ejercicio anterior (no permitir ningún caracter `'/'` tras el patrón buscado) esto es lo que hubiera sucedido:

```
[user@host:/usr]$ ej1.sh "[a-zA-Z]*[^/]*$"
./lib32/libm.a
./lib32/crt1.o
./lib32/gconv/IBM1025.so
./lib32/gconv/IBM851.so
./lib32/gconv/SHIFT_JISX0213.so
./lib32/gconv/EBCDIC-FI-SE-A.so
                                                    ⋮
```

Es decir, al intentar evitar que aparezca un caracter '/' tras el patrón principal, estamos inadvertidamente permitiendo que aparezcan cualquier otro tipo de caracteres (en este caso, puntos y dígitos). Para evitarlo, hemos tenido que cambiar la manera de "prohibir" la aparición de '/' después del patrón: se permite su aparición antes del mismo, pero para que el nombre del fichero sea válido, debe ser posible localizar un caracter '/' a partir del cuál sólo aparecen letras.

Idealmente, deberíamos poder especificar un patrón con la seguridad de que sólo será comparado con lo que aparezca tras la última barra. En el primer tutorial vimos una manera de extraer el texto tras la última barra utilizando `sed`. Existe una solución más sencilla mediante un comando pensado específicamente para ese propósito: `basename`.

```
[user@host:/usr]$ basename include/malloc.h
malloc.h
[user@host:/usr]$ find -type f|basename
basename: missing operand
Try 'basename --help' for more information.
```

El problema que nos encontramos en esta ocasión es que `basename` espera recibir sus parámetros como argumentos de línea de comandos, no a través de su entrada estándar. Se trata de un problema que surge con cierta frecuencia a la hora de programar scripts del shell. Una posible solución es utilizar `xargs`. Se trata de un comando pensado específicamente para recibir argumentos por entrada estándar e invocar un comando pasándole dichos argumentos por línea de comandos:

```
[user@host:/usr]$ find -type f|xargs -n 1 basename
libm.a
crt1.o
IBM1025.so
IBM851.so
SHIFT_JISX0213.so
                                                    ⋮
```

En este caso, con la opción `-n 1` estamos especificando que debe invocar a `basename` una vez por cada parámetro que reciba por entrada estándar. A partir de aquí podríamos escribir nuestro script como anteriormente:

```
[u@host:/usr]$ find -type f|xargs -n 1 basename|grep "^[a-zA-Z]*$"
mailfullmsk
scales
Excl
mensetmanus
black
                                                    ⋮
```

Pero, al igual que ocurría en el ejercicio de la semana pasada, hemos perdido la ruta completa al fichero, lo que además implica que puedan aparecer duplicados, entre otros

problemas. En este punto, una solución pasa por separar el ejercicio en dos: por una parte, almacenar la ruta al fichero y trocearla, comprobando si el nombre del mismo coincide con el patrón buscado; por otra, y en caso afirmativo, mostrar la ruta completa al fichero. Para esto, vamos a introducir los condicionales (lazos e `if`s, ver Sección 2.4 de los apuntes). En primer lugar, veamos cómo realizar una iteración de los elementos de un conjunto:

```
[user@host:/usr]$ for i in 1 2 3 4 5 6 7 8 9 10; \
do \
        echo "$i"; \
done
1
2
3
4
5
6
7
8
9
10
```

Vamos a iterar algo más útil: los resultados de la ejecución de otro comando. Esta funcionalidad, llamada "sustitución de comandos" (ver Sección 2.5.3) se realiza mediante la sintaxis `$(comando)`. El efecto es que, durante la ejecución del script, `comando` será ejecutado antes que la línea en la que se integra, y la salida estándar de su ejecución será sustituida antes de ejecutar la misma:

```
[user@host:/usr]$ for i in $(find -type f); \
do \
        echo "$i"; \
done
./lib32/libm.a
./lib32/crt1.o
./lib32/gconv/IBM1025.so
./lib32/gconv/IBM851.so
./lib32/gconv/SHIFT_JISX0213.so
                                        ⋮
```

A continuación, vamos a extraer el `basename` de cada uno de los ficheros dentro del cuerpo del bucle, y asignarlo a una nueva variable `N`:

```
[user@host:/usr]$ for i in $(find -type f); \
do \
        N="$(basename $i)"
        echo "$i -> $N"; \
done
./lib32/libm.a -> libm.a
./lib32/crt1.o -> crt1.o
./lib32/gconv/IBM1025.so -> IBM1025.so
./lib32/gconv/IBM851.so -> IBM851.soHIFT_JISX0213.so
./lib32/gconv/SHIFT_JISX0213.so -> SHIFT_JISX0213.so
                                        ⋮
```

Ahora podemos utilizar `grep` para intentar emparejar el contenido de `$N` con el patrón buscado. El problema que nos encontramos es, ¿cómo escribir una sentencia condicional

(`if`) que imprima la ruta al fichero dependiendo de si `grep` encuentra el patrón buscado o no? La solución nos viene dada por el status de salida (ver Sección 2.4.2) del comando `grep` usado con la opción `-q`. En este modo de funcionamiento, `grep` no proporciona ninguna salida estándar. En su lugar, comprueba si el patrón buscado se encuentra en la entrada y termina su ejecución devolviendo un valor `ÉXITO` (cero) o `FALLO` (cualquier valor distinto de cero). Los `if`s en un script del shell se escriben utilizando comandos a ejecutar en lugar de condiciones. Cuando los comandos se ejecutan con `ÉXITO`, el intérprete de comandos ejecuta la rama `then`. En caso contrario, se ejecutará la rama `else`. Para resolver el problema que nos ocupa, podemos usar este mecanismo del siguiente modo:

```
[user@host:/usr]$ for i in $(find -type f); \
do \
        N="$(basename $i)"
        if grep -q "^[a-zA-Z]*$" <<< "$N"; then
                echo "$i"; \
        fi \;
done
./include/X11/bitmaps/mailfullmsk
./include/X11/bitmaps/scales
./include/X11/bitmaps/Excl
./include/X11/bitmaps/mensetmanus
./include/X11/bitmaps/black
                                        ⋮
```

Destaca la sintaxis `comando <<< "string"`, no vista hasta el momento. Es lo que se denomina un `HERESTRING` (ver Sección 2.5.1.1 de los apuntes), y se utiliza para proporcionar un string sencillo como entrada estándar a un comando. A pesar de ser ampliamente aceptado por la mayor parte de los intérpretes de comandos en la actualidad, debe hacerse notar que no es parte del estándar POSIX, por lo que podría no ser aceptado por algún intérprete particular (sobre todo versiones antiguas). En particular, en algunos sistemas, la implementación de `/bin/sh` podría no ser compatible con esta sintaxis. Haciendo uso de este mecanismo, podemos reescribir el script `ej1.sh` del ejercicio anterior como:

```
[user@host:~]$ cat > ej2.sh << "EOF"
#! /bin/bash

for i in $(find -type f);
do
        N="$(basename $i)"
        if grep -q "$1" <<< "$N";
        then
                echo "$i";
        fi
done
EOF
```

Nótese que hemos indicado que debe usarse como intérprete `/bin/bash`, para evitar que un `/bin/sh` incompatible falle al ejecutar el código. Destaca también el uso de `''EOF''` entre comillas en la primera línea, que indica que el intérprete no debe intentar realizar ninguna sustitución en el cuerpo del `HEREDOC` (ver Sección 2.5.1.1 de los apuntes). A partir de aquí, podemos utilizar `ej2.sh` para resolver el ejercicio. Por ejemplo:

```
[user@host:/usr]$ . ~/ej2.sh "^[a-zA-Z]*$"
./include/X11/bitmaps/mailfullmsk
./include/X11/bitmaps/scales
```

```
./include/X11/bitmaps/Excl
./include/X11/bitmaps/mensetmanus
./include/X11/bitmaps/black
                                        ⋮
```

En cuanto al patrón en si, destacar que las expresiones entre corchetes POSIX varían su semántica dependiendo del `locale` utilizado. Por ejemplo:

```
[user@host:~/test]$ touch ficherouno
[user@host:~/test]$ touch ficherodos
[user@host:~/test]$ touch ficherótrés
[user@host:~/test]$ export LC_ALL="gl_GL.utf8"
[user@host:~/test]$ ls
ficherodos  ficherótrés  ficherouno
[user@host:~/test]$ . ~/ej2.sh "^[a-zA-Z]*$"
./ficherouno
./ficherodos
./ficherótrés
[user@host:~/test]$ export LC_ALL="C"
./ficherouno
./ficherodos
```

b) En cuanto a la búsqueda de nombres de ficheros que no contienen ningún número, una vez programado el script `ej2.sh` la solución es sencilla:

```
[user@host:/usr]$ . ~/ej2.sh "^[^0-9]*$"
./lib32/libm.a
./lib32/gconv/EBCDIC-FI-SE-A.so
./lib32/gconv/EBCDIC-DK-NO.so
./lib32/gconv/EBCDIC-FI-SE.so
./lib32/gconv/EBCDIC-UK.so
                                        ⋮
```

Otra posibilidad es utilizar las *clases de caracteres* proporcionadas por POSIX. Se trata de un mecanismo para especificar que queremos aceptar o rechazar, por ejemplo, cosas como "caracteres alfabéticos", "caracteres numéricos", "caracteres imprimibles", etc. (ver Sección 2.3.1.1 de los apuntes). Así, podríamos resolver este apartado de la siguiente forma:

```
[user@host:/usr]$ . ~/ej2.sh "^[^[:digit:]]*$"
./lib32/libm.a
./lib32/gconv/EBCDIC-FI-SE-A.so
./lib32/gconv/EBCDIC-DK-NO.so
./lib32/gconv/EBCDIC-FI-SE.so
./lib32/gconv/EBCDIC-UK.so
                                        ⋮
```

c) Este apartado nos pide encontrar nombres de archivos que contengan exactamente tres vocales, acentuadas o no. Comencemos en primer lugar por resolver el problema de encontrar todos los nombres de ficheros que contengan una `'a'`, acentuada o no. Asumiendo un locale español, podríamos escribirlo del siguiente modo:

```
[user@host:/usr]$ . ~/ej2.sh "[aá]"
./lib32/libm.a
./lib32/gconv/gconv-modules.cache
./lib32/libc.a
./lib32/libdl.a
```

```
./lib32/libutil.a
                                                    ⋮
```

Esta no es una solución portable, pues si escribiésemos, por ejemplo, en francés, tendríamos que incluir otros tipos de acentuación de la `'a'` utilizados en ese lenguaje, como `à` o `â`. Para evitar este tipo de consideraciones podemos utilizar un nuevo concepto: las clases de equivalencia de caracteres (ver Sección 2.3.1.1) de los apuntes. Podemos escribir la expresión regular para "cualquier caracter `a` con cualquier tipo de acentuación propia del locale actual" como `[=a=]`:

```
[user@host:/usr]$ . ~/ej2.sh "[[=a=]]"
./lib32/libm.a
./lib32/gconv/gconv-modules.cache
./lib32/libc.a
./lib32/libdl.a
./lib32/libutil.a
                                                    ⋮
```

Por tanto, podemos buscar los nombres de ficheros que contengan al menos una vocal acentuada o no del siguiente modo:

```
[user@host:/usr]$ . ~/ej2.sh "[[=a=][=e=][=i=][=o=][=u=]]"
./lib32/libm.a
./lib32/crt1.o
./lib32/gconv/IBM1025.so
./lib32/gconv/IBM851.so
./lib32/gconv/SHIFT_JISX0213.so
                                                    ⋮
```

En este punto, la mayor dificultad radica en especificar que queremos nombres de ficheros que contengan exactamente tres vocales, ni una más ni una menos. Vamos a definir una variable `VOCAL` que nos permite escribirlo de forma compacta del siguiente modo:

```
[u@host:/usr]$ export V="[=a=][=e=][=i=][=o=][=u=]"
[u@host:/usr]$ . ~/ej2.sh "^[^$V]*[$V][^$V]*[$V][^$V]*[$V][^$V]*$"
```

La expresión puede interpretarse del siguiente modo: "opcionalmente, caracteres no vocales; una vocal; opcionalmente, caracteres no vocales; otra vocal; opcionalmente, caracteres no vocales; una tercera vocal; y opcionalmente, caracteres no vocales".

Como puede observarse, el uso de expresiones regulares básicas (BREs) para expresar este patrón no es del todo cómodo. Dado que queríamos buscar nombres con exactamente tres vocales hemos tenido que construir un patrón con 7 componentes (las tres vocales encerradas entre grupos de no vocales potenciales). Si hubiéramos querido que el número de vocales fuera superior el patrón crecería linealmente, convirtiéndose en inmanejable. Para mejorar esta situación, vamos a agrupar las partes repetidas del patrón en una expresión entre paréntesis y requerir que aparezcan exactamente tres veces, es decir:

```
[u@host:/usr]$ . ~/ej2.sh "^[^$V]*\([$V][^$V]*\)\{3\}$"
```

Nótese cómo, de esta manera, es muy sencillo modificar el patrón para hallar nombres de ficheros con un número de vocales arbitrario.

EJERCICIOS PROPUESTOS

2. Modificar la expresión regular del último apartado del ejercicio anterior para que se encuentren los ficheros con **al menos** tres vocales, en lugar de con tres vocales exactamente.