

Minishell (pre)documentación

Un camino hacia la gloria del shell, de cero a héroe

El objetivo de esta (pre)documentación es ayudar a abordar el proyecto Minishell en la Escuela 42 y hacer que parezca un proyecto menos desalentador. Se presentará una visión general del proyecto, se enumerarán los conocimientos y conceptos necesarios antes de comenzar a escribir el código, se hará un inventario de las implementaciones principales solicitadas por el enunciado y se especificarán los principales desafíos que surgirán durante todo el proceso. También incluirá una lista de verificación y una organización del flujo de trabajo para ayudar a los estudiantes a dividir y conquistar Minishell.

La división del flujo de trabajo y de los objetivos se hace siguiendo lo que, según mi entendimiento, es el enfoque habitual: el análisis y la tokenización por un lado, y la ejecución por el otro. Esto no es un requisito, pero es mi recomendación personal que, siempre que sea posible, los estudiantes formen sus equipos de esta manera para que puedan trabajar y desarrollarse con suficiente independencia y no interfieran en el flujo del otro. Luego, cuando llegue el momento, podrán colaborar en el punto de contacto de esta división (básicamente, el punto en el que el analizador envía sus tokens al ejecutor).

En cualquier caso, el objetivo de este proyecto (o al menos uno de ellos) es que ambos miembros del grupo afiancen sus habilidades y consoliden su conocimiento sobre todos los temas de Minishell. Por lo tanto, todo esto tiene más que ver con el enfoque práctico del proyecto (es decir, cómo abordar el primer día y enfrentar la temida pregunta: "¿Cómo se supone que empezamos a construir esto?"). En este último sentido, este documento también incluye un capítulo dedicado a ayudar a navegar ese primer contacto con el proceso de construcción. Mi única esperanza es que, si este documento termina en tus manos o pantallas, logre que Minishell parezca más manejable, accesible e incluso disfrutable.

Este documento es el resultado de una investigación personal sobre el proyecto Minishell, la información incluida en el enunciado y las hojas de evaluación, y la consulta de fuentes directas con objetivos similares o complementarios, como la entrada del proyecto en el Gitbook escrito por Laura Fabbiano y Simon Aeby, o la "Ultimate Github Collaboration Guide" de Johnathan Mines. También utilicé ChatGPT para profundizar en algunas áreas y para ayudarme a crear las listas de verificación, ejemplos y diagramas incluidos en este archivo. Y dado que se debe dar crédito donde corresponde, aquí va un reconocimiento al pequeño demonio gracioso que vive dentro de la vasta y enredada malla de prompts que ha conquistado el mundo con increíble fuerza y presencia inamovible.

¡Buena suerte con el proyecto!
Esto también pasará.

PD: Recomendando encarecidamente que cada equipo de estudiantes haga su propia pre-documentación en lugar de usar la de otra persona, ya que el proceso de escribirla en sí mismo es una habilidad que deberían practicar. Usa este documento como quieras, ya sea como plantilla para el tuyo o como guía lista para usar, solo ten en cuenta este consejo. ;D

Contenidos

1. **Visión General**
 - ¿Qué es Minishell?
2. **Conocimientos Requeridos**
 - Llamadas al sistema
 - Análisis y Tokenización
 - Variables de entorno
 - Señales
 - Comandos internos (Built-Ins)
 - Descriptores de archivo y Redirecciones
 - Depuración y Manejo de Errores
 - Colaboración en Github
3. **Entregables**
 - Entregables principales
 - Requisitos funcionales obligatorios
 - Requisitos funcionales opcionales (Bonus)
 - Requisitos no funcionales
 - Lista básica de verificación para el éxito
4. **Organización del Trabajo Propuesta**
 - Organización expandida del flujo de trabajo
 - Enfoque para el primer día
 - Tabla de desglose de trabajo (con fases)
 - Estructura de directorios propuesta
 - Gestión de ramas en Git propuesta
5. **Lista de Verificación Detallada para Obligatorios**
 - Responsabilidades de Persona 1 (Minitalker)
 - Configuración inicial y planificación
 - Bucle del shell y manejo de señales
 - Creación y gestión de procesos
 - Comandos internos
 - Recolector de basura
 - Responsabilidades de Persona 2 (Pipexer)
 - Análisis y tokenización de entrada
 - Redirección y Pipes
6. **Lista de Verificación Detallada para Bonus**
 - Manejo avanzado de señales
 - Control de trabajos
 - Redirección y piping avanzados
 - Estructuras complejas de comandos
7. **Diagramas de Flujo de Trabajo**
8. **Lista de tareas de documentación**

1. Visión General

Minishell puede parecer un proyecto gigantesco y abrumador la primera vez que te atreves a pensarlo. Por eso, antes de profundizar en los aspectos básicos del tema, comencemos procesando qué se nos pide, cuáles son los principales objetivos y qué conceptos clave están inscritos en el proceso (es decir, ¿qué se aprende al hacer Minishell?).

¿Qué es Minishell?

Minishell desafía al estudiante a crear desde cero una versión simplificada y funcional de un shell Unix. Un shell sirve como interfaz entre el usuario y el sistema operativo, permitiendo a los usuarios ejecutar comandos, gestionar archivos e interactuar con procesos del sistema.

En Minishell, el estudiante desarrollará una versión simplificada de shells populares como bash o zsh. La tarea es replicar comportamientos fundamentales de un shell mientras se adquiere una comprensión más profunda de la programación a nivel de sistema en Unix.

El objetivo principal de Minishell es replicar el comportamiento de una shell Unix básica cumpliendo con requisitos y restricciones específicas. Esto implica crear un programa shell pequeño y liviano que lea la entrada del usuario, interprete comandos y los ejecute, ya sea como funciones integradas o a través de programas externos.

En este proyecto, harás lo siguiente:

- **Implementar un bucle de lectura y ejecución:** Esta es la base de una shell, que solicita repetidamente al usuario una entrada, la procesa y ejecuta comandos hasta que el usuario elija salir.
- **Manejar el análisis de comandos:** Descomponer la entrada del usuario en componentes significativos (tokens) y resolver complejidades como comillas, tuberías y redirecciones.
- **Gestionar procesos:** Usar llamadas al sistema como `fork` y `execve` para ejecutar comandos en procesos secundarios, manteniendo el control en la shell principal.
- **Implementar comandos integrados:** Algunos comandos, como `echo`, `cd` y `env`, serán manejados directamente por Minishell sin depender de binarios externos.
- **Soportar redirecciones y tuberías:** Permitir a los usuarios redirigir la entrada y salida utilizando símbolos como `<`, `>` y `|`.

A diferencia de las shells completas, Minishell se enfoca solo en las funciones principales, permitiendo al estudiante dominar los conceptos esenciales en un entorno de complejidad controlada. El estudiante trabajará con pautas estrictas, como evitar bibliotecas externas, lo que obliga a depender de llamadas al sistema y profundiza la comprensión de la programación a bajo nivel. Todo esto mientras implementa un manejo de errores robusto para asegurar que Minishell se comporte de manera predecible incluso en casos extremos o escenarios de entrada no válida.

En conjunto, este es un proyecto para obtener experiencia práctica con conceptos fundamentales de Unix:

- **Procesos:** Comprender cómo se crean, gestionan y terminan los procesos.

- **Descriptores de archivos:** Aprender cómo se implementa la redirección de entrada/salida a nivel del sistema. (Pipex es un proyecto muy útil para aprender lo básico sobre redirección de I/O).
- **Señales:** Manejar interrupciones en tiempo real como **CTRL-C** y **CTRL-D**. (Minitalk es un proyecto muy útil para aprender lo básico sobre el manejo de señales).
- **Gestión de memoria:** Asegurarse de una correcta asignación de recursos y evitar fugas de memoria en un programa de larga duración.

Al completar este proyecto, el estudiante adquirirá habilidades que forman la base de muchas tareas avanzadas de programación, como crear aplicaciones de servidor, depurar sistemas complejos y escribir código eficiente para entornos con recursos limitados.

2. Conocimientos requeridos

Para construir exitosamente Minishell, se necesitan dominar varios conceptos fundamentales de Unix y programación en C. Esta sección detalla los temas básicos y proporciona breves explicaciones para ayudarte a comenzar.

2.1. Llamadas al sistema

Minishell depende en gran medida de las llamadas al sistema de Unix, que son funciones de bajo nivel que permiten que los programas interactúen con el sistema operativo. Las principales llamadas al sistema que debes dominar incluyen:

- **fork:** Crea un nuevo proceso (hijo) que es una copia del proceso que lo llama (padre). Ambos procesos continúan ejecutándose desde el mismo punto, pero con diferentes IDs de proceso (pid). La bifurcación es esencial para ejecutar comandos en procesos secundarios. La shell principal bifurca un hijo para ejecutar programas externos, dejando la shell libre para continuar funcionando.
- **execve:** Reemplaza la imagen del proceso actual con un nuevo programa. Después de la bifurcación, el proceso hijo usa **execve** para ejecutar el comando. A diferencia de **fork**, no crea un nuevo proceso.
- **wait / waitpid:** Hace que el proceso padre espere a que los procesos hijos terminen. Asegura una sincronización adecuada y evita procesos zombi. El uso de **waitpid** brinda mayor control.
- **pipe:** Crea un par de descriptores de archivo conectados para la comunicación entre procesos. Los datos escritos en un extremo de la tubería se leen desde el otro.

2.2. Análisis y tokenización

Manejar la entrada del usuario es uno de los aspectos más complejos de Minishell. Necesitarás descomponer las cadenas de entrada en componentes significativos, o tokens, respetando la sintaxis de la shell.

- **Tokenización:** Divide la entrada en comandos, argumentos y operadores utilizando delimitadores como espacios, tuberías (**|**) y símbolos de redirección (**<**, **>**).

- **Ejemplo:** `echo "Hello, world!" | grep Hello` se convierte en:
 - Comando: `echo`
 - Argumento: `"Hello, world!"`
 - Tubería: `|`
 - Comando: `grep`
 - Argumento: `Hello`
- **Manejo de comillas:** Administra comillas simples (') y dobles (") para asegurar que los argumentos con espacios o caracteres especiales se traten como una sola unidad.
 - **Ejemplo:** `echo "Hello, world!"` trata `"Hello, world!"` como un solo argumento.
- **Caracteres de escape:** Maneja `\` para escapar caracteres especiales.
 - Esto no se pide en el enunciado, por lo que la decisión de manejar el carácter de escape depende del estudiante. Si decides no aceptarlo, debes definir cómo actuará Minishell si lo encuentra (por ejemplo, rechazando la entrada).
- **Validación de sintaxis:** Verifica errores comunes, como comillas sin cerrar o ubicaciones inválidas de comandos.

2.3. Variables de entorno

Las shells utilizan variables de entorno para almacenar datos de configuración y compartir información entre procesos.

- **Definición:** Son pares clave-valor (por ejemplo, `PATH=/usr/bin:/bin`) que influyen en el comportamiento de los programas.
- **Comandos para manejarlas:**
 - `env`: Imprime todas las variables de entorno.
 - `export`: Agrega o modifica una variable de entorno.
 - `unset`: Elimina una variable de entorno.
- **Uso:**
 - Accede a las variables de entorno utilizando el array `environ` o la función `getenv`.

2.4. Señales

Las señales permiten que los procesos se comuniquen de manera asíncrona, generalmente para interrumpir o finalizar procesos.

- **Señales comunes:**
 - `SIGINT (CTRL-C)`: Interrumpe el proceso actual.
 - `SIGQUIT (CTRL-\)`: Finaliza el proceso actual y genera un volcado de núcleo.
 - `EOF (CTRL-D)`: Señala el final de la entrada.
- **Manejo de señales:**
 - Usa `signal` o `sigaction` para personalizar cómo responde la shell a estas señales.

2.5. Comandos Integrados (Built-Ins)

Minishell requiere que el estudiante implemente varios comandos integrados, los cuales se ejecutan directamente en el shell sin crear un nuevo proceso. Algunos de los comandos integrados requeridos son:

- **echo**: Imprime texto en la terminal.
 - **cd**: Cambia el directorio de trabajo actual.
 - **pwd**: Muestra el directorio de trabajo actual.
 - **export / unset**: Administra variables de entorno.
 - **env**: Muestra todas las variables de entorno.
 - **exit**: Termina el shell.
-

2.6. Descriptores de Archivo y Redirecciones

Los shells utilizan descriptores de archivo (FDs) para gestionar los flujos de entrada y salida.

- **FDs Estándar:**
 - **STDIN (0)**: Entrada estándar.
 - **STDOUT (1)**: Salida estándar.
 - **STDERR (2)**: Error estándar.
 - **Redirecciones:**
 - **<**: Redirige la entrada desde un archivo.
 - **>**: Redirige la salida a un archivo (sobrescribiendo).
 - **>>**: Añade la salida a un archivo.
 - **2>**: Redirige la salida de errores a un archivo (no requerido en el proyecto).
 - **2>>**: Añade la salida de errores a un archivo (no requerido en el proyecto).
 - **Implementación**: Utiliza **dup** y **dup2** para duplicar descriptores de archivo y gestionar redirecciones.
-

2.7. Depuración y Manejo de Errores

- Asegúrate de proporcionar mensajes de error significativos para entradas no válidas (ej. "comando no encontrado").
 - Evita fallos de segmentación validando las entradas y gestionando casos límite.
 - Utiliza herramientas como **valgrind**, **fsanitize** o **gdb** para depurar problemas de memoria.
-

2.8. Colaboración en Github

Aunque no es un requisito explícito, Minishell es una excelente oportunidad para aprender a configurar repositorios colaborativos en Github y gestionar ramas de trabajo para que las fusiones sean seguras, rastreables y se integren correctamente en el árbol principal del proyecto.

Un buen recurso para abordar este tema es la guía de Johnathan Mines en su página de Medium: **The Ultimate Github Collaboration Guide**.

3. Entregables

En esta sección se detallan los entregables esperados para el proyecto Minishell. Tener claridad sobre los entregables ayuda a mantener el enfoque en los objetivos del proyecto.

3.1. Entregables Principales

Estos son los componentes fundamentales del proyecto. Cada uno de ellos es esencial para el éxito final del proyecto y, en conjunto, garantizan que el shell sea funcional, esté bien documentado y estructurado correctamente.

- **Programa Ejecutable:**
 - **Nombre del archivo:** `minishell`.
 - **Propósito:** Un shell funcional que replica un subconjunto de las funcionalidades de Bash.
 - **Ubicación:** El ejecutable debe crearse en la raíz del directorio del proyecto después de ejecutar el `Makefile`.
- **Documentación:**
 - **Archivo ReadMe:**
 - Descripción breve del proyecto.
 - Instrucciones para compilar y ejecutar el shell.
 - Lista de funcionalidades implementadas.
 - Limitaciones conocidas (opcional pero útil).
 - **Comentarios en el Código:**
 - El código debe incluir comentarios explicando secciones críticas, especialmente en lógica compleja de análisis o ejecución.
- **Cumplimiento de Norminette:**
 - Lista básica de verificación:
 - Funciones de ≤ 25 líneas.
 - Indentación y espaciado adecuados.
 - Nombres de variables claros y descriptivos.
 - Máximo de una variable global (exclusivamente para manejo de señales).
- **Makefile:**
 - Debe soportar al menos las siguientes reglas:
 - **all:** Compila el programa.
 - **clean:** Elimina archivos objeto.
 - **fclean:** Elimina archivos objeto y el ejecutable.

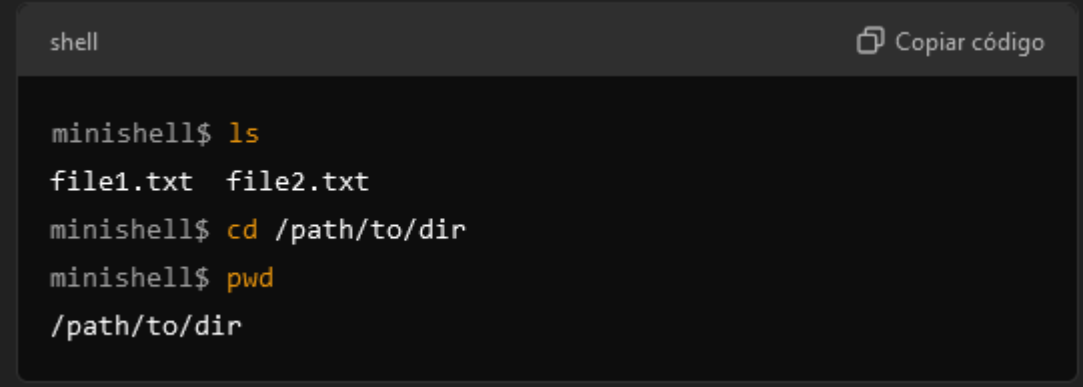
- **re:** Recompila todo desde cero.
 - Debe gestionar eficientemente las dependencias para evitar recompilaciones innecesarias (es una buena oportunidad para aprender a usar archivos `.d`).
-

3.2. Requisitos Funcionales Obligatorios

Esta lista define el comportamiento y las características esperadas para Minishell. Cubre desde comandos integrados básicos hasta funcionalidades complejas como pipes, redirecciones y manejo de señales. Cada requisito se enfoca en un aspecto específico de la funcionalidad del shell y es esencial para asegurar que el shell opere correctamente y cumpla con los objetivos del proyecto.

3.2.1. Prompt del Shell

- **Objetivo:** El shell debe mostrar un prompt al usuario indicando que está listo para recibir comandos. Este prompt puede ser personalizable o predefinido.
- **Comportamiento:**
 - Mientras el shell esté en ejecución, debe mostrar un prompt (por ejemplo, `minishell$`) después de completar un comando o cuando esté inactivo.
 - El prompt debe actualizarse o permanecer estático dependiendo de si el usuario está en un proceso hijo o si se cumplen condiciones específicas (por ejemplo, un cambio en el directorio actual tras un comando `cd`, si se implementa un prompt compuesto).



```
shell Copiar código  
  
minishell$ ls  
file1.txt  file2.txt  
minishell$ cd /path/to/dir  
minishell$ pwd  
/path/to/dir
```

3.2.2 Lectura de Entrada del Usuario

- **Objetivo:** El shell debe ser capaz de leer comandos del usuario en un bucle y ejecutarlos.
- **Comportamiento:**
 - El shell debe aceptar y leer la entrada del usuario en forma de línea de comando. Al presionar Enter, debe procesar y ejecutar el comando.

- Si el usuario introduce un comando no válido o un error de sintaxis, el shell debe mostrar un mensaje de error adecuado.
- La entrada debe ser analizada (parseada) y manejada correctamente, asegurando el tratamiento adecuado de espacios, comillas y caracteres especiales.

```
shell Copiar código  
  
minishell$ echo "Hello, world!"  
Hello, world!  
minishell$ exit
```

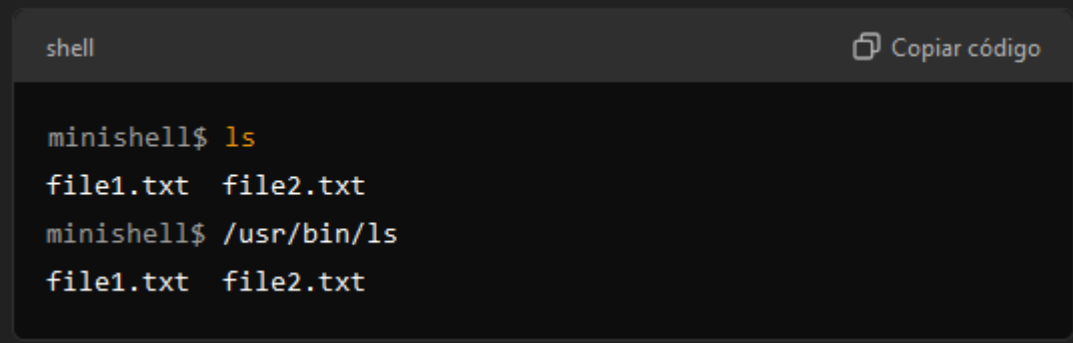
3.2.3 Comandos Integrados (Built-In)

- **Objetivo:** El shell debe soportar un conjunto de comandos integrados para interactuar con el sistema operativo.
- **Comandos a Implementar:**
 - **echo:** Imprime argumentos en la consola.
 - **cd:** Cambia el directorio actual.
 - **pwd:** Muestra el directorio actual.
 - **exit:** Sale del shell.
- **Comportamiento:**
 - Cada comando integrado debe implementarse como una función que pueda llamarse desde el bucle principal del shell.
 - El comando **cd** debe manejar rutas relativas y absolutas correctamente.
 - El comando **exit** debe permitir al usuario salir del shell de manera correcta, especificando un código de salida (o 0 si no se especifica).
 - Se debe implementar manejo de errores. Por ejemplo, si **cd** recibe una ruta no válida, el shell debe mostrar un mensaje de error adecuado.

```
shell Copiar código  
  
minishell$ cd /path/to/directory  
minishell$ pwd  
/path/to/directory  
minishell$ exit
```

3.2.4 Comandos Externos

- **Objetivo:** El shell debe ser capaz de ejecutar comandos externos, aquellos que no son integrados.
- **Comportamiento:**
 - El shell debe buscar los ejecutables en los directorios listados en la variable de entorno `PATH`.
 - Si el comando existe, debe ejecutarse en un proceso separado utilizando las llamadas al sistema `fork()` y `exec()`.
 - Si el comando no existe, el shell debe imprimir un mensaje de error y volver al prompt.



```
shell
minishell$ ls
file1.txt  file2.txt
minishell$ /usr/bin/ls
file1.txt  file2.txt
```

3.2.5 Redirección

- **Objetivo:** El shell debe soportar la redirección de entrada/salida, permitiendo al usuario dirigir la salida de un comando a un archivo o la entrada desde un archivo.
- **Tipos de Redirección:**
 - `>`: Redirige la salida a un archivo (sobrescribe el archivo).
 - `>>`: Añade la salida a un archivo existente.
 - `<`: Redirige la entrada desde un archivo.
 - `<<`: Redirige la entrada desde un "heredoc" (documento aquí).
- **Comportamiento:**
 - El shell debe analizar el comando para identificar los operadores de redirección.
 - Para la redirección de salida (`>`), debe abrir el archivo especificado y escribir en él la salida del comando.
 - Para la redirección de entrada (`<`), debe abrir el archivo especificado y leer de él como si el usuario lo hubiera escrito.
 - Se debe implementar manejo de errores en casos donde el archivo no pueda abrirse, accederse o crearse.

```
shell Copiar código

minishell$ echo "Hello" > output.txt
minishell$ cat < output.txt
Hello
```

3.2.6 Here Documents

- **Objetivo:** El shell debe soportar "here documents", permitiendo al usuario crear una entrada multilínea para un comando directamente en el shell.
- **Comportamiento:**
 - Un "here document" es un tipo especial de redirección que permite proporcionar la entrada a un comando dentro del shell, en lugar de desde un archivo.
 - El usuario puede especificar un delimitador, y todo lo que esté entre el delimitador y la siguiente aparición del mismo será usado como entrada para el comando.
 - Esta funcionalidad es comúnmente utilizada en comandos que requieren entrada multilínea (por ejemplo, `cat`, `grep` o `sed`).

```
arduino Copiar código

minishell$ cat << EOF
This is
a multi-line
text input.
EOF
This is
a multi-line
text input.
```

3.2.7 Pipes

- **Objetivo:** El shell debe soportar pipes, permitiendo que la salida de un comando se utilice como entrada de otro comando.
- **Comportamiento:**

- El shell debe analizar el comando para detectar el operador de pipe (|).
- Debe crear un pipe y bifurcar procesos para ejecutar los comandos en cada lado del pipe.
- La salida del primer comando debe pasar como entrada al segundo comando.
- El shell debe soportar múltiples pipes (por ejemplo, `cmd1 | cmd2 | cmd3 | cmdn`).

```
shell Copiar código

minishell$ ls | grep "file"
file1.txt  file2.txt
```

3.2.8 Manejo de Variables de Entorno

- **Objetivo:** El shell debe soportar variables de entorno, permitiendo obtener, establecer y manipular variables de entorno.
- **Comportamiento:**
 - **env:** Muestra todas las variables de entorno.
 - **setenv y unsetenv:** Funciones para establecer y eliminar variables de entorno, respectivamente.
 - **Expansión de Variables:** El shell debe expandir las variables de entorno en los comandos (por ejemplo, `echo $HOME` debe imprimir el valor de la variable `HOME`).
 - **Exportación de Variables:** Permitir que las variables de entorno se pasen a procesos hijos mediante el comando `export`.

```
shell Copiar código

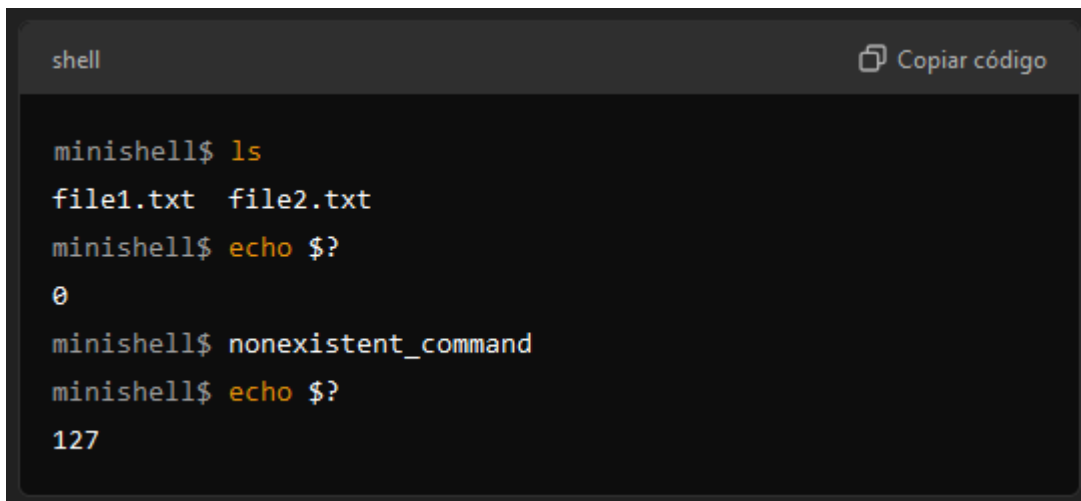
minishell$ echo $HOME
/home/user
minishell$ export VAR=value
minishell$ echo $VAR
value
```

3.2.9 Manejo de Señales

- **Objetivo:** El shell debe manejar adecuadamente las señales, especialmente cuando un proceso se ejecuta en primer o segundo plano.
 - **Comportamiento:**
 - **SIGINT (Ctrl + C):** Detiene el proceso actual en ejecución y muestra un nuevo prompt.
 - **SIGQUIT (Ctrl + \):** Termina el proceso actual.
 - El shell no debe fallar ni congelarse al recibir estas señales; en su lugar, debe recuperarse y presentar el prompt nuevamente.
 - **SIGTSTP (Ctrl + Z):** Debe ser ignorado (esto no significa que no se implemente).
-

3.2.10 Códigos de Estado de Salida

- **Objetivo:** El shell debe rastrear y devolver los códigos de salida de los comandos ejecutados.
- **Comportamiento:**
 - Cada comando debe devolver un código de estado (0 para éxito, diferente de cero para error).
 - El shell debe imprimir el código de salida del último comando cuando se solicite (por ejemplo, utilizando `echo $?`).
 - El comando `exit` debe permitir al usuario terminar el shell y devolver el estado de salida del shell.



```
shell Copiar código

minishell$ ls
file1.txt  file2.txt
minishell$ echo $?
0
minishell$ nonexistent_command
minishell$ echo $?
127
```

3.2.11 Gestión de Memoria

- **Objetivo:** Se debe manejar adecuadamente la asignación y liberación de memoria en todas las operaciones del shell para evitar fugas de memoria.
- **Comportamiento:**
 - Asegúrate de que la memoria se asigne para variables, cadenas y estructuras de datos cuando sea necesario y se libere cuando ya no sea

necesaria (por ejemplo, al salir del programa o después de procesar un comando).

- Maneja los errores liberando cualquier memoria asignada antes de salir debido a una falla.

3.2.13 Autocompletado

- **Objetivo:** El shell debe ofrecer autocompletado básico para comandos, argumentos, nombres de archivo y directorios.
- **Comportamiento:**
 - Implementa autocompletado de línea de comandos escuchando eventos de teclado (generalmente utilizando bibliotecas como `readline`).
 - El shell debe sugerir posibles opciones para comandos, rutas de archivo y argumentos según lo que el usuario haya escrito.
 - Por ejemplo, si el usuario escribe `cd /us` y pulsa Tab, el shell podría sugerir `/usr` o `/user`.



A screenshot of a terminal window with a dark background. The title bar at the top shows 'bash' on the left and a 'Copiar código' button on the right. The terminal content shows a series of commands and their outputs: 'minishell\$ echo "Hello"' followed by 'minishell\$ echo "World"', then 'minishell\$ history' which lists '1 echo "Hello"' and '2 echo "World"'. Finally, 'minishell\$!1' is entered, followed by 'echo "Hello"' on the next line.

```
bash Copiar código

minishell$ echo "Hello"
minishell$ echo "World"
minishell$ history
1  echo "Hello"
2  echo "World"
minishell$ !1
echo "Hello"
```

3.3.14 Personalización del Prompt (NO REQUERIDO)

- **Objetivo:** Permitir al usuario personalizar el prompt con información dinámica, como el directorio actual, el nombre de usuario, el nombre del host o incluso el estado de salida del último comando.
- **Comportamiento:**
 - Permitir al usuario especificar un prompt personalizado utilizando variables de entorno o un archivo de configuración.
 - El prompt podría cambiar dinámicamente según el estado actual, por ejemplo:
 - Mostrando el directorio de trabajo actual.
 - Mostrando el nombre de usuario o el nombre del host.
 - Mostrando el estado de salida del último comando.

```
ruby Copiar código

minishell$ export PS1="\u@\h:\w\$ "
user@hostname:/current/directory$
```

3.3.15 Comillas y Escape

- **Objetivo:** Implementar soporte completo para comillas simples ('), comillas dobles (") y caracteres de escape (por ejemplo, \).
- **Comportamiento:**
 - **Soporte para comillas simples y dobles:**
 - Las comillas simples deben preservar el valor literal de los caracteres dentro de las comillas (sin expansión de variables).
 - Las comillas dobles deben permitir la expansión de variables (por ejemplo, \$HOME).
 - **Manejo de caracteres de escape** para escapar caracteres especiales como espacios, comillas y barras invertidas.

```
bash Copiar código

minishell$ echo "My home is $HOME"
My home is /home/user
minishell$ echo 'My home is $HOME'
My home is $HOME
```

3. 3. Requisitos Funcionales Opcionales

Los requisitos opcionales definen características adicionales que mejoran la funcionalidad de Minishell. Estas incluyen operadores lógicos, paréntesis para priorizar comandos y expansión de comodines. Aunque no son obligatorios, implementarlos demuestra un entendimiento avanzado y habilidades superiores.

3.3.1 Operadores Lógicos: && y ||

- **Objetivo:** La shell debe soportar los operadores lógicos && (AND) y || (OR) para la ejecución condicional de comandos.
- **Comportamiento:**

- **&&**: El segundo comando se ejecuta solo si el primer comando tiene éxito (es decir, devuelve un estado de salida de 0).
 - **||**: El segundo comando se ejecuta solo si el primer comando falla (es decir, devuelve un estado de salida distinto de cero).
-

3.3.2 Paréntesis para Priorización

- **Objetivo:** La shell debe soportar paréntesis para agrupar comandos y definir la prioridad de ejecución.
 - **Comportamiento:**
 - Los comandos entre paréntesis se tratan como una unidad y se ejecutan con prioridad.
 - **Ejemplo:** `(comando1 && comando2) || comando3` → `comando1 && comando2` se evalúan primero. Si ambos tienen éxito, `comando3` se omite. De lo contrario, se ejecuta `comando3`.
-

3.3.3 Expansión de Comodines

- **Objetivo:** La shell debe soportar la expansión de comodines usando `*` para coincidir con archivos y directorios en el directorio de trabajo actual.
- **Comportamiento:**
 - **Coincidencia con Comodines:** El comodín `*` coincide con cero o más caracteres en nombres de archivos y directorios.
 - **Integración con Comandos:** Los comodines deben funcionar sin problemas con otros comandos (por ejemplo: `ls *.txt`).

4. Organización Propuesta del Trabajo

La organización propuesta divide el proyecto de Minishell en dos áreas principales, con tareas asignadas a cada miembro del equipo basándose en las dos ramas principales: **análisis sintáctico (parsing)** y **ejecución (execution)**. Esta estructura permite que cada miembro aproveche sus fortalezas y trabaje eficientemente en componentes complementarios, colaborando en tareas compartidas como el manejo de errores y la depuración, al tiempo que se minimizan los puntos de contacto y sincronización.

Distribución de Tareas:

Persona 1: Análisis Sintáctico (Parsing)

- **Tokenización** de la entrada del usuario en componentes manejables.

- **Análisis de comandos, argumentos y caracteres especiales** (por ejemplo, `|`, `>`, `<`).
- **Manejo adecuado de comillas y secuencias de escape.**
- **Agrupación de comandos y paréntesis** (bonus: soporte para `&&` y `||`).

Persona 2: Ejecución (Execution)

- **Gestión de la creación y ejecución de procesos** tanto para comandos internos como externos.
- **Implementación de redirecciones** (por ejemplo, `<`, `>`, `>>`) y **pipes**.
- **Manejo de variables de entorno** y su expansión.
- **Orden de ejecución adecuado y gestión de estados.**

Ambos deben colaborar en:

- **Manejo de errores y depuración** para asegurar una integración fluida entre el análisis sintáctico y la ejecución.
- **Manejo de casos límite**, como **sintaxis no válida** u **operaciones no soportadas**.

Organización Flexible de Ramas en GitHub

- La división de tareas y la gestión de ramas en GitHub no tienen que seguir exactamente la misma división de tareas.
 - Por ejemplo, la sección de **Parsing** puede dividirse en ramas separadas para **tokenización**, **validación de sintaxis** y **agrupación de comandos**.
 - La sección de **Execution** puede dividirse en ramas para **gestión de procesos**, **redirección** y **manejo de variables de entorno**.
- **Esta flexibilidad permite adaptar el flujo de trabajo a las preferencias del equipo y asegurar un desarrollo eficiente.**

4.3. Estructura Propuesta de Directorios

Para **agilizar el desarrollo y mantener claridad**, el directorio `src/` se divide en **subdirectorios** basados en las áreas principales del proyecto de Minishell. Esta organización refleja los componentes fundamentales de una shell funcional, como el **análisis sintáctico**, **manejo de señales**, **ejecución** y **gestión de variables de entorno**.

Propuesta de Estructura:

```
Minishell/
├─ includes/           # Header files
├─ libs/              # External libraries (if any)
├─ tests/             # Test files
├─ Makefile           # Makefile for building the project
├─ src/               # Source code
│   ├─ builtins/      # Built-in command implementations
│   │   ├─ cd.c
│   │   ├─ echo.c
│   │   ├─ env.c
│   │   └─ exit.c
│   ├─ executor/      # Command execution (piping, redirection, execve)
│   │   ├─ executor.c
│   │   ├─ piping.c
│   │   └─ redirection.c
│   ├─ parser/        # Input tokenization and syntax checking
│   │   ├─ parser.c
│   │   ├─ tokenizer.c
│   │   └─ syntax_checker.c
│   ├─ signals/       # Signal handling
│   │   └─ signals.c
│   ├─ env/           # Environment variable management
│   │   ├─ env.c
│   │   └─ env_utils.c
│   ├─ utils/         # Utility functions
│   │   ├─ string_utils.c
│   │   └─ error_handler.c
│   └─ main/          # Shell loop and main entry point
│       ├─ main.c
│       └─ shell_loop.c
```

builtins/:

- Implementaciones de comandos internos.
- Archivos de ejemplo: `cd.c`, `echo.c`, `env.c`, `exit.c`.

executor/:

- Funciones para ejecutar comandos.
- Maneja pipes (`|`), redirección (`<`, `>`) y comandos externos (`execve`).
- Archivos de ejemplo: `executor.c`, `piping.c`, `redirection.c`.

parser/:

- Maneja la tokenización de la entrada y la validación de la sintaxis.
- Divide la entrada del usuario en comandos, argumentos y operadores.
- Archivos de ejemplo: `parser.c`, `tokenizer.c`, `syntax_checker.c`.

signals/:

- Maneja el comportamiento de las señales para la shell.
- Maneja señales como `SIGINT` y `SIGQUIT`.
- Archivo de ejemplo: `signals.c`.

env/:

- Funciones para gestionar variables de entorno (`getenv`, `setenv`, `unsetenv`).
- Archivos de ejemplo: `env.c`, `env_utils.c`.

utils/:

- Funciones comunes y utilitarias compartidas entre módulos.
- Archivos de ejemplo: `string_utils.c`, `error_handler.c`.

main/:

- Bucle principal de la shell, visualización del prompt y punto de entrada del proyecto.
- Archivos de ejemplo: `main.c`, `shell_loop.c`.

4.5. Gestión Propuesta de Ramas en Git

Para **mantener un árbol de trabajo limpio y escalable** conectando los repositorios locales y GitHub, se recomienda organizar las ramas en **3 niveles** de acuerdo con las mejores prácticas de **Git Flow**.

Tipos de Ramas:

1. Feature Branches

- **Propósito:** Desarrollar características o funcionalidades específicas.

- **Convención de nombres:** Usa nombres claros y descriptivos (por ejemplo, `feature-shell-loop` o `feature-signal-handling`).
 - **Flujo de trabajo:**
 - Crea una rama **feature** a partir de **develop** (o **main** si no existe **develop**).
 - Trabaja en la funcionalidad, **comprometiendo cambios de forma incremental**.
 - Cuando la funcionalidad esté completa y probada, **haz un Pull Request** para **mergearla en develop**.
 - **Beneficios:**
 - Mantiene **main** limpio y permite el **desarrollo paralelo de múltiples funcionalidades**.
-

2. Develop Branch

- **Propósito:** Rama central para integrar todas las ramas **feature**.
 - **Rol en el flujo de trabajo:**
 - Las ramas **feature** se **mergullan en develop** a medida que se completan.
 - Refleja el estado más reciente del proyecto con todas las características en curso **combinadas pero no lanzadas**.
 - **Bug fixes** o **mejoras** de ramas de prueba también se pueden **mergear en develop**.
 - Una vez que **develop** sea **estable**, se puede **mergear en main** para el lanzamiento.
 - **Beneficios:**
 - **Área de preparación** para el desarrollo en curso, manteniendo **main** estable.
-

3. Test Branches

- **Propósito:** Para **experimentar, depurar o probar** partes específicas del código sin interrumpir otras ramas.
 - **Flujo de trabajo:**
 - Crea una rama de prueba a partir de **develop** o una **feature** (por ejemplo, `test-shell-loop`).
 - Realiza pruebas, depuración o cambios experimentales.
 - Decide si **mergear los cambios de nuevo** a la rama padre o **descartar la rama de prueba** después de la prueba.
 - **Beneficios:**
 - Permite la **experimentación segura** sin afectar la producción ni el desarrollo en curso.
-

Resumen del Flujo de Trabajo Propuesto:

1. **Comienza con una rama main limpia.**
2. **Crea una rama develop a partir de main.**
3. **Desarrolla funcionalidades en ramas feature-, mergeando en develop.**
4. **Utiliza ramas test-** según sea necesario para **depuración** o **experimentos**.
5. **Mergea develop en main** regularmente una vez que **todo esté estable** y listo para el **lanzamiento**.

5. Lista de Verificación Obligatoria en Profundidad

Esta lista de verificación proporciona una guía paso a paso para construir el proyecto Minishell. Es importante abordar primero los requisitos principales (como la creación de procesos, comandos internos y redirección) antes de pasar a características más avanzadas como el control de trabajos y el manejo de señales. Asegúrate de que la funcionalidad principal esté sólida antes de avanzar a los requisitos adicionales. Esta lista de verificación debe servir como una herramienta organizativa útil para que los equipos sigan el progreso a lo largo del proyecto. Desglosa las tareas principales en pasos manejables, lo que hace que el proyecto sea menos abrumador y más estructurado.

5.1 Responsabilidades de la Persona 1

5.1. Configuración Inicial y Planificación

5.1.1. Configura el repositorio del proyecto y .gitignore:

- Inicializa un nuevo repositorio Git.
- Crea los directorios `srcs/`, `includes/`, y `libs/` en el proyecto.
- Agrega los archivos apropiados `README.md` y `LICENSE` (si es necesario).
- Configura `.gitignore` para ignorar archivos comunes de compilación (por ejemplo, `*.o`, `*.d`, `*.a`, `*.out`).
- Excluye archivos específicos de IDE (por ejemplo, `.vscode`, `.idea`, etc.) y archivos del sistema (`Thumbs.db`, `.DS_Store`, etc.).

5.1.2. Planifica la estructura de directorios:

- Organiza el proyecto con la siguiente estructura:
 - `srcs/`: Para archivos de código fuente.
 - `includes/`: Para archivos de encabezado.
 - `libs/`: Para cualquier librería externa.
- Asegúrate de que los encabezados estén en `includes/` y los archivos fuente en `srcs/`.

5.2. Bucle de Shell y Manejo de Señales

5.2.1. Bucle Infinito de Shell:

- **Visualización del Prompt:**

- Crea una función para mostrar el prompt de la shell (por ejemplo, `$` o un prompt personalizado).
- Muestra el nombre de usuario y el directorio de trabajo actual (CWD).
- **Manejo de la Entrada del Usuario:**
 - Implementa `getline()` o `get_next_line()` para capturar la entrada del usuario.
 - Elimina los espacios en blanco al inicio y al final.
- **Lógica del Bucle:**
 - Configura el bucle para leer y procesar comandos continuamente hasta salir (usando `exit` o `Ctrl+D`).
 - Agrega lógica para manejar entradas vacías (por ejemplo, no hacer nada si el usuario presiona Enter sin escribir nada).

5.2.2. Manejo de Señales:

- **SIGINT (Ctrl+C):**
 - Crea un manejador de señales personalizado para interceptar `SIGINT` y evitar que la shell termine inmediatamente.
 - Imprime un nuevo prompt cuando se presiona Ctrl+C, sin salir de la shell.
- **SIGTSTP (Ctrl+Z):**
 - Implementa un manejador personalizado para `SIGTSTP` para capturar Ctrl+Z y suspender los procesos en segundo plano, mostrando un mensaje que indique la suspensión.
 - Asegúrate de que la shell siga siendo responsable después de Ctrl+Z.

5.2.3. Copia de Variables de Entorno:

- Antes de ejecutar el bucle infinito, almacena una copia de las variables de entorno en una lista o array.
- Si no hay entorno, usa un nombre de usuario predeterminado (por ejemplo, `root@localhost`) tomado de `/etc/passwd`.

5.2.4. Casos de Ejecución Extremos:

- Minishell debe ejecutarse desde un directorio inexistente:
 - Con `env`, debe mostrar el mismo CWD desde donde fue lanzado y poder usar `cd` para retroceder.
 - Sin `env`, no debe fallar y debe tener una ruta predeterminada de respaldo (`/root`).
 - Muestra un CWD específico cuando no hay `env` y el directorio no existe.

5.3. Comandos Internos

5.3.1. cd (Cambiar Directorio):

- Implementa el comando `cd` para cambiar el directorio de trabajo actual usando `chdir()`.

- Maneja casos para rutas absolutas, relativas, `cd -`, `cd /`, y sin argumento (debe ir al directorio de inicio).
- Normaliza las rutas (por ejemplo, maneja `../`, `./` y caracteres especiales como `~`).
- Asegúrate de que no haya fallos al ejecutar sin entorno.

5.3.2. `echo` (Imprimir en Salida Estándar):

- Analiza los argumentos y los imprime en la salida estándar, separados por espacios.
- Maneja opciones especiales, como `-n` para evitar un salto de línea al final.

5.3.3. `exit` (Salir de la Shell):

- Implementa el comando interno `exit` para terminar la shell.
- Permite un código de estado opcional de salida.
- Asegúrate de que se realicen los cierres de archivos abiertos y la liberación de memoria adecuada.

5.3.4. `env` (Mostrar Variables de Entorno):

- Implementa el comando `env` para imprimir todas las variables de entorno en formato `KEY=VALUE`.

5.3.5. `export` (Establecer o Exportar Variables de Entorno):

- Implementa `export` para crear o modificar variables de entorno.
- Si no se proporcionan argumentos, imprime la lista de variables exportadas.
- Maneja los casos inválidos de manera adecuada.

5.3.6. `unset` (Eliminar Variables de Entorno):

- Implementa `unset` para eliminar variables de entorno.
- Asegúrate de manejar adecuadamente los casos como intentar eliminar variables no existentes.

5.4. Recolector de Basura

5.4.1. Crear una estructura para el recolector de basura:

- Hazla capaz de almacenar casos con punteros simples y dobles.

5.4.2. Crear un constructor para el recolector de basura:

- Permite construir nodos que apunten a punteros simples y dobles.

5.4.3. Crear un protocolo de limpieza del recolector de basura:

- Recorre la lista y libera los punteros simples y dobles adecuadamente.
-

5.2 Responsabilidades de la Persona 2

5.2.1. Análisis y Tokenización de la Entrada

5.2.1.1. Escribir una función de tokenización para dividir la entrada por espacios y manejar caracteres especiales (comillas, pipes, redirección):

- Implementa un tokenizador que divida la cadena de entrada del usuario en tokens según espacios y caracteres especiales.
- Maneja las comillas (simples y dobles) como un solo token y las secuencias de escape.
- Tokeniza y separa pipes (|) y operadores de redirección (<, >, >>).
- Maneja casos extremos, como tokens vacíos o comillas no cerradas, e informa de errores cuando corresponda.

5.2.1.2. Detectar y manejar la ejecución en segundo plano (&):

- Analiza la cadena de entrada para detectar el símbolo & al final de un comando.
- Establece un indicador para la ejecución de procesos en segundo plano.

5.2.2. Redirección y Pipes

5.2.2.1. Implementar redirección de entrada (<), redirección de salida (>, >>), y redirección heredada (<<):

- Maneja la apertura de archivos para entrada/salida usando `open()` y redirige las secuencias con `dup2()`.
- Gestiona archivos temporales para entrada heredada hasta que se encuentre el delimitador establecido.

5.2.2.2. Probar y manejar redirecciones combinadas:

- Implementa la lógica para redirecciones combinadas (por ejemplo, `2>` para `STDERR`).
- Asegúrate de que se manejen correctamente las redirecciones de `STDOUT` y `STDERR`.

5.2.2.3. Diseñar y probar la funcionalidad de pipes (|):

- Detecta el símbolo de pipe (|) en la entrada tokenizada y separa los comandos.
- Implementa la comunicación por pipes usando `pipe()`, `fork()`, `dup2()`, y `execvp()`.
- Maneja múltiples pipes y errores como comandos vacíos.

5.2.2.4. Probar la integración de pipes y redirección:

- Prueba escenarios combinando pipes y redirecciones (por ejemplo, `cmd1 < file1 | cmd2 > file2`).

- Asegúrate de que la integración de pipes y redirección sea fluida.

6. Bonus In-Depth Checklist

6.1 Expansión de comodines

1. Implementar análisis y coincidencia de comodines:
 - Desarrollar la lógica para detectar patrones de comodines (por ejemplo, `*`, `?` y `[]`) en la entrada del usuario.
 - Integrar la expansión de comodines con las funciones `opendir()` y `readdir()` para realizar listas de directorios.
 - Gestionar casos especiales como patrones no válidos o archivos sin coincidencias.
2. Ordenar y formatear los resultados:
 - Implementar la lógica para ordenar los archivos coincidentes alfabéticamente (si es necesario).
 - Asegurarse de que los comodines expandidos se integren correctamente en el comando.
3. Probar la expansión de comodines en diferentes contextos:
 - Probar con comandos como `ls *.c`, `echo file[1-3]`, y `rm test*`.
 - Asegurarse de que la expansión de comodines no modifique los argumentos originales cuando no se encuentre ninguna coincidencia (por ejemplo, `echo "no match *"`).

6.2 Operadores lógicos (&& y ||)

1. Parseo de operadores lógicos:
 - Ampliar el tokenizador para detectar `&&` y `||` como tokens distintos.
 - Asegurarse de que los operadores lógicos se manejen correctamente, tanto con espacio como sin espacio alrededor (por ejemplo, `command1 && command2` y `command1&&command2`).
2. Implementar el flujo de ejecución lógica:
 - Añadir lógica para ejecutar comandos de manera condicional según el estado de retorno del comando anterior:
 - `&&`: Ejecutar el siguiente comando solo si el anterior tiene éxito (código de salida 0).
 - `||`: Ejecutar el siguiente comando solo si el anterior falla (código de salida distinto de 0).
 - Gestionar la cadena de múltiples operadores lógicos en una línea de comando (por ejemplo, `command1 && command2 || command3`).
3. Manejo de paréntesis para agrupar comandos:
 - Parsear y manejar paréntesis para agrupar comandos y definir la precedencia de ejecución (por ejemplo, `(command1 && command2) || command3`).
 - Implementar un enfoque recursivo o basado en pilas para evaluar grupos anidados.

4. Probar varias combinaciones de operadores lógicos:

- Probar casos sencillos: `command1 && command2, command1 || command2`.
- Probar casos mixtos: `command1 && command2 || command3`.
- Probar casos agrupados: `(command1 && command2) || (command3 && command4)`.
- Probar casos extremos: operadores lógicos con comandos inválidos, paréntesis vacíos o errores de sintaxis.

6.3 Paréntesis para agrupamiento de comandos

1. Actualizar el analizador para detectar paréntesis:

- Ampliar el tokenizador para tratar los caracteres (y) como tokens especiales.
- Asegurarse de que los paréntesis desbalanceados o mal colocados generen un error adecuado.

2. Implementar ejecución agrupada de comandos:

- Desarrollar la lógica para ejecutar comandos dentro de paréntesis como una unidad única.
- Redirigir los comandos agrupados para ser ejecutados en un subshell si es necesario.

3. Manejo de paréntesis anidados:

- Soportar múltiples niveles de anidamiento (por ejemplo, `((command1 && command2) || command3))`).
- Usar una pila o enfoque recursivo para mantener el orden de evaluación correcto.

4. Probar funcionalidad de paréntesis:

- Probar casos simples: `(command1)`.
- Probar casos anidados: `((command1 && command2) || (command3 && command4))`.
- Probar casos extremos: paréntesis vacíos, agrupamiento inválido o paréntesis desbalanceados.

6.4 Pruebas de integración adicionales

● Combinar todas las funciones bonus en líneas de comando complejas:

- Ejemplos: `ls *.c && (echo "Success" || echo "Failure")`.
- Manejar comodines, operadores lógicos y paréntesis en la misma línea de comando.

● Asegurar una interacción fluida entre las funcionalidades principales y los extras:

- Verificar que las nuevas características no interfieran con las funcionalidades ya implementadas y que trabajen de manera conjunta.