PROGRAMACIÓN INTEGRATIVA

SEGUNDA SESIÓN PRÁCTICA: CONTENIDO DE AMPLICACIÓN

Este documento pretende profundizar en la creación de scripts dando algunos consejos básicos a la hora de escribir scripts. También se incluyen algunos ejercicios de expresiones regulares.

1. Cómo escribir scripts

En esta sección daremos algunos consejos a la hora de escribir *scripts*. La idea es hacer un esqueleto básico: comprobación de argumentos y de errores.

Comprobación de argumentos

Los *scripts* que programamos pueden tener argumentos tanto opcionales como obligatorios, de ambos tipos o ningún tipo de argumento. En caso de que tengamos argumentos obligatorios, lo lógico es que el script no se ejecute y lance un mensaje de error. En el caso de argumentos opcionales deberían existir opciones por defecto. Del mismo modo, los argumentos pueden ser posicionales, es decir, siguen un orden estricto, o pueden tener un alias, de modo que aparece forma explícita el nombre de cada uno de los argumentos.

Bash nos permite comprobar saber el número de parámetros pasados utilizando la variable `$#`, De la misma forma, la lista de argumentos se puede iterar con la variable `$@`. De esta manera, haciendo referencia a los argumentos obligatorios, una posible forma muy simple de comprobarlos (en caso de ser posicionales) sería:

```
#!/bin/bash

# tres argumentos obligatorios
if [ $# != 3 ];
then
    echo "Error: wrong number of parameters"
else
    VAR1 = $1
    VAR2 = $2
    VAR3 = $3
fi
```

De esta manera, le decimos al usuario que el número de argumentos obligatorios no es el correcto. En caso de argumentos con alias, bash permite usar `getopts`, que es una forma fácil de parsear los argumentos de entrada utilizando un loop `while-do`. La sintaxis es `getopts OPTSTRING VARNAME`, donde `OPTSTRING` es una cadena que indica el nombre de las opciones y la existencia o no parámetros (se detalla a continuación), y `VARNAME` la variable sobre la que se itera. Un ejemplo de uso:

```
#!/bin/bash

while getopts "ax:" opt; do
  case $opt in
    a)  echo "-a was triggered" ;;
    x)  echo "-x was triggered with option $OPTARG" ;;
    \?) echo "Invalid option: -$OPTARG" ;;
  esac
done
```

Como vemos, `getopts` recibe una cadena con el formato de los parámetros de entrada. En nuestro caso, la `OPTSTRING "ax:"` indica que uno de las opciones es `"-a"`, sin ningún parámetro detrás y `"-x <param>"`; cualquier otra opción será considerada como error. `"ax"` implicaría que ni "a" ni "x" llevan parámetro detrás. Del mismo modo, `"a:x:"` llevarían parámetro después de la opción.. `getopts` es una buena forma de asegurarnos que 1) las opciones escogidas son correctas y 2) que no se especifican opciones que no se conocen.

Existe, además, una sintaxis especial de `OPTSTRING` y es utilizar `:` al comienzo, e.g. `:ax:` en el ejemplo. Esto cambiaría la forma *verbose* de reportar errores en el parseo de `getopts` a *silent*. La diferencia básica es que evita mostrar mensajes de error por defecto. Para más información consultar `https://wiki.bash-hackers.org/howto/getopts_tutorial#error_reporting`.

Todavía quedaría una tercera opción para parsear argumentos: `getopt` (no confundir con `getopts`). Esta herramienta pertenece al paquete `util-linux` y, básicamente, extiende la funcionalidad de `getopts` permitiendo parsear cómodamente argumentos largos. La sintaxis podría ser tal que así:

```
# DEPENDENCY: getopt command tool
TMP=‘getopt --options ab:c:d --long arga,argb:,argc:,argd,arge \
             -n "$SCRIPTNAME"\
             -- "$@"‘
# Note the quotes around ‘$TMP’: they are essential!
eval set -- "$TMP"
```

Analizando el código anterior por partes:

- `-options ab:c:d -long arga,argb:,argc:,argd,arge`: las opciones en formato corto y largo. Se ve que no hay equivalencia 1:1 (e.g. `-arge` no tiene versión corta), pero la sintaxis es muy parecida a la vista en `getopts`.

- `- "$@"`: pasar los argumentos "tal cual" a `getopt`. Las comillas dobles son esenciales para separar los parámetros de forma adecuada.
- `eval set - "$TMP"`: parte de la magia reside aquí. `set - "$TMP"` sirve para cambiar los argumentos posicionales del script ($1, $2, etc.) por los que devuelve la ejecución de `"$TMP"`. `eval` permite que no se sobrescriba el valor retornado por `getopt`.

De esta manera, podemos parsear las opciones y sus parámetros utilizando un bucle, como se muestra a continuación:

```
# default values for options
ARGB="";ARGC=0;
while true; do
  case "$1" in
    -a | --arga ) shift ;;
    -b | --argb ) ARGB=$2; shift 2 ;;
    -c | --argc ) ARGC=$2; shift 2 ;;
    -d | --argd ) shift ;;
    --arge ) ARGE=true; shift ;;
    -- ) shift; break ;;
    * ) break ;;
  esac
done
```

Difiere un poco de la primera versión que vimos utilizando `getopts`, dado que es necesario realizar un `shift` a cada opción que leemos (`shift 2` si, aparte de opción, leemos un parámetro).

Quedaría a elección del programador qué forma de comprobar los argumentos es más ventajosa. Recalcar que no es crucial para la funcionalidad del código, pero sí que puede ayudar a la hora de depurar el código y de mantener el mismo. Obviamente, pensando en scripts con pocas LOC, lo más probable, es que no sea necesario este tipo de comprobaciones.

Si el lector quedase con dudas, se recomieda la lectura del siguiente tutorial:

`https://wiki.bash-hackers.org/howto/getopts_tutorial`

Atrapando señales: `trap`

Los comandos que se ejecutan en el script podrían devolver valores, que podrían ser correctos o no. Por otra parte, los scripts también pueden recibir señales que pueden alterar el flujo de ejecución, por ejemplo, interrumpiéndolo. Hay dos formas básicas de atrapar señales: usando `set -e`, que básicamente termina el script cuando cualquier de los comandos termina su ejecución con un código distinto de cero; y con `trap [COMMAND] [SIGNAL]`, de manera que cada vez que termina cualquier de los comandos con código `SIGNAL`, se ejecuta el comando o función `COMMAND`. Por ejemplo:

```
#!/bin/sh
```

```
## remove al temporal files created
cleanup() {
  echo "caught signal: cleaning up..."
  rm -rf /tmp/*.tmp
  echo "done cleanup! quitting..."
  exit 1;
}

## catch signals and execute cleanup
trap cleanup 1 2 3 6

## main loop
for i in *
do
  sed -e "s/FOO/BAR/g" $i > /tmp/${i}.tmp && mv /tmp/${i}.tmp $i
done
```

Este *script* ejecuta la función `cleanup` cuando recibe la señal 1, 2, 3 o 6; que básicamente limpia los ficheros temporales que se crean en el comando del bucle principal, por si acaso quedase algún fichero intermedio.

De nuevo, este tipo de comprobaciones tampoco son cruciales en la mayoría de casos para la funcionalidad de nuestra herramienta o script. Con todo, en proyectos grandes es interesante e importante hacer este tipo de comprobaciones.

Esqueleto básico

Para una mejor lectura, se adjunta un esqueleto básico para programar nuestros scripts en la página 6. De la misma forma, se ha subido el código en un fichero a la plataforma Moodle.

- Cómo usar `getopts` - `https://wiki.bash-hackers.org/howto/getopts_tutorial`

2. Ejercicios: expresiones regulares

A continuación se dejan unos ejercicios para una mejor compresión de las expresiones regulares (BREs y EREs) vistas en las clases magistrales. Algunas posibles soluciones se adjuntan en la última página de este documento. Imaginemos que tenemos el siguiente texto:

```
En mi cuenta corriente de EE.UU. (o U.S.A) hay menos de 1.23e+08 euros (ó
1.43e+08 dólares); de hecho no tengo ni un euro.
```

1. Escribe el comando y expresión regular que utilizarías para extraer los números en notación científica

2. Escribe el comando y expresión regular que utilizarías para quitar los puntos de las las siglas en el texto (teniendo en cuenta que las siglas van en mayúsculas siempre)

3. Escribe el comando y expresión regular que utilizarías para cambiar las monedas a su correspondiente símbolo.

Enlaces de interés:

- Ejercicios interactivos de expresiones regulares - `https://regexone.com/`

A. Esqueleto básico

```
#!/bin/bash
SCRIPTNAME="template.sh"
SCRIPTVERSION="v0.1"

# Print usage
usage() {
  echo -n "$SCRIPTNAME [OPTION]...

main options are described below.

 Options:
  -v,--verbose     Verbose output
  -d,--debug       Debug output
  -h,--help        Display this help and exit
  --version     Displays versions and exits
"
}

version() {
  echo -n "$SCRIPTNAME $SCRIPTVERSION (C) 2019. No warranty."
}

# DEPENDENCY: getopt command tool
TEMP=‘getopt -o vdh --long verbose,debug,help,version \
             -n "$SCRIPTNAME" -- "$@"‘
# Note the quotes around ‘$TEMP’: they are essential!
eval set -- "$TEMP"

# default values for options
while true; do
  case "$1" in
    -v | --verbose ) VERBOSE=true; shift ;;
    -d | --debug ) DEBUG=true; set -x; shift ;;
    -h | --help ) usage; exit 1 ;;
    --version ) version; exit 1 ;;
    -- ) shift; break ;;
    * ) break ;;
  esac
done

################################################
### error handling                           ###
################################################
func_unexpected() {
    echo "Error due to an unexpected error";
    echo "Try again or consult the manual";
    exit 1;
}


################################################
### signals                                  ###
################################################
trap "func_unexpected" 1 # Generic code error
...

# main body
...

# exit
exit 0;
```

B. Soluciones expresiones regulares

1. Escribe el comando y expresión regular que utilizarías para extraer los números en notación científica

Solución:

```
grep -o "[0-9]\.[0-9]\{2,\}e+[0-9]\{2\}" file
```

Faltaría también dejar claro el formato que estamos utilizando, e.g: X.XXe[+,-]XXX. Es importante evitar las ambigüedades y dejar claro los casos de uso de los patrones.

2. Escribe el comando y expresión regular que utilizarías para quitar los puntos de las las siglas en el texto (teniendo en cuenta que las siglas van en mayúsculas siempre)

Solución:

```
sed -e "s/\([A-Z]\+\)\./\1/g" file
```

3. Escribe el comando y expresión regular que utilizarías para cambiar las monedas a su correspondiente símbolo.

Solución:

```
sed -e "s/dólar\(es\)\?/\$/g" -e "s/euro[s]\?/\€/g" file
```