Orientación del Proyecto 2 (*Shell*)

El objetivo del proyecto es hacer un *shell* que simule los *shells* (bash, sh, zsh, etc.) del sistema operativo Linux. El proyecto se debe entregar antes de las 11:59pm del **domingo 19 de diciembre**, en el EVEA. Los equipos son de 2 estudiantes.

NOTA: No se puede usar la función `system()`

Resumen de las funcionalidades

- **basic**: funcionalidades de la 1 a la 8 (3 puntos) **[REQUERIDO]**
  - `cd / exit / > / < / >> / | / #`
- **multi-pipe**: funcionalidad 9 (1 punto) (total 4 puntos)
- **background**: funcionalidad 10 (0.5 puntos) (total 4.5 puntos)
  - `jobs / fg / &`
- **spaces**: funcionalidad 11 (0.5 puntos) (total 5 puntos)
- **history**: funcionalidad 12 (0.5 puntos) (total 5.5 puntos)
  - `history / again`
- **ctrl+c**: funcionalidad 13 (0.5 puntos) (total 6 puntos)
- **chain**: funcionalidad 14 (0.5 puntos) (total 6.5 puntos)
  - `true / false / ; / && / ||`
- **if**: funcionalidad 15 (1 punto) (total 7.5 puntos)
  - `if / then / else / end / true / false`
- **multi-if**: funcionalidad 16 (0.5 puntos) (total 8 puntos)
- **help**: funcionalidad 17 (1 punto) (total 9 puntos) **[REQUERIDO]**
  - `help`
- **variables**: funcionalidad 18 (1 punto) (total 10 puntos)
  - `set / get / unset / ``

Detalles de las funcionalidades

1. Al ejecutar el programa, debe imprimir un *prompt* (ej: `mishell$ `, `my-prompt $ `, etc.).

- Siempre debe haber un signo `$` al final del prompt.
- Debe haber un espacio entre el signo `$` y el comando se que va a escribir.
- **keyword**: **basic**

```
user@machine $ gcc tarea2.c
user@machine $ ./a.out
my-prompt $
```

como debe lucir el shell cuando se ejecuta

2. Debe ejecutar comandos e imprimir la salida de estos.

- **keyword**: **basic**

```
user@machine $ ./a.out
my-prompt $ ls
file1 file2 file3 dir1  dir2 file4
my-prompt $
```

ejecución de comandos

3. Debe poderse ejecutar el comando `cd` (y que cambie de directorio).

- **keyword**: **basic**
- **built-in**: **cd**

```
user@machine $ ./a.out
my-prompt $ pwd
/some/directory
my-prompt $ cd /new/path
my-prompt $ pwd
/new/path
my-prompt $
```

comando 'cd'

4. Redirigir entrada y salida estándar de comandos hacia/desde ficheros con **`>` `<` `>>`.**

- **keyword**: **basic**

```
user@machine $ ./a.out
my-prompt $ command > file
my-prompt $ command < file
my-prompt $ command >> file
my-prompt $ command < file1 > file2
my-prompt $
```

redirección de entrada y salida

5. Usar una tubería `|` (Redirigir la salida estándar de un comando hacia la entrada de otro).

- **keyword**: **basic**

```
user@machine $ ./a.out
my-prompt $ command1 | command 2
my-prompt $ command1 | command 2 > file
my-prompt $
```

una tubería

6. Una vez terminado el comando previo, se debe mostrar el *prompt* nuevamente.

- **keyword**: **basic**

7. Solamente habrá un espacio entre cada uno de los *tokens* de la entrada.

- comandos
- parámetros
- operadores de redirección
- **keyword**: **basic**

8. Además.

- Si aparece un caracter `#` el *shell* debe ignorar desde ese caracter hasta el final de la línea.
- El *shell* debe terminar con el comando `exit` **.**
- **keyword**: **basic**
- **built-in**: **exit**

NOTA: Hacer todos los puntos **del 1 al 8 equivale a 3 puntos** en el proyecto.

NOTA: Cada funcionalidad debe ser compatible con todas las funcionalidades anteriores.

9. Implementar más de una tubería (1pt adicional)

- **keyword**: **multi-pipe**

```
user@machine $ ./a.out
my-prompt $ command1 | command2 | command3
my-prompt $ command1 < file1 | command2 | command3 | command4 > file2
my-prompt $
```

más de una tubería

10. Implementar el operador `&` y tener procesos en el *background* (0.5pt adicional)

- El comando **`jobs`** → lista todos los procesos que estan corriendo en el *background.*
- El comando **`fg <pid>`** → envia el proceso <pid> hacia el *foreground.*
- **`fg`** (sin parametros) → envia el proceso más reciente que fue al *background*, hacia el *foreground.*
- **keyword**: **background**
- **built-in**: **jobs**
- **built-in**: **fg**

NOTA: Aquí tiene que funcionar algo como `$ command1 < file1 | command2 > file2 &`

```
user@machine $ ./a.out
my-prompt $ command1 &
my-prompt $ jobs
[1] command1
my-prompt $ command2 &
my-prompt $ jobs
[1] command1
[2] command2
my-prompt $ fg # aqui el command2 va hacia el foreground
my-prompt $    # si el prompt vuelve a salir es porque el command2 terminó
```

processos en el background y &

11. Cualquier cantidad de espacios entre los comandos y parámetros. **(0.5pt adicional)**

- **keyword**: **spaces**

```
user@machine $ ./a.out
my-prompt $ command1 | command2
my-prompt $ command1|command2
my-prompt $ command1     |   command2
My-prompt $
```

entrada sin restricciones de espacios

12. Implementar un historial de comandos (un comando **`history`**) que permita imprimir los últimos 10 comandos ejecutados. **(0.5pt adicional)**

- Al ejecutar **`history`** se debe imprimir al lado de cada comando anterior un número. consecutivo indicando el orden en que se ejecutaron (el más antiguo es el 1).
- La historia debe mantenerse en corridas sucesivas del *shell* (si cierro el *shell* y lo vuelvo a abrir, la historia debe poder ser consultada).
- Implementar además un comando **`again <number>`** que vuelve a ejecutar el comando `<number>` de la historia.
- El comando **`again`** lo que guarda en la historia es el comando que ejecutó.
- El comando **`history`** también se guarda en el *history.*
- Si el comando empieza con espacio, entonces no se almacena en el historial.
- El historial solo debe mostrar los últimos 10 comandos ejecutados, cuando se hayan ejecutado más de 10, saldrán enumerados a partir de 1 los últimos 10.
- **keyword**: **history**

- **built-in**: **history**
- **built-in**: **again**

```
user@machine $ ./a.out
my-prompt $ command1
my-prompt $ command2
my-prompt $  command-x # hay un espacio al comienzo, esto no se guarda
my-prompt $ command3 | command4
my-prompt $ history
1: command1
2: command2
3: command3 | command4
4: history
my-prompt $ command5
my-prompt $ exit
user@machine $ ./a.out
my-prompt $ history
1: command1
2: command2
3: command3 | command4
4: history
5. command5
6: exit
7: history
my-prompt $ again 1 # esto ejecuta el commmand 1
my-prompt $ history
1: command1
2: command2
3: command3 | command4
4: history
5: command5
6: exit
7: history
8: command1 # en vez de guargar "again 1" guarda "command1"
9: history
my-prompt $ command6
my-prompt $ command7
my-prompt $ command8 | command9
my-prompt $ history
1: history
2: command5
3: exit
4: history
5: command1
6: history
7: command6
8: command7
9: command8 | command9
10: history
my-prompt $
```

history

13. Implementar la posibilidad de dar `Ctrl+C` en el *prompt* y que eso "mate" al proceso que se está ejecutando **(0.5pt adicional)**

- El *prompt* no se debe "morir" al apretar `Ctrl+C`.
- El proceso que se está ejecutando debe recibir la señal `SIGINT` y si ese proceso decide ignorar la señal, pues continuará ejecutándose.
- La 2da vez que se apriete `Ctrl+C` a un proceso en ejecución ya se le manda una señal de terminación total `SIGKILL`.
- **keyword**: **ctrl+c**

```
user@machine $ ./out.out
my-prompt $ command1
...       # command1 se está ejecutando
...       # command1 continúa ejecutándose
...       # command1 no hace nada con la señal (SIG_INT)
Ctrl + C # termina el command1
my-prompt $ command2
...       # command2 se está ejecutando
...       # command2 continúa ejecutándose
...       # command2 captura e ignrora la señal (SIG_INT)
Ctrl + C # el command2 sigue ejecutándose
...       # command2 continúa
Ctrl + C # segundo Ctrl+c el commando es terminado forzosamente
my-prompt $
```

Ctrl + C

14. Operadores para encadenar comandos `;` `&&` `||` (0.5pt adicional).

- Con el operador `;` se pueden ejecutar varios comandos en una sola línea, uno a continuación del otro, en el orden en que aparecen.
- El operador `&&` permite poner 2 (o más) comandos unidos de forma que, el comando *N* solo se ejecuta si el comando *N-1* tiene un *exit status* satisfactorio (cero). O sea, funciona como el operador *AND* lógico en corto circuito.
- En el caso del operador `&&` si el primer comando retorna `TRUE` (cero) entonces se ejecuta el 2do y el valor de retorno de la expresión entera es el valor de retorno (*exit status*) del último comando de la cadena. Si el primer comando retorna `FALSE` (distinto de cero) entonces el 2do comando nunca se ejecuta. El valor de retorno de la cadena de comandos es igual al del 1er comando.
- En el caso del operador `||` funciona como el operador *OR* lógico en corto circuito. Si el primer comando es satisfactorio, no hace falta ejecutar el 2do.Y sólo si el 1er comando "falla" (*exit status* distinto de cero) es que se ejecuta el 2do comando. Siendo el valor de retorno de la cadena, el valor de retorno del último comando que se ejecuta.
- implementar además un comando `true` que no hace nada, solo retorna (*exit status*) cero.
- implementar además un comando `false` que no hace nada, solo retorna uno.
- **keyword**: **chain**
- **built-in**: **true**
- **built-in**: **false**

```
user@machine $ ./out.out
my-prompt $ command1; command2; command3
...       # se ejecuta command1 el cual imprime su salida
...       # luego se ejecuta el command2 que tambien imprime su salida
...       # y por ultimo el command3
my-prompt $ command4 && command5 # si el 1ro falla el 2do no se ejecuta
my-prompt $ command6 || command7 # solo se ejecuta el 2do si el 1ro falla
my-prompt $ command8 && command9 || command10
# si el 1ro falla no se ejecuta más nada
# si el 2do funciona no se ejecuta más nada
my-prompt $ command11 || command12 && command13
# si el primero funciona no se ejecuta más nada
# si el 2do falla no se ejecuta más nada
my-prompt $ command14; command1; command16 &
my-prompt $ command17 && command18; command19 || command20
my-prompt $
```

encadenando comandos con ;, && y ||

NOTA: Si se va a combinar con los procesos en el *background*, hay que tener en cuenta que no hace falta implementar el caso en el que el operador `&` aparezca en un comando que no sea el último de la cadena.

NOTA: No hay que implementar paréntesis.

NOTA: Al combinar esto con las tuberías, el operador `;` tiene menos prioridad que el `|`, por lo tanto en casos como `comando1 | comando2; comando3 | comando4` primero se ejecuta lo que está antes del `;` (con todas las tuberías) y luego lo que está despues del `;`.

15. Expresiones condicionales. `if then else end` **(1pt adicional)**.

- La idea es que en una sola línea se pueda hacer una operación condicional.
- Siempre aparecerán los operadores `if` `then` y `end`.
- El `else` es opcional en la expresión (hay que implementarlo, pero no es obligatorio ponerlo en la expresión para que ésta sea válida).
- Entre el `if` y el `then` aparece un comando (o cadena de comandos) y la expresión evalúa `TRUE` si el valor de retorno (*exit status*) es cero.
- Solo si la expresión del `if` es `TRUE` se evalúa el comando que está despues del `then`.
- Si aparece un `else` entonces debe ejecutarse el comando (o cadena de comandos) que aparece despues del `else` sólo en caso de que la expresión condicional haya sido `FALSE` (*exit status* distinto de cero)
- implementar además un comando `true` que no hace nada, solo retorna (*exit status*) cero.
- implementar además un comando `false` que no hace nada, solo retorna uno.
- **keyword**: **if**
- **built-in**: **if**
- **built-in**: **true**
- **built-in**: **false**

```
user@machine $ ./out.out
my-prompt $ if command1 then command2 else command3 end
my-prompt $
```

if then else end

NOTA: Solo es necesario implementar un `if then else end`.

NOTA: los comantos `true` y `false` también aparecen en la funcionalidad 14.

16. multiples condicionales en una sola linea (0.5pt adicional)

- Se debe permitir anidar operadores `if` uno dentro de otro con varios niveles de profundidad.
- Cada `if` tiene su propio `end`.
- **keyword**: **multi-if**

```
user@machine $ ./out.out
my-prompt $ if command1 then command2 else if command3 then command4 else
command5 end end
my-prompt $
```

multiples if en una sola linea

17. Imprimir una ayuda (comando `help`) de lo que hace el *shell* y de las funcionalidades que tiene y si hay que tener alguna consideración especial **(1pt adicional)**.

- Teniendo en cuenta que cada equipo puede tener una combinación diferente de funcionalidades, cada ayuda será diferente y tendrá solo, lo que el *shell* es capaz de hacer.

- En esa ayuda aparecerá tambien, los nombres y grupos de los integrantes.
- El comando `help` sin parámetros muestra la lista de funcionalidades implementadas. Un *keyword* por línea. **En este documento cada funcionalidad tiene especificado el correspondiente *keyword*.**
- El comando `help <keyword>` puede recibir un parámetro (*keyword* de la funcionalidad) y ahi se explicará que hace esa fucionalidad, como lo hace, etc.
- Queremos que la ayuda sea lo mas explicativa y explícita posible, **una especie de informe** de cada una de las funcionalidades implementadas. En vez de entregarnos un documento *Word* o un *pdf* queremos que el propio *shell* diga lo que hace, bien redactado, con ejemplos si hace falta, detalles de implementación, cosas que funcionan bien, y cosas que no funcionan como es debido, etc.
- La ayuda debe imprimir además todos los comandos *built-in* implementados.
- La ayuda debe imprimir además la puntuación de cada funcionalidad y la cantidad máxima de puntos que tiene el *shell*.
- El *keyword* `help` debe aparecer en la ayuda también.
- **keyword**: **help**
- **built-in**: **help**

```
user@machine $ ./out.out
my-prompt $ help
Integrantes: Anthony Edward Stark y Steve Grant Rogers

Funcionalidades:
basic: funcionalidades básicas (3 puntos)
multi-pipe: multiples tuberías (1 punto)
ctrl+c: captuar y enviar señales a procesos (0.5 puntos)
help: ayuda (1 punto)

Comandos built-in:
cd: cambia de directorios
exit: termina el shell
help: muestra esta ayuda

Tota: 5.5 puntos

my-prompt $ help ctrl+c

Nuestro shell permite que cuando un comando se esté ejecutando se le pueda
enviar un Ctrl+C. Nuestro shell captura la señal …. bla bla bla.

Para implementar eso, tuvimos que primero crear un manejador de señarles
(sig_handler) que se encarga de … ble ble ble

Luego tuvimos que tener en cuenta que, por la forma que habíamos
implementado el shell, teníamos que hacer la siguiente modificación … bli
bli bli

En nuestro caso, tuvimos que hacer un “truquito” que consiste en … blo blo
blo

Y por último, hay un casito que no funciona bien que es cuando se da la
siguiente condición … blu blu blu.

my-prompt $
```

help

18. Variables y comillas invertidas `` ` ` `` (1pt adicional).

- Se debe permitir la declaración de variables en el *shell* usando el comando `set var <value>`
- Después del comando `set` viene el nombre de la variable.
- Todo lo que viene a continuación del nombre de la variable es **el valor, puede contener espacios**.
- El comando `set` sin parámetros imprime el nombre de cada variable y a continuación el valor.
- Se debe implementar además un comando `get var` que imprime el valor de la variable
- El comando `get` en caso de no recibir parámetros no hace nada.
- Se debe implementar además un comando `unset var` que elimina una variable de la lista de variables del *shell*.
- El valor de las variables debe poder ser accedido en cualquier momento, mientras el *shell* esté ejecutándose.
- El valor de una variable puede ser el resultado de ejecutar un comando con **comillas invertidas** `set var `command1 | command2``.
- Las comillas invertidas funcionan de la siguiente forma: todo lo que se encuentre dentro de las comillas se interpreta como un comando (que puede contener tuberías y todo lo demás) y la salida de ese comando pasa a ver el valor de la variable.
- Todas las funcionalidades implementadas en el *shell* deben funcionar dentro de las comillas invertidas.
- El comando `get` debe poder ponerse (y ejecutarse correctamente) dentro de las comillas.
- **keyword**: **variables**
- **built-in**: **set**
- **built-in**: **get**
- **built-in**: **unset**

```
user@machine $ ./out.out
my-prompt $ set a 5
my-prompt $ set b hola
my-prompt $ set c d e f g
my-prompt $ set
a=5
b=hola
c=d e f g
my-prompt $ ls
file1 file2 dir1 dir2
my-prompt $ set d `ls`
my-prompt $ set
a=5
b=hola
c=d e f g
d=file1 file2 dir1 dir2
my-prompt $ get a
5
my-prompt $ get d
file1 file2 dir1 dir2
my-prompt $ unset d
my-prompt $ unset c
my-prompt $ set
a=5
b=hola
my-prompt $
```

variables y comillas invertidas