KobaShell

Kobayashi82

INDICE

Tabla de contenido

¿QUÉ ES MINISHELL?

Minishell es una versión simplificada del intérprete de comandos Bash en la que se proporciona una interfaz básica de línea de comandos que permite a los usuarios interactuar con el sistema operativo.

Aunque carece de características avanzadas como scripting, Minishell permite ejecutar comandos del sistema, redirigir la entrada y salida, y crear pipelines para dirigir la salida de un comando a la entrada de otro.

Minishell soporta las siguientes características:

- Ejecución de comandos y argumentos
- Variables de Shell y Entorno
- Expansión de variables
- Wildcards (*, ?, [ ], ~)
- Operaciones condicionales (&&, ||)
- Tuberías (|)
- Separación de comandos (&, ;)
- SubShell y preferencias de ejecución (( ), $( ))
- Redirecciones (<, <<, <<<, >, >>)
- Manejo de señales
- Comandos propios, built-ins

REQUISITOS DEL PROYECTO

OBLIGATORIO

- Mostrar un prompt mientras espera un comando nuevo.
- Tener un historial de comandos.
- Buscar y ejecutar el ejecutable correcto.
- ~~No interpretar comillas sin cerrar, caracteres especiales, barra invertida o punto y coma.~~
- Gestionar las comillas simples y dobles
- Implementar redirecciones.
- Implementar pipes.
- Gestionar las variables de entorno y su expansión.
- Gestionar la variable de código de salida ($?).
- Gestionar señales (CTRL + C, CTRL + D y CTRL + \ ).
- Implementar built-ins:
  - echo con la opción -n.
  - cd con una ruta relativa o absoluta.
  - pwd sin opciones.
  - export sin opciones
  - unset sin opciones.
  - env sin opciones o argumentos.
  - exit sin opciones.

BONUS

- Gestionar AND y OR (&&, ||).
- Gestionar prioridades con paréntesis.
- Implementar wildcard de comodín (*).

EXTRA

- Gestionar comillas sin cerrar, caracteres especiales, barra invertida y punto y coma.
- Gestionar las variables de shell y su expansión.
- Implementar wildcards (?, [ ]).
- Implementar built-ins:
  - echo con la opción -e.
  - cd con – y ~.
  - export con la opción -n y -p.
  - env con la opción -s y -0.
  - history
  - about, help y banner sin opciones.

Todos los builtins menos ‘about’, ‘help’ y ‘banner’ tienen las opciones ‘--help’ y ‘--version’.

- Expansión de comandos de subshell ($( )).
- Implementar la tilde ~ para la ruta HOME del usuario.
- Gestionar las siguientes variables:
  - !! Último comando ejecutado
  - $$ PID del proceso.
  - $_ Último argumento ejecutado.
  - $SECONDS Tiempo que lleva ejecutado del shell.
  - $RANDOM Número aleatorio entre 0 y 32767.
  - $SHLVL Nivel de profundidad del shell.
  - $KOBASHELL Ruta absoluta de KobaShell.

ARCHIVOS DE MINISHELL

Minishell está dividida en varias secciones.

La carpeta INC contiene los archivos de cabecera (.h)

La carpeta SRC contiene los archivos fuente del proyecto (.c). Estos están divididos en subcarpetas.
Dentro tenemos las carpetas de la librería LIBFT y de MINISHELL.

MINISHELL

Dentro de MINISHELL tenemos:

- builtin
- clean
- executer
- lexer
- main
- terminal
- variables
- wildcards

MAKEFILE

Otro archivo del proyecto sería 'Makefile' que permite compilar el proyecto con el comando 'make'.

LIBFT

Estos son los archivos de librería LIBFT

Algunas de estas funciones se explicarán más adelante.

INC

Dentro de INC tenemos archivos de cabecera para las carpetas de MINISHELL.

La única diferencia es que el archivo de cabecera de correspondiente a 'main' es 'minishell', y además hay un archivo extra llamado 'colors.h' que incluye macros para los colores usados en el proyecto.

COMPILACIÓN DE MINISHELL

Para compilar Minishell necesitamos ejecutar el comando ‘make’. Pero antes hay que instalar la librería ‘readline’

INSTALAR EN LINUX

Para instalar ‘readline’ en linux, simplemente ejecutar el siguiente comando:

```
sudo apt-get install lib32readline8 lib32readline-dev
```

En el archivo ‘Makefile’ solamente hay que añadir el flag -readline en los objetos y ejecutable.

INSTALAR EN 42 MACS

Para instalar ‘readline’ en los Macs de 42 necesitas instalar previamente ‘brew’.

```
curl -fsSL https://rawgit.com/kube/42homebrew/master/install.sh | zsh
```

Ahora ya podemos instalar readline:

```
brew install readline ; brew link --force readline
echo 'export C_INCLUDE_PATH="$HOME/.brew/include:$C_INCLUDE_PATH"' >> ~/.zshrc
echo 'export LIBRARY_PATH="$HOME/.brew/lib:$LIBRARY_PATH"' >> ~/.zshrc
source ~/.zshrc
```

En el ‘Makefile’ debemos añadir los siguientes flags:

Al compilar el ejecutable, usar el flag: `-L${HOME}/.brew/opt/readline/lib -lreadline`

Al compilar los objetos, usar el flag: `-I${HOME}/.brew/opt/readline/include`

COMPILAR

Ya podemos compilar el Minishell ejecutando ‘make’.

Al finalizar la compilación se creará la carpeta ‘build’ que contendrá los archivos necesarios para compilar (.o, .d, y .a). Aparte, tendremos el ejecutable, el cual debe de ejecutarse con el comando ‘./minishell’.

EJECUCIÓN

Además de poder ejecutar Minishell de la manera tradicional, también podemos usar la opción ‘-c’ que nos permite ejecutar un comando sin necesidad de entrar dentro de Minishell.

Por ejemplo, si ejecutamos (`./minishell -c "echo Haz un word count de esto | wc"`), Minishell mostrará (1 6 26)
1 línea, 6 palabras y 26 caracteres.

PRIMERA EJECUCIÓN

La primera vez que se ejecute Minishell (sin usar la opción -c) se mostrará un mensaje de bienvenida. La siguiente ejecución no se mostrará.

El método usado es bien sencillo, para los más curiosos, lo que hago es crear un archivo oculto llamado ‘.first_start’ al ejecutar ‘make’. Cuando Minishell se abre, si existe el archivo, muestra el mensaje de bienvenida y lo elimina.

PROMPT

Cuando se abre Minishell se nos presentará un prompt y se espera a que el usuario introduzca un comando.

El prompt está formado por [usuario]-MiniShell:ruta/actual$

En caso de que no pueda accederse a la variable USER se intentará usar el valor de la variable LOGNAME y si tampoco se puede, se extrae el nombre de usuario de la variable HOME.
Si ninguna de esas opciones obtiene el nombre de usuario, se mostrará [guest] como nombre de usuario.

Si existe la variable HOME, el prompt convertirá la ruta de HOME por el carácter ~

INPUT

A la hora de introducir comandos en Minishell, puedes dividir el comando en varias líneas.

Si un comando tiene una comilla doble, simple o paréntesis sin cerrar, se mostrará una nueva línea cada vez que se pulse la tecla ENTER, hasta que el comando esté completo.

Además, si el comando termina en AND, OR, PIPE o BARRA INVERTIDA se mostrará una nueva línea para continuar el comando.

Si pulsamos la tecla de cursor arriba, podremos navegar entre los comandos ejecutados con anterioridad.
Cada vez que se ejecuta un comando, este se añade al historial de comandos.

La función ‘readline’ nos permite también autocompletar los nombres de archivos y rutas usando la tecla TAB.

Más adelante veremos más información sobre las comillas y caracteres de escape.

ESTRUCTURA DE DATOS PRINCIPAL

S...

LEXER

El ‘lexer’ es la parte que se encarga de dividir un comando en partes. De esta forma se puede controlar que es un comando, un argumento, una redirección u otro tipo de token como pueden ser los pipes, AND, OR, o el punto y coma.

TOKENIZER

Esta parte se encarga de dividir un comando en ‘tokens’. Lo que quiere decir que separa en partes los comandos, de las redirecciones, pipes, etc…

Por ejemplo:

```
> outfile1  echo patata  > outfile2  frita  |  wc  &&  date
```

Este comando se dividiría de la siguiente forma:

```
REDIRECCIÓN   |   COMANDO   |  REDIRECCIÓN  | PIPE | COMANDO  |  AND  | COMANDO
——————————————————————————————————————————————————————————————————————
 > outfile1   | echo patata frita |  > outfile2  |  |  |  wc  |  &&  |  date
```

Cada token se guarda en una lista de tipo ‘t_token’, la cual es una estructura de datos con la siguientes propiedades:

```
typedef struct s_token
{
    int     type;         ————————————> Tipo de token
    char    *cmd;         ————————————> Comando completo del token
    char    **args;       ————————————> Comando dividido en argumentos en un array
    t_args  *args_lst;    ————————————> Comando dividido en argumentos en una lista
    t_redir *redir;       ————————————> Lista de redirecciones
    t_redir *s_redir;     ————————————> Lista de redirecciones (se aplican a subshell)
    int     fd[2];        ————————————> FDs usados para los PIPES
    int     pid;          ————————————> PID del proceso hijo (si se ejecuta en un hijo)
    bool    heredoc;      ————————————> El token es una redirección de tipo heredoc
    bool    redirection;  ————————————> El token es una redirección sin comando
    bool    wait;         ————————————> Esperar a que terminen los comandos anteriores
    bool    quoted;       ————————————> Indica si hay alguna comilla en el token
    bool    expand;       ————————————> Expandir variables en una redirección de entrada
    bool    no_exec;      ————————————> No ejecutar el token
    t_data  *data;        ————————————> Puntero a la estructura de datos principal
    t_token *prev;        ————————————> Puntero al token anterior
    t_token *next;        ————————————> Puntero al token siguiente
}   t_token;
```

En la tokenización recorremos carácter a carácter el input.

Si el carácter actual no es un token de comando, se agrega el token de ese tipo. En caso contrario se continúa recorriendo la cadena en busca de un token que no sea de comando, y cuando se ha encontrado o terminado el input se crea un token de comando con la cadena recorrida hasta ese punto.

Luego se repite el mismo proceso.

En el caso de que se encuentre un token de redirección se realiza un paso extra.

En primer lugar se agrega el token de redirección con la siguiente palabra como archivo a leer/escribir/delimitador, pero si hubieran más palabras después, hay que tratarlas como un comando o argumento (en caso de que hubiera un comando anterior).

A tener en cuenta, un token escapado, entre comillas o dentro de un paréntesis de una expansión de subshell $( ) no se deberá procesar.

Veamos el ejemplo anterior:

```
> outfile1  echo patata  > outfile2  frita  |  wc  &&  date
```

Si hacemos el proceso de tokenización con este comando, encontramos un token de redirección y por lo tanto creamos un token de redirección de tipo LT (LessThan o <) y le establecemos la propiedad 'token->cmd' en outfile1.

Pero nos encontramos con echo patata que son palabras extras. En este caso recorremos los tokens hacia atrás usando la función prev_cmd_token() y si devuelve NULL, creamos un nuevo token de tipo CMD (comando) estableciendo 'token->cmd' en echo patata.

En el siguiente paso encontramos otra redirección, que hacemos igual, creamos un token de redirección con outfile2, pero ahora está la palabra extra, frita.

En este otro caso prev_cmd_token() nos devuelve un puntero al token de comando anterior que acabamos de crear y contiene echo patata, así que unimos esta palabra extra, que es un argumento en realidad, con su comando, quedando 'token->cmd' en echo patata frita.

Después viene un token de tipo PIPE y se sigue el proceso de tokenización normal.

Hay que destacar que la función prev_cmd_token() sólo buscará un token de comando hasta que se encuentre uno, se encuentre un token diferente a una redirección o se llegue al comienzo del input.

PARSER

El parser se encarga de dividir el contenido de 'token->cmd' en argumentos. La división se realiza separando las palabras que están entre espacios.

Cada argumento se guarda en un nodo de una lista de argumentos 'token->args_lst':

```
typedef struct s_args
{
    char    *arg;        ————————————> Argumento
    t_token *token;      ————————————> Puntero al token al que pertenece el argumento
    t_data  *data;       ————————————> Puntero a la estructura de datos principal
    t_args  *prev;       ————————————> Puntero al argumento anterior
    t_args  *next;       ————————————> Puntero al argumento siguiente
}   t_args;
```

El hecho de tener una lista en lugar de un array facilita el trabajo a la hora de expandir variables y wildcards, ya que las listas permiten sustituir e insertar nodos de manera rápida.

El proceso de parsear el comando en argumentos es relativamente sencillo y se realiza como al tokenizar, recorriendo carácter a carácter. Nada de ft_split.

Se recorre la cadena de texto comprobando si hay un carácter de espacio que no esté escapado o se ha llegado al final de la cadena. Si se encuentra un espacio, se guarda lo recorrido en un nodo nuevo de la lista y se continúa.

Si se encuentran unas comillas o paréntesis, se avanza hasta su cierre, ignorando los espacios que hubieran en su interior.

Destacar que los paréntesis son un tipo de token, pero la expansión de subshell $( ) la trato como texto plano, y por ello la necesidad de determinar el principio y fin de los paréntesis.

SINTAXIS

La última parte de la función del lexer es comprobar que el comando introducido tiene una sintaxis válida.

Por ejemplo, dos pipes juntos o un comando que comienza con una redirección seguida de un paréntesis no son sintaxis válidas.

El proceso de análisis de sintaxis comprueba lo siguiente:

• Se comienza por un token diferente a un comando, redirección o paréntesis.

• Si comienza por redireccionar, el siguiente token no es un paréntesis.

• Si hay una redirección sin archivo o del limitador seguido de otro token.

• Si termina el comando por una redirección sin archivo ni delimitador.

• Si termina el comando por un token que no sea un ; o & (el & lo trato como un igual a un ;).

• Si hay paréntesis sin comandos en su interior o paréntesis huérfanos..

• Si hay comillas sin cerrar.

Cuando se produce un error de sintaxis se muestra el mensaje apropiado y el comando no se ejecuta.

Dentro de las expansiones de subshell pueden producirse errores de sintaxis que impedirán la ejecución del contenido de esa subshell y por lo tanto se expandirá a nada. El mensaje se mostrará, pero no afectará en la ejecución del comando principal.

REDIRECTS

A pesar de que esta parte del código se aplica durante la ejecución de los tokens, realmente forma parte de la tokenización, debido a la forma en la que he administrado las redirecciones.

La función redirects realiza los siguientes pasos:

Determina si el token actual es un comando o una redirección, en caso contrario sale de la función.

Crea una lista de redirecciones de tipo ‘t_redir’, que es un tipo de estructura que contiene la información necesaria para realizar las redirecciones.

Para crear la lista se recorre desde el token actual hasta el siguiente token que no sea CMD ni una redirección.
En cada iteración se crea un nodo en la lista con la información de la redirección.

Luego se obtiene un puntero del comando al que pertenecen las redirecciones.
En caso de no existir un comando para las redirecciones, la primera redirección se convierte en un token de tipo CMD con la propiedad ‘redirection’ en TRUE.
Esto evitará que se ejecute el comando, pero si se aplicarán las redirecciones.

Una vez tenemos el token de tipo CMD, le asignamos la lista de redirecciones que hemos creado en ‘token->redir’

Por último, se eliminan los tokens de redirección, ya que no son necesarios.

Un detalle a tener en cuenta. Cuando se aplican redirecciones a un subshell, no se usa un token de tipo CMD, en este caso se le asigna la lista de redirecciones al token de cierre de paréntesis

VARIABLES

Minishell administra dos tipos de variables. Variables de shell y variables de entorno.

Las variables de shell son variables locales que se heredan a otros subshell, pero no a los comandos ejecutados con 'execve'

Las variables de entorno se heredan también a otros subshell, pero al contrario que las de shell, éstas variables si son pasadas al proceso ejecutado con 'execve' a través del array 'envp'.

VARIABLES DE SHELL

Para establecer una variable de shell hay que usar el formato VAR=VALUE, pudiendo asignar varias variables en la misma línea (var1=value1 var2=value2 …).

También se puede convertir una variable de entorno a una variable de shell con el comando 'export -n VAR' o crearla directamente con 'export -n VAR=VALUE'.

Para eliminar una variable de shell se debe usar el builtin 'unset' con el nombre de la variable (unset var1 var2 …).

VARIABLES DE ENTORNO

Para establecer una variable de entorno hay que usar el builtin 'export' con las asignaciones como argumentos.

Por ejemplo, (export var1=value1 var2=value2 var3 …).

Cómo ha podido observar, no hemos indicado un valor para 'var3'. En este caso la variable 'var3' se guardará para ser exportada, pero no aparecerá al usar el builtin 'env'. Por el contrario habría que usar 'export' sin argumentos o 'export -p', para mostrar todas las variables.

La diferencia entre 'export' y 'export -p' radica en que esta última muestra todas las variables a exportar y 'export' sin argumentos omite algunas de ellas.

A efectos de Minishell, la diferencia real es que 'export' no muestra la variable '_= ' que se asigna automáticamente con el último argumento del comando ejecutado con anterioridad.

Para eliminar una variable de entorno se usa el mismo comando que para eliminar una variable de shell, con 'unset'.

Para más información sobre las variables vea en la siguiente sección los builtins 'export', 'unset' y 'env'.

BUILTINS

Minishell dispone de múltiples builtins. A continuación se detallarán en profundidad su uso y funcionalidad.

Todos los builtins, a excepción de ‘banner’, ‘about’ y ‘help’ tiene las opciones ‘--help’ y ‘--version’

CD (ChangeDir)

Este builtin nos permite cambiar el directorio activo.

Su uso es bastante sencillo, solamente hay que pasar la ruta (absoluta o relativa) como argumento.

Si se ejecuta sin ningún argumento se cambiará la ruta actual a la carpeta HOME del usuario.
En caso de que la variable HOME no exista, se mostrará el mensaje ‘HOME not set’.

Para volver a la ruta anterior hay que usar un guión como argumento ‘cd - ’. Al ejecutarlo se nos mostrará la ruta a la que se intentará acceder en pantalla y se cambiará la ruta actual.
Si la variable OLDPWD no existe, se mostrará el mensaje ‘OLDPWD not set’.

Al usar la tilde ‘ ~ ’ como argumento se cambiará la ruta actual a la carpeta HOME del usuario. Al igual que si lo ejecutaremos sin argumentos. Pero en este caso podemos indicar una ruta también. Por ejemplo: (cd ~/francinette).

Cuando ejecutamos ‘cd’ se comprueba si hay más de un argumento y muestra un mensaje de error y código de salida 1 (solo en versiones de bash de este siglo, no en las de los Macs de 42).

Si no tenemos permisos de lectura, se nos muestra otro mensaje de error y código de salida 1.

Si por fin podemos cambiar de directorio, se actualiza o crea la variable OLDPWD con el valor de la variable PWD.
La variable PWD se actualiza o crea con el valor de la ruta actual, que se obtiene con la función ‘getcwd’.

Cuando se ha cambiado correctamente la ruta activa, el código de salida es 0.

PWD (Print Working Directory)

Este builtin no acepta ningún argumento (a excepción de ‘--help’ y ‘--version’).

Simplemente muestra en pantalla la ruta actual.

Primero se intentará determinar la ruta actual con la función’ cwd’, pero si esta fallara (nos encontramos en un directorio que se ha eliminado, por ejemplo), intentaría obtener la ruta actual desde la variable PWD.

Si aún así no podemos obtener la ruta actual, no mostrará nada. Pero llegados a este caso, el prompt debería cambiar también y mostrar un ‘ . ‘ indicando que no se tiene acceso a la ruta actual.

ECHO

Este builtin imprime en la terminal lo que se le pasa como argumento.

Aunque a simple vista pueda parecer algo sencillo, el builtin ‘echo’ puede ser una potente herramienta, ya que se puede usar junto a expansiones de shell, variables y caracteres especiales como el tabular o salto de línea y redireccionarlo a un archivo o mostrarlo en pantalla.

El builtin ‘echo’ sin argumentos imprime un salto de línea solamente.

Si se añaden argumentos, estos se imprimirán separados por un espacio.

‘echo’ dispone de varias opciones.

Al ejecutarlo con ‘-n ’ se omite el salto de línea que se aplica al final de la cadena a imprimir. Por lo tanto ‘echo -n’ no imprimirá nada.

La opción ‘-e ’ convierte los caracteres especiales que estén entre “comillas dobles”. Por ejemplo:

```
echo -e “\tpatata\nfrita”
      patata
frita
```

‘echo’ devolverá un código de salida de 0 si todo se ha impreso correctamente y un código 1 en caso de que se produzca un fallo al imprimir. En esencia un error en la función ‘write’.

Para imprimir en la terminal se utiliza una función especial llamada ‘print’, de la que se hablará en la sección ‘Funciones Especiales’.

ENV

El builtin ‘env’ es uno de los builtins relacionados con variables. Su función es mostrar en pantalla la lista de variables.

Si se ejecuta sin argumentos ‘env’ se nos mostrará la lista de variables de entorno.

Si usamos la opción ‘-s ’ se mostrará la lista de variables de shell. Esta opción no existe en el builtin ‘env’ de bash. Es único de Minishell, ya que no se ha implementado el builtin ‘set’.

La opción ‘-0 ‘ mostrará la lista de variables sustituyendo el salto de línea por un carácter nulo, por lo tanto, se mostrarán todas las variables en una sola línea.

EXPORT

‘export’ permite crear variables de entorno y de shell.

Al ejecutarse con asignaciones de variables, estas se agregan a las variables de entorno.
Por ejemplo (export var1=value1 var2=value2 …).

Si se ejecuta sin argumentos, muestra la lista de variables de entorno que se exportarán al ejecutar un programa.
Con la opción '-p ' se mostrará la misma lista pero se incluirán todas las variables a exportar (en este caso solo se incluye '_= ').

La opción '-n ' transfiere una variable de entorno existente a la lista de variables de shell.
En caso de que no existiera la variable pasada como argumento, se crearía esta variable en las variables de shell, pero solo si se ha indicado un valor para la variable.

En el caso de que sea una variable que se encuentre solamente en la lista de 'export' y por lo tanto no tenga valor asignado, se eliminará y no se agregará a las variables de shell.

Establece el código de salida 0 a no ser que haya alguna variable que no pueda crearse, que en ese caso el código es 1.

UNSET

Este builtin nos permite eliminar variables, tanto de entorno como de shell.

Su uso es muy sencillo, 'unset' seguida del nombre de las variables a eliminar.
Por ejemplo (unset var1 var2 …).

Establece el código de salida 0 a no ser que haya alguna variable que no pueda crearse, que en ese caso el código es 1.

EXIT

Otro builtin sencillo. Se puede ejecutar sin argumentos y saldremos de Minishell con el último código de salida que tengamos.

Si usamos un argumento, se saldrá con el código indicado en el argumento.

Un detalle a tener en cuenta, el código de salida debe ser menor o igual a 255 y por lo tanto se debe de aplicar el módulo de 256 al código de salida.

HISTORY

Este builtin muestra una lista de todos los comandos que hemos ejecutado con anterioridad cuando no se indica ningún argumento.

El máximo número de comandos que se mostrará está limitado a 500 (más que suficiente).

Pero no es necesario mostrar todos los comandos, podemos indicar el número de comandos que mostrar pasándoselo como argumento, por ejemplo (history 5) nos mostrará los últimos 5 comandos (el comando 'history 5' incluido).

Podemos vaciar el historial de comandos con la opción '-c '.

Tenga en cuenta que no podrá acceder a los comandos anteriores pulsando la flecha arriba una vez haya vaciado el historial. Pero se irán creando nuevos comandos en el historial a medida que los vaya ejecutando.

El código de salida de 'history' siempre es de 0.

BANNER

Simplemente muestra el banner de Minishell y establece el código de salida en 0.

ABOUT

Muestra el mensaje de about de Minishell y establece el código de salida en 0.

HELP

Muestra una información de ayuda y establece el código de salida en 0.

WILDCARDS

S...

TEST DE EVALUACIÓN

BUILTS-IN

•echo
•echo prueba de comando
•echo "-nprueba" de' comando'
•echo \n"prueba\nde"comando
•echo -e \n"prueba\nde" comando
•echo -nnnnneeeen \n"prueba\nde"comando
• cd ~ •cd – •cd . •cd … •cd 1 2 •cd bad • pwd

VARIABLES

REDIRECCIONES

•< file1 echo prueba < file2 de comando
•cat < bad < file1
•cat < file1
•echo prueba > out1 de > out2 comando > out3
•cat < out3 > out3
•echo
•echo
•echo

PIPES

•echo
•echo
•echo
•echo
•echo
•echo
•echo
•echo

WILDCARDS

•echo
•echo
•echo
•echo
•echo
•echo
•echo
•echo

CÓDIGOS DE SALIDAS Y ERRORES

•; ; test
•| test
•echo > <
•echo | |

AND, OR, (;)

PARENTESIS

•echo
•echo
•echo
•echo
•echo
•echo
•echo
•echo

PRUEBAS ESPECIALES

•echo
•echo
•echo
•echo
•echo
•echo
•echo
•echo

GO CRAZY

•echo
•echo
•echo
•echo
•echo
•echo
•echo
•echo

TEST DE EVALUACIÓN

BUILTS-IN

•echo
•echo prueba de comando
•echo "-nprueba" de' comando'
•echo \n"prueba\nde"comando
•echo -e \n"prueba\nde" comando
•echo -nnnnneeeen \n"prueba\nde"comando

VARIABLES

• Gestionar señales (CTRL + C, CTRL + D y CTRL + \ ).
• Implementar built-ins:
◦ echo con la opción -n.
◦ cd con una ruta relativa o absoluta.
◦ pwd sin opciones.
◦ export sin opciones

REDIRECCIONES

• Mostrar un prompt mientras espera un comando nuevo.
• Tener un historial de comandos.
• Buscar y ejecutar el ejecutable correcto.
• No interpretar comillas sin cerrar, caracteres especiales, barra invertida o punto y coma.
• Gestionar las comillas simples y dobles

PIPES

• Gestionar señales (CTRL + C, CTRL + D y CTRL + \ ).
• Implementar built-ins:
◦ echo con la opción -n.
◦ cd con una ruta relativa o absoluta.
◦ pwd sin opciones.
◦ export sin opciones

WILDCARDS

CÓDIGOS DE SALIDAS Y ERRORES

• Mostrar un prompt mientras espera un comando nuevo.
• Tener un historial de comandos.
• Buscar y ejecutar el ejecutable correcto.
• No interpretar comillas sin cerrar, caracteres especiales, barra invertida o punto y coma.
• Gestionar las comillas simples y dobles

• Gestionar señales (CTRL + C, CTRL + D y CTRL + \ ).
• Implementar built-ins:
◦ echo con la opción -n.
◦ cd con una ruta relativa o absoluta.
◦ pwd sin opciones.
◦ export sin opciones

AND, OR, (;)

PARENTESIS

• Mostrar un prompt mientras espera un comando nuevo.
• Tener un historial de comandos.
• Buscar y ejecutar el ejecutable correcto.
• No interpretar comillas sin cerrar, caracteres especiales, barra invertida o punto y coma.
• Gestionar las comillas simples y dobles

• Gestionar señales (CTRL + C, CTRL + D y CTRL + \ ).
• Implementar built-ins:
◦ echo con la opción -n.
◦ cd con una ruta relativa o absoluta.
◦ pwd sin opciones.
◦ export sin opciones

PRUEBAS ESPECIALES

GO CRAZY

• Mostrar un prompt mientras espera un comando nuevo. • Tener un historial de comandos. • Buscar y ejecutar el ejecutable correcto. • No interpretar comillas sin cerrar, caracteres especiales, barra invertida o punto y coma. • Gestionar las comillas simples y dobles	• Gestionar señales (CTRL + C, CTRL + D y CTRL + \ ). • Implementar built-ins: ◦ echo con la opción -n. ◦ cd con una ruta relativa o absoluta. ◦ pwd sin opciones. ◦ export sin opciones