Minishell.c

INDICE

La función main recibe tres parámetros:

- **argc:** el número de argumentos que se pasan al programa.
- **argv:** una lista de los argumentos en forma de cadenas de texto.
- **envp**: una lista de las variables de entorno. (*ej. "PATH=/usr/bin" , NOMBRE=VALOR*)

-Verificamos que el número de argumentos sea 1. Ya que solo ejecutaremos **/.minishell**

-Ignoramos tanto argc como argv.

-Reservamos memoria para nuestra estructura y la inicializamos con **init_struct**.

-Llamamos a get_envp(data, envp, -1)

-La función toma las variables de entorno (envp) y las convierte en una lista enlazada de estructuras t_envp, almacenándolas en data->envp.

`char **curr`: Un array temporal que almacena el nombre y el valor de una variable de entorno separados.

`t_envp *tmp`: Un puntero temporal para crear un nodo de la lista enlazada que representará cada variable de entorno.

Divide la cadena en NOMBRE y VALOR:

curr = ft_split(envp[i], '=');

- envp[i] es una cadena del tipo NOMBRE=VALOR.
- ft_split divide esta cadena por el carácter = y retorna un array:
    - curr[0]: Contiene el nombre de la variable (NOMBRE).
    - curr[1]: Contiene el valor de la variable (VALOR), si existe

tmp->name = ft_strdup(curr[0]);

- Se duplica el contenido de curr[0] en tmp->name, asignando el nombre de la variable.

Si no hay valor en curr[1], se asigna "=" como contenido (para representar una variable sin valor).

Si existe un valor, se concatena un = al inicio del valor usando ft_strjoin

-tmp->ind: Probablemente un índice o bandera, inicializado en 0.

-tmp->next: Se establece como NULL porque este nodo es el último de la lista por ahora.

lst_addenv_back(&data->envp, tmp);

- Se llama a una función auxiliar, lst_addenv_back, para agregar el nodo al final de la lista data->envp.

free_split(curr); curr = NULL; Libera la memoria usada por curr, ya que ya se copiaron sus valores.

set_envp_index(data);

- Llama a set_envp_index(data) que ajusta los índices en la lista para ayudar con la gestión de las variables de entorno

La función set_envp_index

Asigna un índice a cada nodo en data->env, ordenándolos según el orden alfabético de sus nombres. Esto permite que la lista enlazada se pueda recorrer
de forma ordenada según estos índices, facilitando su uso en operaciones cómo imprimir las variables de entorno en orden alfabético**.**

- **t_data *data:** Contiene la lista enlazada envp, que se debe organizar asignando índices.
- **int i:** Contador que empieza en 1 y se usa como índice asignado a cada nodo.
- **t_envp *first:** Nodo de referencia que contiene temporalmente el menor nombre de variable aún no indexado.
- **t_envp *tmp:** Nodo usado para recorrer la lista y comparar nombre

Recorre toda la lista enlazada y pone ind = 0 en cada nodo. Esto asegura que todos los nodos estén inicialmente sin índice.

Se ejecuta un número de iteraciones igual al tamaño de la lista enlazada (size_envp(data->envp) devuelve el número total de nodos).

Recorre la lista enlazada buscando el nodo con el menor name (ordenado alfabéticamente), que aún no tenga un índice (ind == 0).

- exp_cmp(first->name, tmp->name): Compara alfabéticamente los nombres.
  - Si el resultado es positivo (> 0), significa que tmp->name es menor que first->name.
- Actualiza first si encuentra un nodo menor.
- Asigna el índice actual (`i`) al nodo que contiene el menor nombre de variable.

Llama a la función `get_first(data)`

- Encuentra el nodo con el nombre alfabéticamente mayor (o "último") que no ha sido indexado todavía (ind == 0)

size_envp(data->envp)

- Devuelve el número total de nodos en la lista envp.

exp_cmp(first->name, tmp->name)

- Se usa para determinar el orden entre los nombres de variables de entorno (o envp) y saber cuál viene primero lexicográficamente.

get_first(data)

- Devuelve un nodo para reiniciar la búsqueda de nombres no indexados.
- Podría ser simplemente el primer nodo en la lista.
- Encuentra el nodo con el nombre alfabéticamente mayor (o "último") que no ha sido indexado todavía (ind == 0).

LLamamos a la función minishell

Esta función es el núcleo de la ejecución del programa de tu minishell. Coordina las señales, maneja la entrada del usuario y ejecuta los comandos en un bucle infinito, liberando recursos conforme se usan.
int minishell(t_data *data, char **env)

- **t_data *data**: Estructura que contiene información necesaria para la ejecución, como nodos de comandos, entradas del usuario, y variables de entorno.
- **char **env**: La lista de variables de entorno del sistema, pasada al minishell.

El bucle principal mantiene al shell en funcionamiento hasta que el usuario lo detenga explícitamente.

SIGINT: Señal que se envía cuando el usuario presiona Ctrl+C.

- Se maneja con la función signals, probablemente para personalizar cómo el shell responde (por ejemplo, no cerrarlo).

SIGQUIT: Señal que se envía con Ctrl+\.

- Se ignora (`SIG_IGN`) para evitar interrupciones inesperadas.

signal(SIGINT, signals):

- Asocia la señal SIGINT (enviada cuando el usuario presiona CTRL+C) a la función signals.
- Esto permite que, en lugar de interrumpir y cerrar minishell, se ejecute la lógica personalizada definida en signals.

signal(SIGQUIT, SIG_IGN):

- Ignora la señal SIGQUIT (enviada al presionar CTRL+\), evitando que minishell se cierre y mostrando un comportamiento más robusto.

sig_ctrlslash
Este manejador imprime "Quit: 3" cuando el usuario presiona **CTRL+**, emulando el comportamiento de shells como bash y zsh.
rl_on_new_line():

Indica a readline que comience una nueva línea.

rl_replace_line("", 0):

Limpia cualquier texto que el usuario haya escrito en la línea actual.

sig_ctrlc
Este manejador personaliza el comportamiento de CTRL+C:

printf("\033[K\n");: Limpia la línea actual en la terminal (\033[K es un código de escape ANSI) y baja a la siguiente línea.

rl_on_new_line() y rl_replace_line("", 0):

Restablecen el estado de readline, limpiando la línea de entrada y preparando una nueva.

SIGNALS

Este es otro manejador para CTRL+C que combina:

- rl_redisplay(): Redibuja el prompt de readline en la nueva línea.
- Limpieza de pantalla: Similar a sig_ctrlc, limpia la línea actual y baja a la siguiente.

int minishell(t_data *data, char **env) llama a:

GETPROMPT

Es responsable de construir y mostrar el prompt en tu minishell, y luego obtener la entrada del usuario. Vamos a desglosarla paso a paso.

- **direc[500]**: Un arreglo de caracteres que almacenará la ruta del directorio de trabajo actual (con un tamaño de 500 caracteres).
- **direcf**: Un puntero a cadena que se usará para almacenar la concatenación de la ruta actual con el prompt

getcwd(direc, sizeof(direc)): La función getcwd obtiene la ruta del directorio actual de trabajo y la almacena en direc. Si falla (es decir, si la ruta no puede obtenerse), retorna NULL y la función termina con EXIT_FAILURE.

ft_strjoin(direc, " % "): Utiliza ft_strjoin (para concatenar cadenas) para unir la ruta del directorio (direc) con el string " % " (que será el separador de tu prompt).

Si ft_strjoin falla y retorna NULL, se termina la función con EXIT_FAILURE.

readline(direcf): La función readline se utiliza para mostrar el prompt (con el contenido de direcf, que es la ruta seguida de "% ") y esperar la entrada del usuario.

- El valor ingresado por el usuario se guarda en **data->prompt**.

free(direcf): Libera la memoria de direcf que fue asignada por ft_strjoin para evitar fugas de memoria.

Luego, establece direcf a NULL para evitar el uso accidental de un puntero no válido.

Si **data->prompt** es NULL (lo que significa que readline falló o el usuario cerró la entrada), imprime "exit" y termina el programa con EXIT_FAILURE

Si todo va bien, la función retorna EXIT_SUCCESS, indicando que el prompt se mostró correctamente y la entrada fue recibida.

int minishell(t_data *data, char **env) comprueba:

Comprobación de Entrada del Usuario:

ft_strncmp(data->prompt, "/0", 1) == EXIT_FAILURE

Verifica si data->prompt no es una cadena vacía ("/0" en tu código parece un error tipográfico y debería ser "\0" para comparar con una cadena vacía).

Si el comando no está vacío, lo añade al historial con add_history.

Esto permite que el usuario use la funcionalidad de historial de readline (navegar con flechas arriba/abajo).

int minishell(t_data *data, char **env) llama a:

INPUTS

Esta función principal coordina todas las etapas necesarias para procesar la entrada del usuario. Cada paso verifica errores y retorna EXIT_SUCCESS si se detecta algún problema.

Primero comprueba el lexer:

LEXER:
Verifica que las comillas estén correctamente balanceadas.
Lógica del Bucle:
Recorre la cadena carácter por carácter.

- Comillas simples ('):
    - Si encuentra una comilla simple y no está dentro de comillas dobles (!qdoubles), activa la bandera qsimple.
- Comillas dobles ("):
    - Si encuentra una comilla doble y no está dentro de comillas simples (!qsimple), activa la bandera qdoubles.

Comprobación Final:
Si alguna de las banderas (qsimple o qdoubles)
sigue activa, significa que hay comillas sin cerrar, y
retorna EXIT_FAILURE.

LLama seguidamente a get_path
GET_PATH

Obtiene las rutas de la variable PATH para buscar ejecutables.

Esta función busca la variable PATH en la lista de variables de entorno y la divide en una lista de rutas para buscar comandos

Flujo de Trabajo:

1. Si d->path ya tiene un valor, lo libera para evitar fugas de memoria.
2. Recorre la lista de variables de entorno (d->envp) para encontrar la variable PATH.
3. Cuando encuentra PATH, usa ft_split para dividir su contenido en un array de cadenas usando : como delimitador.
4. Almacena el resultado en d->path.

Resultado: ["/usr/bin", "/bin", "/usr/local/bin", NULL]

Llama a get_cmds

GET_CMDS

La función get_cmds se encarga de dividir la entrada del usuario (d->prompt) en una lista de comandos individuales, separados por el carácter | (pipe). Además, verifica errores de sintaxis como pipes vacíos o mal formados, y maneja adecuadamente las comillas simples (') y dobles (").

Elimina los espacios o tabulaciones al inicio de la entrada para no procesar un comando vacío.
Si la entrada solo contiene espacios en blanco, retorna un error (EXIT_FAILURE).
Usa splits_cmd para dividir la entrada en comandos separados por el carácter |. Cada comando resultante se almacena en el array d->cmd.Si splits_cmd falla (por ejemplo, debido a un error de memoria o sintaxis), retorna un error.

LLama seguidamente a splits_cmd

La función `splits_cmd` tiene como objetivo dividir una cadena de entrada (`str`) en múltiples subcadenas, utilizando un delimitador especificado (`c`), y devolviendo un arreglo de cadenas de caracteres (un arreglo de `char *`).

`i`: Se utilizará como un índice para recorrer las posiciones de los subcomandos en el arreglo `s`.
`j`: Se utilizará para recorrer la cadena `str` a medida que se procesan los subcomandos.
`s`: Un arreglo de punteros a `char`, que almacenará las subcadenas

`malloc`: Reservamos memoria para el arreglo de cadenas `s`. El tamaño de este arreglo será el número de "comandos" (subcadenas) que se obtendrán al dividir la cadena `str` con el delimitador `c`. La función `count_cmd` nos proporciona cuántos subcomandos habrá.

`++i < count_cmd(str, c, 0)`: Este bucle se ejecuta tantas veces como el número de comandos (subcadenas) que se espera, que es determinado por la función `count_cmd`.

`i` se incrementa en cada iteración para almacenar cada subcomando en el arreglo `s`.

`while (str[j] == c)`: Este bucle salta cualquier ocurrencia del delimitador `c` (por ejemplo, un pipe `|`), avanzando el índice `j` hasta que se encuentra con un carácter diferente.
`while (str[j] == ' ')`: Este bucle salta los espacios en blanco entre los delimitadores y las subcadenas, avanzando `j` hasta que se encuentra con un carácter no blanco.

`ft_substr(str, j, size_cmd(str, c, j))`: Usamos la función `ft_substr` para extraer una subcadena de `str` comenzando desde el índice `j`, con un tamaño determinado por la función `size_cmd`. `size_cmd` calcula cuántos caracteres componen el subcomando (sin contar los delimitadores y espacios). Esta subcadena se almacena en `s[i]`.
`if (!(s[i]))`: Si `ft_substr` devuelve `NULL` (lo que significa que hubo un error al crear la subcadena), liberamos toda la memoria previamente asignada con `free_split(s)` y retornamos `0` (indicando un error).

`j = j + size_cmd(str, c, j)`: Después de extraer la subcadena, avanzamos el índice `j` para que apunte al siguiente "comando" o subcadena. Esto se hace sumando a `j` el tamaño de la subcadena recién extraída.

La función `count_cmd` tiene como objetivo contar cuántos comandos (o subcadenas) existen en la cadena de entrada `str`, separadas por el delimitador `c`. Esta función también verifica si hay errores de sintaxis, como el uso incorrecto de los delimitadores.

`s`: Se declara una variable `s` de tipo `char`, que se usará más tarde en la función `counts_extend` para manejar ciertos caracteres especiales (como comillas o redirecciones).

El bucle `while (*str != '\0')` recorre la cadena `str` hasta que se alcanza el final de la cadena (es decir, hasta que `*str` es `\0`).

`counts_extend(&str, c, &s)`: Se llama a la función `counts_extend`, que probablemente maneja ciertos casos especiales, como comillas y operadores de redirección (`<`, `>`, etc.). La función `counts_extend` también avanza el puntero `str` y establece el valor de `s` dependiendo del tipo de carácter especial encontrado.

Si `counts_extend` devuelve `-2`, se interpreta como un error de sintaxis. En ese caso, la función `count_cmd` retorna `-2`, indicando que hubo un error.

`if (*str == c && str++)`: Esta línea verifica si el carácter actual es igual al delimitador `c` (por ejemplo, un `|` para separar comandos). Si es así, avanza el puntero `str` a la siguiente posición.

`str++`: La expresión `str++` avanza el puntero `str` al siguiente carácter, es decir, se mueve a la siguiente parte de la cadena después del delimitador.

El bucle salta los espacios en blanco que siguen al delimitador `c`. Si encuentra espacios, avanza el puntero `str` hasta que ya no haya más espacios.

Después de saltar los espacios, se verifica si el siguiente carácter es otro delimitador `c` o el final de la cadena `\0`. Si es uno de estos casos, significa que hay un error de sintaxis (por ejemplo, dos delimitadores consecutivos sin comandos entre ellos).

Si se detecta un error, se imprime el mensaje de error `Error: Syntax Pipes` y la función retorna `-2` para indicar un error de sintaxis.

Si no se ha producido ningún error, se incrementa el contador `i` en uno. Este contador lleva el registro de cuántos "comandos" o subcadenas se han encontrado hasta ahora.

La función `counts_extend` tiene como objetivo procesar una cadena de caracteres (`s1`), buscando ciertos tokens de sintaxis como redirecciones (`<`, `>`) y comillas (`'`, `"`), mientras maneja los errores de sintaxis relacionados con el uso del delimitador `c`. Si se detectan errores, devuelve un código de error (`-2`), si no, avanza el puntero `s1` y continúa procesando la cadena.

La función comienza comprobando si el primer carácter al que apunta `*s1` es igual al delimitador `c` (por ejemplo, un pipe `|`).Si es así, se imprime un mensaje de error (`"Error: Syntax token |"`) indicando que el delimitador c(en este caso, probablemente un pipe) está mal colocado, y la función retorna-2`, lo que significa un error de sintaxis.

El bucle recorre la cadena de caracteres apuntada por `*s1`, avanzando hasta que encuentra el delimitador `c` o el final de la cadena (`\0`).

`**s1` es el valor del carácter al que apunta el puntero `*s1`, por lo que el bucle continuará hasta que se encuentre con un delimitador `c` o con el final de la cadena.

La función comprueba si el carácter actual es una redirección (`<` o `>`). Si es así, avanza el puntero `*s1` al siguiente carácter con `(*s1)++`.

Después de encontrar una redirección, salta los espacios en blanco que siguen (con el bucle `while (**s1 == ' ') (*s1)++`).Si después de los espacios encuentra el delimitador `c`, se imprime un mensaje de error que indica que la sintaxis de los tokens es incorrecta (`"Error: Syntax token |"), y se devuelve -2` para indicar un error de sintaxis.

Si el carácter actual es una comilla simple (`'`) o doble (`"`), se guarda este carácter en `*s2` (esto indica qué tipo de comilla se encontró).

Luego, el puntero `*s1` se avanza al siguiente carácter, y el bucle `while (**s1 != *s2)` avanza hasta que encuentra el cierre de la comilla correspondiente (es decir, se busca el mismo tipo de comilla que se encontró al principio).

Este paso permite que la función maneje correctamente las comillas dentro de la cadena, asegurándose de que las comillas de apertura y cierre estén balanceadas.

Después de procesar un carácter (ya sea parte de una redirección, una comilla o un espacio), el puntero `*s1` se avanza al siguiente carácter de la cadena.

Este paso asegura que la función recorra toda la cadena y procese correctamente cada uno de los caracteres, saltando los que ya han sido procesados

Entrada: `"echo 'hello | world' | wc"`
Llamada: `count_extend("echo 'hello | world' | wc", '|', &s2)`

La función `size_cmd` calcula el tamaño de un comando o una parte de la cadena de caracteres (`str`), teniendo en cuenta que puede haber comillas simples (`'`) o dobles (`"`) que rodean argumentos, y el delimitador `c` que indica el final del comando o parte de la cadena.

El propósito de `size_cmd` es determinar la longitud de una porción de la cadena `str`, que va desde la posición `j` hasta que encuentra el delimitador `c` o el final de la cadena (`'\0'`). Durante este proceso, la función también maneja adecuadamente las comillas, asegurándose de contar correctamente los caracteres dentro de las comillas.

La variable `size` se inicializa en 0 y se utilizará para contar la longitud de la porción de la cadena.

`s` es una variable temporal para almacenar el carácter de la comilla (simple o doble) cuando se encuentra dentro de un conjunto de comillas.

El bucle recorre la cadena `str` comenzando desde el índice `j` hasta que se encuentra con el delimitador `c` o el final de la cadena (`'\0'`).

Si se alcanza el delimitador `c` o el final de la cadena, el bucle se detiene.

Si el carácter en `str[j]` es una comilla simple (`'`) o doble (`"`), se entra en este bloque.

Se guarda el tipo de comilla (`'` o `"`) en la variable `s`.

Se avanza el índice `j` al siguiente carácter (`j++`).

Se incrementa `size` en 1 para contar la comilla de apertura.

Luego, un bucle interno avanza el índice `j` hasta que se encuentra con la misma comilla de cierre (es decir, la misma comilla que se guardó en `s`).

Durante este proceso, `size` se incrementa para contar cada carácter dentro de las comillas.

Este bloque asegura que todo lo que esté dentro de comillas se cuente correctamente como parte de la longitud del comando, incluyendo las comillas mismas.

Si el carácter en `str[j]` no es una comilla, simplemente se incrementa `size` en 1 para contar el carácter en `str[j]`.

Luego, el índice `j` se incrementa para pasar al siguiente carácter de la cadena.

Ejemplo:

Entrada: `"echo 'hello world' | wc"`
Llamada: `size_cmd("echo 'hello world' | wc", '|', 5)`
Proceso:

- `size = 11` **(por** `'hello world'`**)**

LLama seguidamente a expand(data)

EXPAND

Función	Propósito
`expand`	Orquesta todas las expansiones (`$`, `~`) y actualiza los comandos en `d->cmd`.
`dollar_expand`	Expande variables de entorno (`$VAR`) y valores especiales (`$?`).
`prime_expand`	Maneja la expansión de `~` para reemplazarlo con el directorio `HOME`.
`next_expand`	Procesa texto literal, manejando correctamente las comillas simples (`'`).
`envp_content`	Busca el valor de una variable de entorno o valores especiales como `$?`.

Ejemplo Completo

Entrada:

bash
```
echo "My home is ~ and my user is $USER. Last status: $?"
```

1. Inicialización:
    - Se establece `flag = 0` y `expand = NULL`.
2. Expansión de Comandos:
    - `~`: Se reemplaza con el valor de `HOME`.
    - `$USER`: Se reemplaza con el valor de la variable `USER`.
    - `$?`: Se reemplaza con el último código de salida (`data->status`).
3. Resultado Final:
   ```
   d->cmd[0] = "My home is /home/user and my user is john. Last
   status: 0";
   ```

La función expand realiza la expansión de variables dentro de los comandos, probablemente para un shell. Se encarga de procesar las cadenas de comandos contenidas en d->cmd, y dentro de cada cadena, expande las variables que están precedidas por el signo de dólar ($) o las comillas (seguramente comillas simples o dobles). La función recorre los comandos y los modifica según las expansiones encontradas.

flag: Se utiliza como un indicador para controlar el estado de la expansión (por ejemplo, si ya se procesó una variable o si una expansión ya fue aplicada).

expand: Es un puntero a la cadena que va a almacenar temporalmente el resultado de la expansión.

El bucle recorre cada comando en d->cmd. El índice d->i se usa para acceder a cada comando. El bucle continuará hasta que se llegue al final de los comandos.

d->j: Es un índice interno utilizado dentro del comando para recorrer sus caracteres.

i: Es otro índice que probablemente está relacionado con la posición de expansión de variables (como $ o comillas) dentro del comando.

Este bucle recorre los caracteres de un comando específico (d->cmd[d->i]). Para cada carácter, se evalúan las posibles expansiones, como variables de entorno (precedidas por $) o comillas.

Estas funciones (dollar_expand y prime_expand) parecen encargarse de las expansiones de las variables o comillas.

- **dollar_expand** probablemente maneja la expansión de variables precedidas por $, como $HOME o $USER.
- **prime_expand** probablemente maneja la expansión de comillas, que pueden representar variables o expandir a lo largo de las cadenas (como variables de entorno dentro de comillas dobles).

Ambas funciones toman parámetros como el índice i, el flag para controlar el estado, y el puntero expand que se irá modificando con el resultado de las expansiones.

Primero, se libera la memoria ocupada por el comando original (d->cmd[d->i]), que ya no es necesario porque se va a reemplazar.

Se asigna a d->cmd[d->i] el valor de expand, que ahora contiene la cadena del comando con las expansiones realizadas.

Luego, se libera la memoria de expand para evitar fugas de memoria.Después de procesar el comando actual, se incrementa d->i para pasar al siguiente comando en d->cmd.

Después de procesar todos los comandos, el índice d->i se restablece a 0, probablemente para preparar la estructura d para futuros usos o para reiniciar el procesamiento en otra parte del código.

La función dollar_expand se encarga de expandir las variables de entorno dentro de un comando, específicamente aquellas que están precedidas por el signo de dólar ($). En un shell, esto significa que se deben sustituir las variables de entorno por sus valores. Por ejemplo, si tienes una variable $HOME, esta se expandirá a la ruta del directorio de inicio del usuario.

Se verifica si el carácter actual en el comando (d->cmd[d->i][d->j]) es un signo de dólar ($).

Además, se comprueba que el flag sea 0, lo que indica que no se ha procesado ninguna expansión previamente.

Si el siguiente carácter es el final de la cadena (\0) o un espacio, entonces no hay ninguna variable para expandir, solo se añade el signo de dólar tal cual a la cadena expand.

La función ft_strjoin_gnl se usa para concatenar el valor de expand con el signo de dólar ("$").

Luego, el índice d->j se incrementa para continuar procesando el siguiente carácter.

Si el signo de dólar está seguido por una variable (es decir, no está seguido por un espacio o final de línea), se empieza a procesar la variable.

Se incrementa d->j para saltar el $ y se asigna el valor de d->j a *i, lo que marca el comienzo de la variable.

Luego, se avanza con el índice d->j hasta encontrar un espacio, otro signo de dólar, o una comilla simple o doble. Estos caracteres indican el final de la variable.

Con la función ft_substr, se extrae el nombre de la variable entre *i y d->j.

Después, se llama a envp_content(d, aux) para obtener el valor de la variable de entorno correspondiente al nombre extraído.

La función ft_strjoin_gnl se utiliza para concatenar el valor de expand con el contenido de la variable de entorno.

Finalmente, se libera la memoria ocupada por aux (el nombre de la variable) para evitar fugas de memoria.

La función `envp_content` busca en una lista de variables de entorno (presumiblemente representadas por una estructura `t_envp`) el valor asociado a un nombre de variable dado (por ejemplo, `HOME` o `?` para el código de salida). Si encuentra la variable, devuelve su valor. Si no, devuelve una cadena vacía.

Si el `str` recibido es el símbolo de pregunta `?`, que se utiliza típicamente para representar el código de salida de un proceso en muchos shells, la función convierte el valor de `d->status` (probablemente el código de salida del último comando ejecutado) en una cadena de caracteres con `ft_itoa()`. Luego, devuelve esa cadena que representa el código de salida.

La función recorre la lista `d->envp`, que es una lista enlazada de estructuras `t_envp` que almacenan las variables de entorno.

Para cada nodo en la lista, compara el nombre de la variable (`tmp->name`) con el nombre proporcionado en `str` usando `ft_strncmp()`. Si encuentra una coincidencia (el nombre de la variable coincide con el argumento `str`), entonces...

- Toma el valor de `tmp->content`, que representa el contenido de la variable de entorno, y elimina el primer carácter (posiblemente un `=`) usando `ft_substr()`.
- Luego devuelve ese valor sin el primer carácter.

Si no encuentra ninguna coincidencia en la lista de variables de entorno, la función devuelve una cadena vacía.

La función prime_expand maneja la expansión de ciertos caracteres especiales en un comando, como el tilde (~) que se expande a la ruta del directorio home del usuario. Además, también controla la expansión de comillas simples para que no se realicen otras expansiones dentro de ellas (marcando la expansión de texto dentro de un contexto protegido por comillas). La función recorre los caracteres del comando buscando el tilde y otras expansiones.

La función empieza con un bucle que recorre los caracteres de la cadena del comando (d->cmd[d->i]).

Se asegura de que el carácter actual no sea un signo de dólar ($) o que el flag no esté activado. Si el flag está activado, significa que no se deben hacer expansiones dentro de las comillas simples (por ejemplo, no expandir dentro de echo 'HOME').

Se verifica si el carácter actual es un tilde (~) y si no está dentro de comillas (es decir, si flag es 0).

También se comprueban otras condiciones, como:

- Si el tilde está rodeado por espacios.
- Si está al final de la cadena.
- Si está seguido por una barra (/), lo que puede indicar que es parte de una ruta.

Estas condiciones permiten determinar si el tilde debe ser expandido a la ruta del directorio home del usuario.

Si la expansión del tilde es válida (según las condiciones anteriores), la función verifica si el carácter actual es una comilla simple (').

- Si es así, alterna el valor de flag para indicar que el contenido dentro de las comillas no debe ser expandido más adelante.

Luego, busca el valor de la variable de entorno HOME mediante envp_content(d, "HOME") y lo guarda en la variable next.

A continuación, se concatena este valor al contenido de expand usando la función ft_strjoin_gnl.

Después, se libera la memoria de next para evitar fugas de memoria.

Se incrementa d->j para pasar al siguiente carácter.

Si el carácter actual no es un tilde, se llama a la función next_expand para realizar otras expansiones, probablemente relacionadas con otros caracteres especiales.

Después de completar la expansión, se limpia cualquier memoria temporal almacenada en d->aux (si es que se utilizó).

La función `next_expand` es responsable de manejar la expansión de cadenas en el comando, especialmente cuando el carácter actual no es un signo de dólar (`$`) ni un tilde (`~`). La función también se encarga de manejar el control de comillas simples (`'`), que deben ser ignoradas en la expansión.

La función primero verifica si el `flag` no está activado (lo que significa que no estamos dentro de comillas simples) y si el carácter actual no es un tilde (`~`).

Si ambas condiciones son verdaderas, la función comienza a realizar la expansión.

Se guarda la posición actual de `d->j` en `*i` para marcar el inicio de la cadena que será extraída para expansión.

La función entra en un bucle que recorre la cadena (`d->cmd[d->i]`) desde la posición `d->j` hasta que encuentra un signo de dólar (`$`), un tilde (`~`), o una comilla simple (`'`).

La expansión continúa mientras no se encuentre un signo de dólar (`$`), a menos que el `flag` esté activado (lo que indica que estamos dentro de comillas simples).

Si el carácter actual es una comilla simple (`'`), se alterna el valor de `flag` para controlar si estamos dentro de comillas. Si el `flag` estaba desactivado (fuera de comillas), ahora se activa, y si estaba activado (dentro de comillas), se desactiva.

Se incrementa `d->j` para avanzar al siguiente carácter en la cadena.

Una vez que se ha completado el recorrido de la cadena (sin encontrar un signo de dólar o tilde, o habiendo cambiado de estado con las comillas), se extrae una subcadena desde la posición `*i` hasta `d->j` usando `ft_substr`.

Esta subcadena se concatena al resultado acumulado en `*expand` usando `ft_strjoin_gnl`.

Si el primer bloque de condiciones no se cumplió (es decir, si el carácter actual es un tilde o estamos dentro de comillas), el bloque `else` maneja este caso.

De nuevo, la función guarda la posición de `d->j` en `*i` y luego entra en un bucle similar al anterior, recorriendo la cadena hasta que encuentra un signo de dólar (`$`), un tilde (`~`), o una comilla simple (`'`).

Finalmente, al igual que en el bloque anterior, se extrae la subcadena desde `*i` hasta `d->j` y se concatena al resultado en `*expand`.

LLama seguidamente a parsing(data, 0, 0)

PARSING

Se encarga de analizar (parsear) la cadena de comandos (cmd) recibida para extraer tokens, comandos y archivos de entrada/salida, y luego agregar esos datos a una estructura de tipo lista de nodos (data->nodes).

El bucle principal recorre cada comando en data->cmd, que es una lista o array de cadenas que contiene los comandos que el usuario ha ingresado.Se detiene cuando encuentra un valor NULL en data->cmd, lo que indica que no hay más comandos que procesar.Se inicializa j a 0, que se usará como índice para recorrer cada comando.

Luego, si el comando en la posición i es una cadena vacía (es decir, data->cmd[i][j] == '\0'), la función retorna un error (EXIT_FAILURE), ya que no puede procesar un comando vacío.

Se asigna memoria para un nuevo nodo de tipo t_parser utilizando ft_calloc. Este nodo se utilizará para almacenar los datos del comando que se está analizando.

Se inicializan las variables fileout y filein, que probablemente se utilizarán más adelante para almacenar los archivos de salida y entrada redirigidos en el comando.

Este bucle avanza el índice j mientras encuentra espacios en blanco al principio del comando. Esto es para saltarse cualquier espacio extra antes de empezar a analizar el contenido del comando.

La función get_tokens se encarga de extraer los tokens (por ejemplo, los operadores, como los pipes, redirecciones, etc.) de la cadena del comando. Si esta operación falla (devuelve EXIT_FAILURE), se libera la memoria del nodo actual y la función retorna un error.

La función get_command obtiene el comando principal (la acción a realizar, como echo, ls, etc.) de la cadena del comando. Si falla, se libera la memoria del nodo y se retorna un error.

Después de obtener el comando principal, se vuelve a llamar a get_tokens para obtener más tokens que podrían estar presentes, como operadores de redirección, pipes, etc. Si la operación falla, se libera la memoria y se retorna un error.

La función input_files se encarga de identificar y manejar los archivos de entrada y salida que podrían estar presentes en el comando (por ejemplo, redirección de archivos). Si hay un error, se libera la memoria y se retorna un error.

Después de procesar el comando, se agrega el nodo data->node a la lista de nodos data->nodes utilizando la función ft_lstadd_back. Esta lista contiene todos los comandos que se van a ejecutar, con sus respectivos detalles, como los tokens, archivos de entrada/salida, etc.Se incrementa i para pasar al siguiente comando en data->cmd.

La función `get_tokens` es responsable de identificar y procesar los tokens de redirección de archivos (`<` y `>`) que se pueden encontrar en un comando. Estos tokens indican redirección de entrada o salida de archivos, y la función debe manejarlos adecuadamente, cerrando los archivos correspondientes si ya están abiertos y asegurándose de que las redirecciones se configuren correctamente.

Este bucle recorre los caracteres del comando en busca de tokens de redirección (`<` o `>`). Los índices `*i` y `*j` apuntan a la posición actual en la cadena del comando.

El bucle sigue ejecutándose mientras encuentre caracteres de redirección (`<` o `>`).

Si se encuentra un token de redirección, se comprueba si los archivos de entrada (`filein`) o salida (`fileout`) ya están abiertos:

- Si el archivo de entrada (`filein`) ya está abierto (es decir, no es igual a 0), se cierra con `close()`.
- Si el archivo de salida (`fileout`) ya está abierto (es decir, no es igual a 1), se cierra con `close()`.
- Las condiciones aseguran que los archivos solo se cierren si ya estaban abiertos previamente.

La función `get_token_filein` se encarga de procesar un token de redirección de entrada (`<`). Puede abrir el archivo correspondiente y asociarlo al nodo (`node`).

Si ocurre un error durante este proceso, la función retorna `EXIT_FAILURE`, lo que indica que algo salió mal al procesar el token de redirección de entrada.

Similar al caso de la redirección de entrada, la función `get_token_fileout` se encarga de procesar un token de redirección de salida (`>`). Puede abrir el archivo correspondiente y asociarlo al nodo (`node`).

Si ocurre un error al procesar la redirección de salida, se retorna `EXIT_FAILURE`.

Después de procesar los tokens de redirección, se restablecen las variables `data->size` y `data->a`. Estas variables podrían estar relacionadas con el procesamiento de otros tokens o elementos en el comando, y se resetean para asegurar que no haya interferencias con futuras operaciones.

Si todo ha ido bien, la función retorna `EXIT_SUCCESS`, indicando que los tokens de redirección se procesaron correctamente.

La función get_token_filein en el contexto de tu minishell es responsable de procesar los tokens relacionados con los redireccionamientos de entrada (< y <<) en los comandos.

Se inicializan dos banderas:

- flag_hered: Se utiliza para indicar si se está procesando un redireccionamiento de entrada heredado (es decir, <<).
- flag_token: Se marca como 1 para indicar que se está procesando un token.

Si el carácter actual en el comando (data->cmd[*i][*j]) es el símbolo de redireccionamiento de entrada <, se avanza al siguiente carácter (++(*j)).

Si después de < se encuentra un carácter nulo (fin de línea) sin más datos, se muestra un error de sintaxis y se retorna EXIT_FAILURE.

Si después de < aparece un >, también se considera un error de sintaxis, porque no es una secuencia válida.

i se encuentra otro <, se activa la bandera flag_hered para indicar que es un redireccionamiento heredado (<<).

Errores relacionados con <<: Si se detecta una secuencia incorrecta o se encuentra un > en un redireccionamiento heredado, la función muestra un mensaje de error y retorna EXIT_FAILURE:

Si hay espacios en blanco después del < o <<, la función avanza en el comando y llama a get_next_token para obtener el siguiente token. Si get_next_token falla, la función retorna un error.

Finalmente, se llama a get_last_token, que probablemente obtiene el archivo o entrada redirigida después del símbolo <. Si falla, la función retorna EXIT_FAILURE.
Finalización de la función: Si no se detectan errores, la función retorna EXIT_SUCCESS.

La función get_token_fileout maneja los tokens de redireccionamiento de salida (> y >>) en tu minishell. Similar a la función get_token_filein, esta función se encarga de verificar la sintaxis y procesar correctamente los símbolos de redirección de salida, con las siguientes especificaciones:

flag_add: Se utiliza para indicar si se está procesando un redireccionamiento de salida adicional (>>).

Flag_token: Se marca como 2 para identificar que este token es un redireccionamiento de salida.

Si el carácter actual en el comando (data->cmd[*i][*j]) es >, se avanza al siguiente carácter (++(*j)).

Si después de > se encuentra un carácter nulo (fin de línea), la función muestra un error de sintaxis y retorna EXIT_FAILURE.

Si después de > aparece un <, también se considera un error de sintaxis.

Si se encuentra otro >, se activa la bandera flag_add, indicando que es un redireccionamiento adicional (>>).

Si se detecta una secuencia incorrecta o se encuentra un < o > después de >>, la función muestra un mensaje de error y retorna EXIT_FAILURE:Si hay espacios en blanco después del > o >>, la función avanza en el comando y llama a get_next_token para obtener el siguiente token. Si get_next_token falla, la función retorna EXIT_FAILURE.

Finalmente, la función llama a get_last_token para obtener el archivo o salida redirigida después del símbolo > o >>. Si falla, la función retorna EXIT_FAILURE.Si no se detectan errores, la función retorna EXIT_SUCCESS.

La función get_next_token en tu minishell se encarga de verificar la validez del siguiente token en el comando

Si el carácter actual es el fin de línea (\0), lo que significa que el comando ha terminado o hay un error de sintaxis, se muestra el mensaje "syntax error" y la función retorna EXIT_FAILURE.

Si el siguiente carácter es un símbolo de redireccionamiento de entrada (<), se considera un error de sintaxis, ya que en este contexto no se espera que aparezca un < de forma aislada, y la función retorna EXIT_FAILURE con el mensaje de error.

Similar a la verificación del símbolo <, si el siguiente carácter es un símbolo de redireccionamiento de salida (>), también se considera un error de sintaxis en este contexto y la función retorna EXIT_FAILURE.Si no se encuentra ninguno de los errores mencionados anteriormente, la función retorna EXIT_SUCCESS, indicando que el siguiente token es válido.

La función get_last_token en tu minishell tiene como propósito identificar el último token en una cadena de comandos, manejando especialmente los casos de redireccionamiento de entrada (<) y salida (>). Esta función también maneja el almacenamiento de estos tokens (archivos) en las variables correspondientes (filein o fileout) y asegura que no haya errores de sintaxis.

d->size se inicializa en 0 y se utilizará para contar el tamaño del token que se está procesando.

d->a se inicializa en 0 y se utiliza como índice para almacenar los caracteres en filein o fileout.

Este bucle recorre la cadena del comando, caracter por caracter, hasta que encuentra un espacio (' ') o el final de la cadena ('\0').

Si durante este recorrido encuentra un símbolo de redirección (< o >), retorna un error de sintaxis (EXIT_FAILURE), porque no se espera que estos símbolos aparezcan en medio de un token.

Si el token es para redirección de entrada (flag_token == 1), se reserva memoria para filein.

Si el token es para redirección de salida (flag_token == 2), se reserva memoria para fileout.

Se asegura de que la memoria se reserve para el tamaño del token más un espacio adicional para el carácter nulo ('\0').

El valor de d->size se actualiza para representar la posición correcta en la cadena del comando, ajustándose a la longitud del token que se está procesando.

Este bucle recorre el token y copia cada carácter en la variable filein o fileout según el valor de flag_token (1 para filein, 2 para fileout).

d->a es el índice de almacenamiento, y se incrementa a medida que se almacenan los caracteres.

Después de procesar el token, se llama a una función auxiliar (get_last_token_util) para realizar tareas adicionales (probablemente relacionadas con el manejo de archivos o tokens).

Si esta función devuelve 1, se libera la memoria reservada para filein y se retorna EXIT_FAILURE.

Si todo el proceso se completa sin problemas, la función retorna EXIT_SUCCESS, indicando que el token se ha procesado correctamente.

La función get_last_token_util se encarga de manejar las acciones relacionadas con la apertura de archivos para redirección de entrada y salida, así como de procesar las redirecciones heredadas (por ejemplo, con el operador << para redirección heredada en una shell)

Si flag_hered es 0 (lo que indica que no es una redirección heredada) y flag_token es 1 (lo que indica que estamos manejando un token de entrada), se abre el archivo para redirección de entrada (filein) con el modo O_RDONLY (solo lectura).

Si el archivo se abre correctamente, su descriptor de archivo se asigna a (*node)->filein.

Si flag_add es 0 (lo que indica que no es una adición a un archivo existente) y flag_token es 2 (lo que indica que estamos manejando un token de salida), se abre el archivo para redirección de salida (fileout) con los siguientes flags:

- O_WRONLY: solo escritura.
- O_CREAT: crea el archivo si no existe.
- O_TRUNC: trunca el archivo a 0 bytes si ya existe.
- El permiso 0644 se asigna al archivo.

Si el archivo se abre correctamente, su descriptor de archivo se asigna a (*node)->fileout.

Si flag_hered es 1 (lo que indica que estamos tratando con una redirección heredada, como <<) y flag_token es 1 (lo que indica que estamos manejando un token de entrada), se llama a la función ft_heredoc.

La función ft_heredoc probablemente maneje la creación de un archivo temporal o el manejo de la entrada heredada en lugar de un archivo real.

Si flag_add es 1 (lo que indica que estamos usando el operador >> para añadir al archivo en lugar de sobrescribirlo) y flag_token es 2 (lo que indica que estamos manejando un token de salida), se abre el archivo para redirección de salida con los siguientes flags:

- O_WRONLY: solo escritura.
- O_CREAT: crea el archivo si no existe.
- O_APPEND: añade al final del archivo si ya existe.

El permiso 0644 se asigna al archivo.

Si el archivo se abre correctamente, su descriptor de archivo se asigna a (*node)->fileout.

Si alguno de los descriptores de archivo (filein o fileout) es igual a -1 (lo que indica que ocurrió un error al abrir el archivo), se imprime un mensaje de error y la función retorna EXIT_FAILURE.

También se establece el estado de la variable data->status a 1, lo que podría indicar un error en el proceso.

Si flag_token es 1 (token de entrada), se libera la memoria asignada para data->filein.

Si flag_token es 2 (token de salida), se libera la memoria asignada para data->fileout.

Después de procesar el token, el índice j se incrementa mientras encuentre espacios en blanco en la cadena del comando.

Si todo el proceso se completa sin problemas, la función retorna EXIT_SUCCESS, indicando que el procesamiento de los tokens y las redirecciones fue exitoso.

La función get_command es responsable de identificar y procesar el comando en una posición dada de la cadena de comandos (data->cmd). Esta función se encarga de extraer una subcadena que representa el comando y luego lo procesa.

Esta condición verifica si el carácter actual de la cadena data->cmd[*i][*j] no es un espacio en blanco (' '), un final de cadena ('\0'), o uno de los tokens de redirección ('>' o '<').

Si el carácter no es ninguno de esos, significa que hemos encontrado el inicio de un comando y podemos procesarlo.

data->a se establece en la posición actual de *j, que es el inicio de la parte del comando que se está procesando.

data->size se inicializa a 0, y se usará para contar la longitud del comando que estamos procesando.

Este bucle sigue ejecutándose mientras no se encuentren los tokens de redirección (<, >) ni el final de la cadena (\0).

Básicamente, este bucle se encarga de recorrer todos los caracteres del comando hasta que se encuentra con un delimitador (redirección o fin de cadena).

Si se encuentra una comilla simple (') o doble ("), se entra en el bloque que maneja los literales entre comillas.

data->quote almacena el tipo de comilla (' o ") para poder detectar el cierre de la misma.

Se aumenta data->size por cada carácter dentro de las comillas.

El bucle interno recorre todos los caracteres hasta que encuentra la comilla de cierre correspondiente (comilla simple o doble).

Por cada carácter dentro del comando (que no sea un delimitador como <, >, o un final de cadena), data->size se incrementa para contar la longitud del comando.

Después, el índice *j se incrementa para avanzar al siguiente carácter.

Una vez que se ha recorrido el comando, se pasa la información a la función process_command para que procese el comando (probablemente construyendo el nodo o ejecutando el comando, dependiendo de la implementación de esa función).

Si process_command devuelve EXIT_FAILURE, significa que hubo un error, y se retorna EXIT_FAILURE en la función get_command para indicar que algo salió mal.

Al final de la función, la memoria ocupada por data->str se libera, ya que probablemente se usó para almacenar una cadena temporal y ya no se necesita más.

Si todo se ha ejecutado correctamente, la función retorna `EXIT_SUCCESS`, indicando que el comando se procesó correctamente.

La función process_command es responsable de procesar un comando extraído de la cadena de entrada y almacenarlo en la estructura t_parser.

Se asigna memoria para data->str con un tamaño igual a data->size (que es la longitud del comando que se está procesando), más un byte adicional para el carácter nulo \0 que marcará el final de la cadena.

ft_calloc se utiliza para inicializar toda la memoria a cero.

data->b es el índice para llenar data->str. Se inicializa en 0 porque estamos comenzando a llenar la cadena desde el principio.

Este bucle recorre los caracteres de data->cmd[*i] hasta encontrar un delimitador de redirección (<, >) o el final de la cadena (\0).

La variable data->a es el índice del carácter actual en data->cmd[*i], y el bucle procesa cada carácter hasta que se alcanza un delimitador o el final del comando.

Si se encuentra el carácter de comilla (simples o dobles) que se había almacenado en data->quote (esto indica que estamos dentro de una cadena entre comillas), se copia el carácter de comilla en data->str y se avanza el índice data->a y data->b.

Luego, el bucle interno recorre los caracteres entre las comillas y los copia en data->str hasta encontrar la comilla de cierre correspondiente.

Si el carácter no es una comilla, simplemente se copia en data->str y se incrementan los índices data->a y data->b para seguir procesando el siguiente carácter

Una vez que hemos construido el comando completo (almacenado en data->str), se pasa a la función get_words. Esta función probablemente divide el comando en palabras o tokens, separándolos por el espacio (' ').

El resultado se almacena en (*node)->all_cmd, que es una lista de las palabras/tokens extraídos del comando.

Si get_words devuelve NULL (lo que significa que algo salió mal al dividir el comando), se libera la memoria de data->str y se retorna EXIT_FAILURE para indicar que hubo un error.

Si todo salió bien, se retorna EXIT_SUCCESS para indicar que el comando se procesó correctamente.

La función input_files es responsable de manejar la redirección de entrada de un comando en una shell (como en la construcción de cmd < file)

Este bucle recorre la cadena de comandos data->cmd[*i] desde el índice *j hasta que se alcance el final de la cadena ('\0').

data->a es el índice donde se almacenará el nombre del archivo de entrada.

data->size se utiliza para contar los caracteres hasta encontrar el archivo de entrada.

Este bucle recorre la cadena data->cmd[*i] desde el índice *j hasta encontrar un espacio (' ') o el final de la cadena.

Se incrementa data->size para contar la longitud del nombre del archivo.

Se asigna memoria para data->filein con un tamaño igual a data->size + 1 para almacenar el nombre del archivo de entrada y el carácter nulo ('\0').

Después de contar el tamaño del archivo de entrada, este bloque copia el nombre del archivo desde data->cmd[*i][data->size] en el arreglo data->filein, hasta encontrar un espacio o el final de la cadena.

Si ya se había abierto un archivo de entrada en (*node)->filein, se cierra antes de abrir uno nuevo.

Se intenta abrir el archivo de entrada data->filein en modo lectura (O_RDONLY).

El descriptor de archivo se guarda en (*node)->filein.

Si el archivo no se puede abrir (open devuelve -1), se libera la memoria de data->filein y se imprime un mensaje de error. Luego, se retorna con un error.

Después de procesar el archivo de entrada, se saltan los espacios en la cadena de comandos para seguir con el procesamiento del siguiente token.

Si todo salió bien, la función retorna EXIT_SUCCESS.

La funcion llama a `if (data->nodes == NULL)`

Aquí se comprueba si la estructura data tiene un miembro nodes que es un puntero.

data->nodes probablemente sea una lista o conjunto de nodos que representan los comandos o las partes del comando que se van a ejecutar.

Si data->nodes es NULL, significa que no se encontraron nodos válidos o no se procesaron correctamente los comandos.

- Esto podría suceder si, por ejemplo, el comando no tiene una sintaxis válida después del análisis de parsing.

La funcion llama a `proccess route`

PROCCESS_ROUTE

La función process_route realiza varias comprobaciones y manipulaciones en una lista de nodos que contienen comandos, rutas y otras configuraciones.

tmp = data->nodes;: Se asigna la lista de nodos de data a un puntero temporal tmp. data->nodes probablemente sea una lista enlazada que contiene nodos de tipo t_parser, donde cada nodo tiene información sobre un comando.

data->flag_pipe = 0;: Se inicializa la bandera flag_pipe a 0. Esta bandera probablemente se utiliza para indicar si hay un pipe (|) presente entre los comandos (lo cual es común en shells).

Este bucle recorre los comandos en data->cmd y los nodos de tmp, mientras haya un comando presente en data->cmd[i] y que el nodo actual contenga comandos (all_cmd).

get_route: Esta función parece encargarse de procesar la ruta del comando actual. Recibe data y el nodo actual (tmp->content), con -1 como parámetro adicional. Si get_route devuelve 1, se establece data->status = 127 y se retorna EXIT_FAILURE.

- El valor 127 generalmente indica un error en el shell relacionado con un comando no encontrado, lo que sugiere que si la ruta del comando no es válida, se marca un error de ejecución.

Si el índice i es igual a 1, se establece la bandera flag_pipe en 1. Esto sugiere que, en algún punto de la ejecución, se ha encontrado un pipe (|) o se está configurando la ejecución para que los comandos sean ejecutados en una cadena de pipes.

i++: Se incrementa el índice i, lo que sugiere que el bucle avanza al siguiente comando o token en data->cmd.

tmp = tmp->next;: Si hay un siguiente nodo en la lista (tmp->next != NULL), se avanza al siguiente nodo en la lista tmp.

Después de salir del bucle, si el comando del nodo actual (all_cmd) es NULL, entonces se establece route en NULL. Esto indica que no hay una ruta definida para ese comando, probablemente porque el comando no se encontró o no tiene una ruta válida.

Finalmente, si todo se procesó correctamente, la función retorna EXIT_SUCCESS para indicar que la operación fue exitosa.

La función get_route está encargada de obtener la ruta del comando especificado en un nodo, verificando si el comando está en las rutas de búsqueda del sistema (por ejemplo, las rutas definidas en la variable de entorno PATH). Si no encuentra el comando en esas rutas, también verifica si el comando es un ejecutable en el directorio actual.\

Se verifica si el primer comando en el nodo (node->all_cmd[0]) es NULL. Si es así, significa que no hay un comando válido, por lo que se imprime un mensaje de error y se retorna 1 para indicar un fallo.

Se llama a la función check_builts, que probablemente verifica si el comando actual es un comando interno del shell (como cd, echo, etc.). Si es un built-in, no se necesita una ruta de archivo, por lo que la función retorna con EXIT_SUCCESS y no sigue buscando la ruta.

Búsqueda en las rutas definidas en data->path:

- El código recorre cada ruta en el array data->path (que es el valor de la variable de entorno PATH).
- Para cada ruta, se construye el nombre completo del comando utilizando ft_strjoin. Primero, se une la ruta (data->path[i]) con una barra (/), luego se agrega el nombre del comando (node->all_cmd[0]).
- **access(aux2, F_OK)**: Verifica si el archivo existe.
- **access(aux2, X_OK)**: Verifica si el archivo es ejecutable.
- Si ambas condiciones son verdaderas, significa que el archivo existe y es ejecutable, por lo que se asigna esta ruta a node->route y se retorna con EXIT_SUCCESS.
- Si no se encuentra el archivo, se libera la memoria de aux2 y se sigue buscando en las siguientes rutas.

Si el comando no se encuentra en las rutas del PATH, el código verifica si el comando está en el directorio actual.

- **get_dir(node->all_cmd[0]) == 0**: Esta función probablemente verifica si el comando puede ser encontrado en el directorio actual.
- **access(node->all_cmd[0], F_OK)**: Verifica si el archivo existe en el directorio actual.
- **access(node->all_cmd[0], X_OK)**: Verifica si el archivo es ejecutable en el directorio actual.
- Si ambas condiciones son verdaderas, el comando se encuentra en el directorio actual y se asigna su ruta a node->route.

Si no se encontró la ruta del comando después de buscar en las rutas del PATH y en el directorio actual, se imprime un mensaje de error y la función retorna 1.

Si se encontró la ruta del comando (ya sea en el PATH o en el directorio actual), se retorna EXIT_SUCCESS, indicando que la ruta se ha asignado correctamente.

La función get_dir verifica si un directorio existe y es accesible

La función opendir intenta abrir el directorio especificado por el argumento aux, que es una cadena de caracteres (probablemente una ruta de directorio).

Si aux es un directorio válido y se puede abrir, opendir devuelve un puntero a un tipo DIR. Si no puede abrir el directorio, retorna NULL.

Si dir no es NULL, significa que el directorio fue abierto exitosamente. En ese caso:

- Se cierra el directorio utilizando closedir(dir) para liberar los recursos asociados con el directorio.
- La función retorna 1 indicando que el directorio es válido y se puede acceder.

Si dir es NULL, significa que el directorio no existe o no es accesible. En este caso, la función retorna 0, indicando que el directorio no es válido o no se puede acceder.

La función check_builts verifica si el comando almacenado en node->all_cmd[0] corresponde a uno de los comandos internos de la shell, como cd, echo, env, pwd, unset, export, o exit. Si encuentra una coincidencia, asigna un valor específico a node->route y retorna un código de error para indicar que se trata de un comando interno (built-in)

Primero se verifica si node->all_cmd[0] (el comando) no es NULL y si empieza con la cadena "cd".

Si es cierto, la función asigna "b" a node->route (lo que indica que es un comando interno) y retorna EXIT_FAILURE para indicar que no se debe buscar una ruta para ese comando.

Se repite el mismo proceso para otros comandos internos (echo, env, pwd, unset, export, exit).

Si el comando coincide con uno de estos, se asigna el valor "b" a node->route y se retorna EXIT_FAILURE.

Si no se ha encontrado ninguna coincidencia con los comandos internos, se retorna EXIT_SUCCESS, lo que indica que el comando no es un built-in y debe ser procesado como un comando externo.

La funcion llama a executer

EXECUTER

Es responsable de ejecutar los comandos de un programa en una shell personalizada, gestionando tanto la ejecución de comandos individuales como los de una secuencia con pipes (tuberías)

Se configuran dos señales:

- SIGINT: se asocia con la función sig_ctrlc, lo que generalmente sirve para manejar la señal de interrupción (Ctrl+C) durante la ejecución del proceso.
- SIGQUIT: se asocia con la función sig_ctrlslash, que maneja la señal de salida (Ctrl+), generalmente usada para terminar un proceso de forma "forzada".

Se verifica que node no sea nulo, que node->all_cmd no sea nulo y que al menos el primer comando en node->all_cmd esté presente. Esto asegura que hay comandos para ejecutar.

Si data->flag_pipe es igual a 1, se indica que estamos manejando una secuencia de comandos que utiliza pipes.

Si el comando es cat y no tiene más argumentos (node->all_cmd[1] == NULL) y no tiene redirección de entrada (node->filein == 0), entonces se llama a la función catnls, que probablemente maneje una operación especial para el comando cat. Si la ejecución de catnls es exitosa (== 0), la función termina de manera exitosa (EXIT_SUCCESS).

Si no se trata del caso cat anterior, la función entra en un bucle para procesar los elementos de aux (probablemente una lista de comandos encadenados con pipes).

Si el siguiente comando es el último en la lista (aux->next == NULL) y la ruta del comando es "b" (probablemente para comandos internos como los built-ins), entonces se llama a la función ex_builts para ejecutar comandos internos.

Si no es el último comando o no es un comando interno, se llama a ex_routepipes para manejar la ejecución de comandos con pipes.

El bucle continúa procesando cada nodo en la lista aux.

Si no se está trabajando con pipes (data->flag_pipe == 0), la función simplemente llama a ex_route, que probablemente maneje la ejecución de un único comando sin pipes.

Si todo el proceso se ejecuta correctamente, la función retorna EXIT_SUCCESS, indicando que la ejecución fue exitosa.

La función catnls está diseñada para manejar un caso específico relacionado con un comando cat que se procesa de una manera especial en tu shell personalizada.

Se declara una variable j que se utilizará como contador.

tmp es un puntero a la lista data->nodes, que probablemente contenga los nodos de los comandos que se están procesando.

line es un puntero que se utilizará para almacenar las líneas leídas

La función verifica si se cumplen varias condiciones sobre los comandos en los nodos de data->nodes:

- **Primer comando**: El primer comando debe ser "car" (el primer comando en el nodo tmp).
- **Segundo comando**: El segundo comando, en el siguiente nodo, debe tener un solo argumento (no debe tener más de uno) y debe ser un comando de redirección de salida (fileout == 1).
- **Tercer comando**: El tercer comando debe ser "ls".

Si todas estas condiciones se cumplen, se procede a ejecutar un caso especial.

Se llama a ex_route para ejecutar el tercer comando, que se espera que sea ls. La función ex_route probablemente maneja la ejecución de comandos en la shell.

Se lee la primera línea de la entrada estándar usando get_next_line(0), que obtiene una línea de la entrada (probablemente de stdin).

Luego, se entra en un bucle que lee hasta dos líneas. En cada iteración, se libera la memoria de line y se lee una nueva línea. La variable j actúa como contador para garantizar que solo se lean dos líneas.

Después de leer las dos líneas, se libera la memoria de line y la función retorna EXIT_SUCCESS, indicando que la ejecución fue exitosa.

Si las condiciones iniciales no se cumplen, la función retorna EXIT_FAILURE, lo que indica que no se realizó ninguna acción.

La función ex_builts se encarga de ejecutar comandos internos ("built-ins") en una shell personalizada. Estos comandos son específicos de la shell y no requieren ejecutar un programa externo.

Se verifica si n->all_cmd es NULL o vacío. Si es así, significa que no hay ningún comando a ejecutar, y la función retorna EXIT_FAILURE

La función compara el primer comando de n->all_cmd con los nombres de varios comandos built-in, y según el comando, llama a la función correspondiente. Aquí están los casos:

Si el primer comando es cd, se llama a la función ex_cd, que maneja el cambio de directorio.

Si el comando es echo, se llama a la función ex_echo, que maneja la impresión de texto en la salida estándar.

Si el comando es env, se llama a la función ex_envp, que imprime las variables de entorno.

Si el comando es pwd, se llama a la función ex_pwd, que imprime el directorio de trabajo actual.

Si el comando es unset, se llama a la función ex_unset, que elimina una variable de entorno especificada en n->all_cmd[1].

Si el comando es export, se llama a la función ex_export, que agrega o modifica una variable de entorno.

Si el comando es exit, se llama a la función ex_exit, que termina la ejecución de la shell.

Si el comando no coincide con ninguno de los anteriores, la función retorna EXIT_FAILURE, indicando que el comando no es un built-in.

Si el comando es ejecutado correctamente, la función retorna EXIT_SUCCESS.

La función `ex_routepipes` maneja la ejecución de comandos que están conectados mediante pipes (`|`) en una shell personalizada.

Se declara una variable `pid` para almacenar el identificador de proceso que se obtiene al usar `fork`.

`pipe(data->fd)`: Se crea un pipe que conectará la salida de un comando con la entrada del siguiente. `data->fd[0]` es el extremo de lectura y `data->fd[1]` es el extremo de escritura.

Si `pipe` falla, se llama a `error_msg` para manejar el error y posiblemente terminar el programa.

`fork`: Se crea un nuevo proceso.

- En el proceso hijo, `pid` será igual a `0`.
- En el proceso padre, `pid` será mayor que `0`.

Si `fork` falla se cierran los extremos del pipe y se llama a `error_msg` para manejar el error.

Caso del proceso hijo (`pid == 0`):

- Si hay otro comando después del actual (`tmp->next != NULL`), se llama a la función `handle_child_pipe`, que configura el pipe y ejecuta el comando en el proceso hijo.
- Si es el último comando en la cadena de pipes, se llama a `process_final_pipe` para manejar la ejecución final.

Si hay otro comando después del actual (`tmp->next`) y no tiene un archivo de entrada (`filein == 0`), se asigna `data->fd[0]` como su entrada estándar.

Si no hay más comandos, se cierra `data->fd[0]` porque ya no se necesita.

El proceso padre cierra el extremo de escritura del pipe porque ya no lo usará.

El proceso padre espera a que el proceso hijo termine antes de continuar.

Se llama a `check_cargs` para realizar verificaciones adicionales en los argumentos o el estado actual del comando y los datos.

La función `handle_child_pipe` se encarga de configurar el entorno de un proceso hijo en una shell personalizada para manejar la entrada y salida de datos, especialmente en el contexto de pipes. También establece señales predeterminadas y ejecuta el comando correspondiente.

Se restauran las acciones predeterminadas de las señales:

- **`SIGINT` (Ctrl+C)**: Interrumpe el proceso.
- **`SIGQUIT` (Ctrl+)**: Genera un volcado de núcleo y termina el proceso.

Esto es importante porque el proceso hijo no debe usar los manejadores personalizados del proceso padre.

Si el proceso está utilizando un archivo temporal creado por un heredoc (`flag_hered == 1`):

1. Se redirige el archivo heredoc (`node->filein`) al **entrada estándar** del proceso (`STDIN_FILENO`) usando `dup2`. Si falla, se lanza un mensaje de error.
2. Se elimina el archivo temporal (`here_doc.tmp`) porque ya no es necesario.
3. Se cierra el archivo heredoc.

Si el nodo tiene un archivo de entrada (`filein`) diferente al estándar:

1. Se redirige el archivo de entrada a la **entrada estándar** del proceso usando `dup2`. Si falla, se lanza un mensaje de error.
2. Se cierra el archivo una vez redirigido, ya que no se necesita mantener abierto.

En el proceso hijo, no se necesita leer del pipe. Se cierra el extremo de lectura (`d->fd[0]`).

Se llama a la función `handle_child_command` para ejecutar el comando correspondiente en el nodo.

`handle_child_command` probablemente redirige la salida del proceso hijo al extremo de escritura del pipe y ejecuta el comando usando funciones como `execve`.

La función `handle_child_command` gestiona la configuración de la salida estándar de un proceso hijo y ejecuta un comando, ya sea un comando "builtin" (interno de la shell) o un comando externo usando `execve`

Caso 1: No hay redirección específica, usar pipe

Si el nodo actual no tiene un archivo de salida específico (`fileout == 1`):

- Redirige la salida estándar del proceso hijo (`STDOUT_FILENO`) al extremo de escritura del pipe (`d->fd[1]`).
- Si `dup2` falla, lanza un mensaje de error.

Caso 2: Redirección hacia un archivo específico

Si el nodo tiene un archivo de salida definido (`fileout != 1`):

- Redirige la salida estándar hacia ese archivo.
- Cierra el archivo después de redirigirlo, ya que no es necesario mantenerlo abierto.

Una vez que la salida estándar se ha redirigido correctamente, se cierra el extremo de escritura del pipe, ya que no se usará más.

Se verifica si el comando actual es un "builtin" comparando `n->route` con `"b\0"`. Si lo es:

- Si no hay un archivo de salida definido (`fileout == 1`) y el siguiente nodo del pipe no tiene un archivo de entrada (`filein == 0`), se redirige el extremo de lectura del pipe (`d->fd[0]`) al archivo de salida del siguiente nodo.
- Se llama a la función `ex_builts`, que ejecuta el comando builtin correspondiente.
- Luego, el proceso hijo termina con `exit(1)`, ya que los comandos "builtin" no usan `execve`.

Si el comando no es un "builtin", se ejecuta con `execve`:

- `n->route`: Ruta al ejecutable.
- `n->all_cmd`: Array de argumentos, donde el primer elemento es el nombre del comando.
- `envp`: Array de variables de entorno.

Si `execve` tiene éxito, reemplaza el proceso hijo con el nuevo programa y no retorna. Si falla, el proceso continúa y termina (aunque esto no está manejado explícitamente en esta función).

La función `process_final_pipe` gestiona la configuración de la entrada estándar del proceso para el último comando en una secuencia de pipes. También llama a una función adicional para manejar la salida del pipe y ejecutar el comando correspondiente.

Si el nodo tiene un archivo de entrada (`filein != 0`) y la entrada proviene de un "here document" (`flag_hered == 1`):

- Redirige la entrada estándar (`STDIN_FILENO`) al archivo correspondiente (`filein`) usando `dup2`.
- Si `dup2` falla, se muestra un mensaje de error.

Si hay un archivo de entrada (`filein != 0`) pero no es un "here document" (`flag_hered == 0`):

- Redirige la entrada estándar al archivo.

Si el extremo de escritura del pipe (`fd[1]`) es la salida estándar (`STDOUT_FILENO`), significa que este nodo no tiene un archivo de entrada definido.

- Redirige la entrada estándar al extremo de lectura del pipe (`fd[0]`).
- Si `dup2` falla, muestra un mensaje de error.

Si no hay un archivo de entrada (`filein == 0`), intenta redirigir la entrada estándar.

Después de redirigir la entrada estándar, cierra el archivo de entrada para liberar recursos.

Llama a `process_pipe_out`, una función que probablemente configura la salida estándar y ejecuta el comando final.

La función `process_pipe_out` se encarga de configurar la salida estándar para un comando y luego ejecutarlo.

Condición: Si el archivo de salida del nodo (`fileout`) no es la salida estándar predeterminada (`STDOUT_FILENO`):

1. Redirige la salida estándar (`STDOUT_FILENO`) al archivo de salida (`fileout`) utilizando `dup2`.
    - Esto asegura que cualquier salida del comando se escriba en el archivo especificado.
2. Si `dup2` falla:
    - Muestra un mensaje de error: `"Last command wrt error\n"`.
3. Cierra el archivo de salida (`fileout`) para liberar recursos.

Usa `execve` para ejecutar el comando:

1. **`node->route`**: Ruta del ejecutable del comando.
2. **`node->all_cmd`**: Argumentos del comando (incluye el nombre del ejecutable como el primer argumento).
3. **`envp`**: Variables de entorno pasadas al nuevo proceso.

Si `execve` tiene éxito, reemplaza el proceso actual con el nuevo, y no retorna.

Si falla, retorna un error y se guarda el estado en `data->status`.

Esta función generalmente se llama después de haber configurado correctamente la entrada estándar y otros recursos necesarios. Es la última etapa para ejecutar un comando. Se asegura de que:

- La salida del comando se redirija adecuadamente a un archivo, si es necesario.
- El comando se ejecute con las configuraciones de entorno proporcionadas.

Redirección de salida:

    - Si el nodo tiene un archivo de salida diferente de la salida estándar:
        - Redirige la salida estándar al archivo.
        - Libera el descriptor del archivo de salida.
- **Ejecución del comando**:
    - Ejecuta el comando con la ruta, argumentos, y entorno especificados.
    - El estado del comando se almacena en `data->status`.

Esta función asegura que los resultados del comando se dirijan al lugar correcto antes de ejecutar el proceso. Si hay errores en la redirección o ejecución, los maneja de manera adecuada.

La función ex_route se encarga de gestionar la ejecución de un comando en un proceso hijo. Dependiendo de las configuraciones de entrada y salida del nodo (filein y fileout), la función selecciona entre ejecutar un comando interno (built-in), un comando por defecto, o manejar la ejecución en un proceso hijo.

Comprobación: Llama a ex_builts, que maneja la ejecución de comandos internos como cd, echo, export, etc.

- Si el comando es un built-in y se ejecuta correctamente, la función regresa (return) y no continúa.
- Esto evita la necesidad de crear un proceso hijo para comandos internos.

Condición: Si la entrada del comando es la estándar (filein == 0) y la salida es la estándar (fileout == 1):

- Llama a default_execute, que ejecuta el comando sin redirecciones adicionales.
- Este camino es más directo y no implica procesos hijos complejos.

Escenario: Si el comando tiene configuraciones personalizadas para entrada o salida (redirecciones), se maneja en un proceso hijo:

1. **Obtener nodo actual**:
    - Se accede al nodo actual desde la lista data->nodes.
2. **Crear un proceso hijo**:
    - Usa fork para dividir el proceso actual:
        - El proceso hijo manejará la ejecución del comando.
        - El proceso padre esperará su finalización.

Si fork() falla:

- Configura el estado de error (data->status = 1) y muestra un mensaje de error.

En el proceso hijo:

- Llama a la función route_child_ex, que configura las redirecciones necesarias (entrada/salida) y ejecuta el comando mediante execve.

En el proceso padre:

- Usa waitpid para esperar a que el proceso hijo termine su ejecución.
- Esto asegura que el flujo de comandos se mantenga ordenado.

Llama a check_cargs para realizar una verificación de los argumentos del comando.

Esto puede incluir validaciones adicionales o actualizaciones del estado de data.

La función default_execute se utiliza para ejecutar comandos que no tienen redirecciones de entrada ni salida (es decir, cuando filein == 0 y fileout == 1)

Usa fork para crear un proceso hijo:

- Si fork falla (pid == -1):
    - Se establece un estado de error en data->status (1 indica error).
    - Llama a error_msg para imprimir un mensaje indicando que ocurrió un error al intentar crear el proceso.
- Si tiene éxito, el código continúa separadamente en el proceso padre e hijo.

En el proceso hijo (pid == 0):

- Llama a execve para ejecutar el comando:
    - node->route: Ruta del ejecutable que se quiere ejecutar.
    - node->all_cmd: Array de argumentos del comando, donde el primer elemento es el comando mismo.
    - envp: Array de variables de entorno heredado del proceso padre.
- Si execve tiene éxito:
    - Sustituye el proceso hijo con el nuevo ejecutable, y el código del hijo no continúa más allá de este punto.
- Si execve falla:
    - El proceso hijo termina y devuelve un estado de error en data->status.

En el proceso padre:

- Llama a waitpid para esperar a que el proceso hijo termine su ejecución.
- Esto asegura que el comando se complete antes de que el proceso padre continúe con otras tareas.

Llama a check_cargs para realizar verificaciones adicionales en los argumentos del comando.

Esto puede incluir actualizaciones en data o validaciones específicas del comando ejecutado.

1. **Crea un proceso hijo**:
    - Usa fork para dividir la ejecución entre el padre y el hijo.
2. **Ejecuta el comando en el hijo**:
    - Sustituye el proceso hijo con el nuevo comando usando execve.
    - Si execve falla, registra el error.
3. **Espera en el padre**:
    - El proceso padre espera a que el hijo termine para continuar.

check_cargs valida o actualiza información según los resultados del comando.

La función route_child_ex configura un proceso hijo para que ejecute un comando con redirecciones de entrada y salida según sea necesario.

Restablece las señales:

- SIGINT (Ctrl+C) y SIGQUIT (Ctrl+) se configuran a su comportamiento por defecto (SIG_DFL) en el proceso hijo.
- Esto asegura que el proceso hijo maneje las señales de manera estándar, ya que el padre podría haber configurado un manejo personalizado.

Si data->flag_hered es 1:

- **Redirige la entrada estándar** (STDIN_FILENO) desde el archivo temporal creado por el here-document, utilizando dup2.
- Si falla, imprime un mensaje de error y termina el proceso.
- Elimina el archivo temporal here_doc.tmp con unlink después de redirigir la entrada.

Si node->filein no es el descriptor estándar 0 (entrada estándar):

- **Redirige la entrada estándar** desde el archivo especificado en node->filein utilizando dup2.
- Si falla, imprime un mensaje de error y termina el proceso.

Si node->fileout no es el descriptor estándar 1 (salida estándar):

- **Redirige la salida estándar** (STDOUT_FILENO) al archivo especificado en node->fileout utilizando dup2.
- Si falla, imprime un mensaje de error y termina el proceso.

Ejecuta el comando utilizando execve:

- node->route: Ruta del ejecutable.
- node->all_cmd: Array con el nombre del comando y sus argumentos.
- envp: Array con las variables de entorno.

Si execve tiene éxito:

- El proceso hijo es reemplazado completamente por el programa especificado.

Si execve falla:

- No retorna al programa en ejecución; el sistema operativo termina el proceso hijo con un código de error.

La función check_cargs evalúa los argumentos de un comando para determinar si son válidos según ciertas reglas específicas.

file_inf:

- Estructura usada para almacenar información de un archivo con la función stat.

i y j:

- Variables de índice para recorrer los argumentos y caracteres del comando.

Objetivo:

- Saltar los argumentos que contienen un guion ('-'), presumiblemente porque son opciones o flags (por ejemplo, -l, -a).

Lógica:

- Avanza i mientras haya un argumento en node->all_cmd[i] y este contenga un guion ('-').

Contexto:

- Comprueba el último argumento que no es un flag (node->all_cmd[i - 1]).

Proceso:

1. Si node->all_cmd[i - 1] no es NULL, recorre cada carácter de ese argumento.
2. Si encuentra un dígito (ft_isdigit), establece data->status en 1.
- Esto sugiere que los números en ciertos argumentos son problemáticos para el programa.

Objetivo:

- Comprueba si el argumento actual (node->all_cmd[i]) es un archivo válido.
- Si stat devuelve 0:
    - El archivo existe, y no hay errores en el argumento.
- Si node->all_cmd[i] es NULL:
    - No hay más argumentos, y no se modifica data->status.
- Si se cumple alguna de estas condiciones, la función termina.

Acción:

- Si stat falla (el archivo no existe) y hay un argumento, se establece data->status en 1, indicando un error.

ERRORES

Las dos funciones error_msg_pipe y error_msg son prácticamente idénticas y están diseñadas para manejar errores en el programa, pero con un pequeño detalle que las diferencia

ERROR_MSG_PIPE

unlink("here_doc.tmp") (Línea 1):

- Elimina el archivo temporal "here_doc.tmp" (probablemente usado en un proceso de "heredoc").
- Se asegura de que si hubo un error relacionado con la creación de este archivo o su uso, el archivo temporal sea eliminado antes de que el programa termine.

data->status = 1; (Línea 2):

- Establece el campo status de la estructura data en 1, lo que generalmente indica un error. Este valor podría ser utilizado en otro lugar del programa para verificar el estado de la ejecución.

perror(error); (Línea 3):

- Muestra un mensaje de error específico, precedido del mensaje "error". perror imprime el valor de la cadena error seguida del mensaje de error del sistema (proporcionado por errno).
- El mensaje impreso es el error relacionado con la llamada de sistema que falló (por ejemplo, "No such file or directory" si hay un problema con un archivo).

exit(EXIT_FAILURE); (Línea 4):

- Termina el programa inmediatamente con el código de salida EXIT_FAILURE (generalmente 1), indicando que la ejecución ha fallado.

ERROR_MSG

unlink("here_doc.tmp") (Línea 1):

    - Al igual que en error_msg_pipe, elimina el archivo "here_doc.tmp", que parece estar relacionado con un proceso de "here doc". Esto asegura que cualquier archivo temporal creado para manejar la entrada heredada se elimine al producirse un error.
- **data->status = 1; (Línea 2)**:
    - Establece el estado de la ejecución en 1, indicando un error, al igual que en la función anterior.
- **perror(error); (Línea 3)**:
    - Imprime el mensaje de error dado, seguido de la descripción del error proporcionado por errno.
- **exit(EXIT_FAILURE); (Línea 4)**:
    - Termina la ejecución del programa con el código de salida EXIT_FAILURE, indicando un fallo en la ejecución.

UNLINK

La función unlink("here_doc.tmp"); elimina el archivo temporal llamado "here_doc.tmp" del sistema de archivos.

Explicación detallada:

- **unlink** es una llamada de sistema que elimina un archivo de manera permanente.
- Cuando se ejecuta unlink("here_doc.tmp");, el archivo here_doc.tmp se borra del sistema de archivos. Sin embargo, si el archivo está abierto por algún proceso (es decir, si está siendo usado por un descriptor de archivo), el archivo no se eliminará inmediatamente del sistema de archivos, pero dejará de ser accesible para otros procesos. En cuanto el archivo ya no esté siendo usado, será completamente eliminado.

Contexto:

En el contexto del código, el archivo "here_doc.tmp" parece ser un archivo temporal utilizado, posiblemente, para almacenar datos de una entrada heredada (un "here document") durante un proceso en el que el programa lee múltiples líneas de entrada antes de ejecutar un comando.

Usar unlink en este contexto sugiere que el archivo temporal se ha usado para algún propósito durante la ejecución del programa (por ejemplo, almacenar datos para un here_doc), y se está eliminando una vez que ya no se necesita, probablemente como una limpieza al final de un proceso o después de un error para evitar dejar archivos residuales en el sistema.

Comportamiento:

- Si el archivo existe, se elimina.
- Si el archivo no existe, no se hace nada (no genera un error).

En resumen, unlink("here_doc.tmp"); es una forma de asegurarse de que un archivo temporal llamado "here_doc.tmp" sea eliminado del sistema después de su uso.

FREES

La función free_split se encarga de liberar la memoria de un arreglo de cadenas de caracteres (un arreglo de punteros a char), también conocido como un "split" en muchos lenguajes de programación, donde cada elemento del arreglo es una cadena de caracteres.

Explicación paso a paso:

1. **int i = 0;**:
   - Se inicializa la variable i como un índice para iterar a través del arreglo de cadenas (tmp).
2. **while (tmp != NULL && tmp[i] != NULL)**:
   - Este while recorre cada uno de los elementos del arreglo tmp. La condición asegura que:
     - tmp no sea NULL (es decir, el arreglo de cadenas existe).
     - tmp[i] no sea NULL, lo que indica que aún hay cadenas que liberar.
   - Dentro del bucle, cada cadena de caracteres es liberada.
3. **free(tmp[i]);**:
   - La función free(tmp[i]) libera la memoria ocupada por cada cadena individual en el arreglo.
   - Después de liberar la memoria de tmp[i], se asegura que el puntero se ponga a NULL para evitar acceder a memoria liberada (lo que podría causar un comportamiento indeterminado o "segfault").
4. **tmp[i] = NULL;**:
   - Después de liberar el puntero de tmp[i], se asigna NULL para prevenir el acceso a memoria previamente liberada (por ejemplo, evitar "dangling pointers").
5. **i++;**:
   - Se incrementa el índice i para avanzar al siguiente elemento del arreglo y repetir el proceso de liberación.
6. **if (tmp != NULL)**:
   - Después de liberar cada cadena en el arreglo, se verifica si el arreglo tmp no es NULL (es decir, si el arreglo existe). Si es así, se procede a liberar la memoria del propio arreglo de punteros.
7. **free(tmp);**:
   - Aquí se libera la memoria del arreglo de punteros tmp en sí (la memoria que fue reservada para almacenar las direcciones de las cadenas).
8. **tmp = NULL;**:
   - Finalmente, se asigna NULL al puntero tmp para asegurarse de que no haya referencias a memoria que ya ha sido liberada, evitando que el puntero apunte a una dirección inválida.
9. **tmp = NULL;**:
   - Esta línea adicional de tmp = NULL; no tiene un efecto directo en la ejecución, ya que el puntero tmp ya fue asignado a NULL en el paso anterior.

La función free_cd se encarga de liberar la memoria relacionada con ciertos punteros dentro de la estructura t_data. A continuación te explico cada línea de la función paso a paso:

Explicación paso a paso:

1. **free(data->aux);**:
   - Aquí se está liberando la memoria que fue previamente asignada a data->aux. El puntero aux puede ser un arreglo dinámicamente asignado o una estructura, dependiendo de cómo se haya utilizado en el programa.
   - La función free() se encarga de liberar la memoria reservada para este puntero.
2. **free(data->error_cd);**:
   - Similar a la línea anterior, esta línea libera la memoria reservada para el puntero data->error_cd. Este puntero podría estar apuntando a una cadena de caracteres u otro tipo de datos que necesite ser liberado.
3. **data->status = 1;**:
   - Se establece el valor de data->status a 1, lo que probablemente indica un error o un estado específico dentro del programa.
   - Aunque no libera memoria, esta línea podría estar indicando un código de error para reflejar que algo salió mal en el proceso relacionado con cd (probablemente un cambio de directorio en un shell o alguna operación similar).

La función free_node se encarga de liberar toda la memoria asociada con los nodos de una lista enlazada, asegurándose de liberar tanto los elementos dentro de cada nodo como el propio nodo.

if (!*lst) return ;:

- Primero se verifica si el puntero lst (puntero a la cabeza de la lista) es NULL. Si la lista está vacía (es decir, *lst es NULL), la función termina inmediatamente sin hacer nada. Esto evita errores si intentamos liberar una lista vacía.

while (*lst):

- Inicia un bucle que recorre todos los nodos de la lista. El bucle continuará mientras *lst (el puntero a la cabeza de la lista) no sea NULL.

tmp = (*lst)->next;:

- Almacena el siguiente nodo de la lista en tmp. Esto es necesario porque en el siguiente paso vamos a liberar el nodo actual (*lst), y al hacerlo perderíamos la referencia al siguiente nodo si no la guardamos primero.

i = -1;:

- Inicializa una variable i en -1, que se utilizará como índice para recorrer el arreglo all_cmd dentro del nodo.

if (((t_parser *)((*lst)->content))->all_cmd):

- Verifica si el contenido del nodo (almacenado en (*lst)->content) es un puntero a una estructura t_parser y si esa estructura tiene el puntero all_cmd que no es NULL. Si all_cmd es NULL, significa que no hay comandos que liberar, por lo que este bloque no se ejecuta.

while (((t_parser *)((*lst)->content))->all_cmd[++i]):

- Si all_cmd contiene un puntero válido, se recorre cada cadena de caracteres dentro de all_cmd. La variable i se incrementa para acceder a cada elemento de all_cmd uno a uno.

free(((t_parser *)((*lst)->content))->all_cmd[i]);:

- Se libera cada cadena de caracteres en el arreglo all_cmd. Cada elemento dentro de all_cmd es una cadena dinámica que debe ser liberada para evitar fugas de memoria.

((t_parser *)((*lst)->content))->all_cmd[i] = NULL;:

- Después de liberar cada cadena, se establece el puntero correspondiente a NULL, lo que ayuda a evitar referencias erróneas a memoria liberada.

free_command(lst);:

- Se llama a una función llamada free_command, que probablemente libere otros recursos asociados con el comando o la estructura dentro del nodo. Esta función no se muestra aquí, pero es probable que también haga limpieza de otros datos dentro de la estructura t_parser.

free((*lst));:

- Después de liberar el contenido del nodo, se libera la memoria del propio nodo (*lst), que es un elemento de la lista enlazada.

***lst = tmp;**:

- Avanza el puntero lst al siguiente nodo (tmp), de manera que el siguiente ciclo de la lista opere sobre el siguiente nodo.

***lst = NULL;**:

- Después de haber liberado todos los nodos, se establece el puntero lst a NULL, lo que asegura que la lista se marca como vacía.

La función free_command se encarga de liberar los recursos asociados a un nodo específico en una lista enlazada, más específicamente a la estructura t_parser que está almacenada en el campo content de ese nodo.

Explicación paso a paso:

1. **free(((t_parser *)((*lst)->content))->all_cmd);**:
   - Se accede al contenido del nodo ((*lst)->content), que es un puntero a una estructura t_parser. Dentro de esa estructura, se accede al puntero all_cmd, que es un arreglo de cadenas.
   - Luego, se libera la memoria ocupada por all_cmd, que contiene las cadenas de comandos.
2. **((t_parser *)((*lst)->content))->all_cmd = NULL;**:
   - Después de liberar la memoria de all_cmd, se establece el puntero all_cmd a NULL para evitar referencias a memoria que ya ha sido liberada. Esto es una buena práctica para prevenir errores si accidentalmente se intenta acceder a esta memoria más tarde.
3. **free(((t_parser *)((*lst)->content))->route);**:
   - Se libera la memoria ocupada por el puntero route de la estructura t_parser. Este puntero probablemente contiene la ruta del comando o algún otro dato relevante.
4. **((t_parser *)((*lst)->content))->route = NULL;**:
   - Después de liberar la memoria de route, el puntero route se establece en NULL por las mismas razones mencionadas anteriormente (para evitar referencias a memoria ya liberada).
5. **if (((t_parser *)((*lst)->content))->filein != 0)**:
   - Se verifica si filein (un campo de la estructura t_parser) es distinto de 0. El valor 0 generalmente se usa para indicar que no se ha abierto ningún archivo de entrada.
   - Si filein no es 0, esto implica que se ha abierto un archivo para lectura, por lo que se procede a cerrarlo.
6. **close(((t_parser *)((*lst)->content))->filein);**:
   - Si el archivo de entrada (filein) ha sido abierto, se cierra utilizando la función close.
7. **if (((t_parser *)((*lst)->content))->fileout != 1)**:
   - Se verifica si fileout (un campo de la estructura t_parser) es distinto de 1. El valor 1 generalmente se usa para indicar que no se ha abierto ningún archivo de salida, o que se está escribiendo en la salida estándar.
   - Si fileout no es 1, se procede a cerrar el archivo de salida.
8. **close(((t_parser *)((*lst)->content))->fileout);**:
   - Si el archivo de salida (fileout) ha sido abierto, se cierra utilizando la función close.
9. **free(((t_parser *)((*lst)->content)));**:
   - Después de haber cerrado los archivos y liberado los punteros dentro de la estructura t_parser, se libera la memoria ocupada por la estructura t_parser misma.

10. **(*lst)->content = NULL;**:
    - Finalmente, se establece (*lst)->content a NULL para evitar que el puntero apunte a memoria que ha sido liberada.

La función free_t_parser se encarga de liberar todos los recursos asociados a un objeto de tipo t_parser. A continuación, te explico paso a paso lo que hace cada parte de la función:

Explicación paso a paso:

1. **if (node == NULL) return;**:
    - La función comienza verificando si el puntero node es NULL. Si es NULL, simplemente retorna sin hacer nada, ya que no hay nada que liberar. Esto previene errores si se pasa un puntero nulo a la función.
2. **if (node->all_cmd != NULL)**:
    - Se verifica si el puntero all_cmd dentro de la estructura t_parser no es NULL. all_cmd probablemente sea un arreglo de cadenas de caracteres que contiene los comandos.
3. **free_split(node->all_cmd);**:
    - Si all_cmd no es NULL, se llama a la función free_split para liberar la memoria ocupada por all_cmd. Es probable que free_split libere cada cadena dentro del arreglo y luego libere el arreglo mismo.
4. **node->all_cmd = NULL;**:
    - Después de liberar la memoria de all_cmd, se establece node->all_cmd a NULL. Esto evita referencias a memoria que ya ha sido liberada.
5. **if (node->route != NULL) free(node->route);**:
    - Se verifica si route (un puntero dentro de t_parser que probablemente contiene la ruta de un comando) no es NULL.
    - Si no es NULL, se libera la memoria ocupada por route con la función free.
6. **if (node->filein != 0) close(node->filein);**:
    - Se verifica si filein (un archivo de entrada) es diferente de 0, lo que indica que el archivo ha sido abierto.
    - Si es diferente de 0, se cierra el archivo de entrada con close. El valor 0 generalmente representa que no hay un archivo de entrada o que se está utilizando la entrada estándar (stdin), por lo que no se cierra en ese caso.
7. **if (node->fileout != 1) close(node->fileout);**:
    - Se verifica si fileout (un archivo de salida) es diferente de 1, lo que indica que el archivo ha sido abierto.
    - Si es diferente de 1, se cierra el archivo de salida con close. El valor 1 generalmente representa que no hay un archivo de salida o que se está utilizando la salida estándar (stdout), por lo que no se cierra en ese caso.
8. **free(node);**:
    - Finalmente, se libera la memoria ocupada por la estructura t_parser misma, que fue originalmente asignada dinámicamente. Esto elimina el objeto node.

FREE_ALL

La función free_all está diseñada para liberar todos los recursos asociados con la estructura t_data. Esta estructura parece contener múltiples punteros a memoria dinámica que deben liberarse cuando ya no sean necesarios para evitar fugas de memoria. A continuación, te explico paso a paso lo que hace cada parte de la función:

Explicación paso a paso:

1. **if (data->path != NULL)**:
   - Se verifica si el puntero path dentro de la estructura t_data no es NULL. Si no es NULL, significa que se ha asignado memoria para almacenar la ruta de los comandos o algún otro dato relacionado.
2. **free_split(data->path);**:
   - Si path no es NULL, se llama a la función free_split para liberar la memoria ocupada por path. Probablemente, path sea un arreglo de cadenas (como un arreglo de rutas), por lo que free_split liberará cada cadena dentro del arreglo y luego liberará el arreglo mismo.
3. **data->path = NULL;**:
   - Después de liberar la memoria de path, se asigna NULL a data->path para evitar referencias a memoria que ya ha sido liberada.
4. **if (data->prompt != NULL)**:
   - Se verifica si el puntero prompt (probablemente una cadena que almacena el prompt de la shell) no es NULL.
5. **free(data->prompt);**:
   - Si prompt no es NULL, se libera la memoria que ocupa con la función free.
6. **data->prompt = NULL;**:
   - Después de liberar la memoria de prompt, se asigna NULL a data->prompt para evitar referencias a memoria liberada.
7. **if (data->envp != NULL)**:
   - Se verifica si envp (probablemente un puntero a un arreglo de cadenas que contiene las variables de entorno) no es NULL.
8. **free_envp(&data->envp);**:
   - Si envp no es NULL, se llama a la función free_envp para liberar la memoria asociada a las variables de entorno. free_envp probablemente recorra y libere cada cadena en el arreglo de variables de entorno.
9. **if (data->nodes != NULL)**:
   - Se verifica si nodes (probablemente una lista de nodos, que podrían representar los comandos o procesos en la shell) no es NULL.
10. **free_node(&data->nodes);**:
    - Si nodes no es NULL, se llama a la función free_node para liberar la memoria asociada a los nodos. free_node probablemente recorra la lista y libere cada nodo y sus recursos internos.
11. **data->nodes = NULL;**:
    - Después de liberar los nodos, se asigna NULL a data->nodes para evitar referencias a memoria que ya ha sido liberada.

12. **if (data->cmd != NULL)**:
    - Se verifica si cmd (probablemente un arreglo de comandos) no es NULL.
13. **free_split(data->cmd);**:
    - Si cmd no es NULL, se llama a la función free_split para liberar la memoria ocupada por cmd. Esta función probablemente liberará cada cadena dentro del arreglo y luego liberará el arreglo mismo.
14. **data->cmd = NULL;**:
    - Después de liberar la memoria de cmd, se asigna NULL a data->cmd para evitar referencias a memoria liberada.
15. **free(data);**:
    - Finalmente, se libera la memoria ocupada por la estructura t_data misma. Esto elimina la estructura data y todos los punteros que apuntan a memoria dinámica dentro de ella, que ya han sido liberados anteriormente.

La función `free_envp` se encarga de liberar la memoria asociada a una lista de variables de entorno (presumiblemente de tipo `t_envp`). Aquí te explico paso a paso lo que hace:

Explicación paso a paso:

1. **`if (!*lst)`**:
   - Se verifica si el puntero `*lst` (la lista de variables de entorno) es `NULL`. Si es `NULL`, significa que la lista está vacía y no se necesita hacer nada. En este caso, la función retorna inmediatamente.
2. **`while (*lst)`**:
   - Si la lista no está vacía, se entra en un bucle `while` que recorre toda la lista. La condición del `while` asegura que el bucle continuará hasta que `*lst` sea `NULL`, lo que indica que hemos llegado al final de la lista.
3. **`tmp = (*lst)->next;`**:
   - Se almacena el puntero al siguiente nodo de la lista en la variable `tmp`. Esto es necesario porque se va a modificar `*lst` en el siguiente paso, y necesitamos tener acceso al siguiente nodo de la lista para continuar el recorrido.
4. **`free_all_envp(*lst);`**:
   - Se llama a la función `free_all_envp` pasando el nodo actual (`*lst`). Esta función se encarga de liberar toda la memoria asociada a ese nodo, lo que incluye cualquier cadena o estructura interna que el nodo pueda contener (como los valores de las variables de entorno).
5. **`*lst = tmp;`**:
   - Se actualiza el puntero `*lst` para que apunte al siguiente nodo en la lista (`tmp`). Esto avanza al siguiente nodo de la lista, preparándose para liberarlo en la siguiente iteración del bucle.
6. **`*lst = NULL;`**:
   - Una vez que el bucle ha recorrido toda la lista y todos los nodos han sido liberados, se asigna `NULL` a `*lst` para asegurarse de que el puntero de la lista se apunte a `NULL`, lo que indica que la lista está completamente vacía y que ya no hay referencias a memoria liberada.

La función free_all_envp se encarga de liberar la memoria asociada a un solo nodo de la lista de variables de entorno (t_envp). A continuación te explico paso a paso lo que hace:

Explicación paso a paso:

1. **if (!lst)**:
   - Se verifica si el nodo lst es NULL. Si es NULL, la función retorna inmediatamente, ya que no hay nada que liberar. Este paso previene intentos de liberar memoria en un puntero nulo.
2. **free(lst->name);**:
   - Se libera la memoria asociada al campo name del nodo lst. El campo name probablemente contiene el nombre de la variable de entorno (por ejemplo, "PATH"). Si este campo ha sido asignado dinámicamente (con malloc o similar), debe ser liberado.
3. **lst->ind = 0;**:
   - Se pone a cero el valor de ind. Este campo parece ser un índice o un indicador relacionado con el nodo, y ponerlo a cero asegura que no haya valores residuales en esta variable después de liberar la memoria asociada al nodo.
4. **free(lst->content);**:
   - Se libera la memoria asociada al campo content del nodo lst. El campo content probablemente contiene el valor o contenido de la variable de entorno (por ejemplo, "usr/bin" en el caso de una variable PATH). De igual manera que con name, se debe liberar la memoria asignada dinámicamente.
5. **free(lst);**:
   - Finalmente, se libera la memoria asociada al propio nodo lst. Este paso libera la memoria que se asignó para la estructura t_envp que representa el nodo en la lista.
6. **lst = NULL;**:
   - Este paso es redundante, ya que lst es un puntero local dentro de la función. Modificar lst a NULL no afectará el puntero original que se pasó a la función. Sin embargo, es una práctica común para evitar que se sigan usando punteros no válidos después de liberar la memoria.

BUILTINS

EX_CD

La función ex_cd maneja la ejecución del comando cd en una shell, permitiendo cambiar el directorio actual del programa.

char currdir[500];:

- Se declara un arreglo currdir de 500 caracteres que se utilizará para almacenar el directorio actual antes de cambiarlo. Esto se hace para poder restaurar el directorio anterior si es necesario.

if (ft_strncmp(str[0], "cd\0", 3) == EXIT_SUCCESS && !str[1]):

- Aquí se verifica si el primer elemento de str (el comando que se ejecuta) es cd y si no hay argumentos adicionales (es decir, str[1] es NULL).
- Si ambas condiciones son verdaderas, se llama a la función update_cd(data) que, presumiblemente, actualizará el directorio actual de alguna forma cuando no se pase un directorio específico (como en el caso de cd sin argumentos, que generalmente va al directorio home).

else if (!ft_strncmp(str[0], "cd\0", 3) && str[1]):

- Si el primer argumento es cd y también se ha proporcionado un directorio como argumento (str[1] no es NULL), entonces se ejecuta el siguiente bloque de código.

getcwd(currdir, 500);:

- La función getcwd obtiene el directorio de trabajo actual y lo guarda en la variable currdir. El valor de currdir se usará más adelante para poder restaurar el directorio anterior si es necesario.

flag = chdir(str[1]);:

- La función chdir intenta cambiar el directorio de trabajo a str[1], que es el directorio especificado por el usuario.
- El valor de retorno de chdir se almacena en flag. Si chdir tiene éxito, devuelve 0; si ocurre un error (por ejemplo, si el directorio no existe), devuelve -1.

if (flag == 0):

- Si chdir devuelve 0, significa que el cambio de directorio fue exitoso.
    - **up_prevdir(data, currdir);**: Aquí se llama a la función up_prevdir, que parece actualizar el directorio anterior. Le pasa el valor de currdir (el directorio que estaba activo antes de cambiar), lo que podría permitirle al sistema recordar el directorio anterior.

    - **up_currdir(data);**: Luego se llama a la función up_currdir, que parece actualizar el directorio actual a la nueva ubicación a la que se cambió.

else if (flag == -1):

- Si chdir devuelve -1, significa que hubo un error al intentar cambiar al directorio especificado en str[1] (por ejemplo, si el directorio no existe).
    - **data->aux = ft_strjoin("error cd", str[1]);**: Aquí se crea un mensaje de error concatenando la cadena "error cd" con el directorio que se intentó cambiar.
    - **if (!data->aux) return;**: Si la función ft_strjoin falla y no puede asignar memoria para data->aux, la función retorna sin hacer nada más.
    - **data->error_cd = ft_strjoin(data->aux, " no directoryu");**: Se genera un mensaje de error completo concatenando el mensaje anterior con la cadena " no directory".
    - **if (!data->error_cd) return;**: Si la función ft_strjoin falla nuevamente, la función retorna sin continuar.
    - **ft_putendl_fd(data->error_cd, 2);**: Si todo salió bien hasta aquí, se imprime el mensaje de error en la salida de error estándar (file descriptor 2).
    - **free_cd(data);**: Se llama a la función free_cd para liberar cualquier memoria dinámica asociada con los mensajes de error generados.

La función update_cd está diseñada para manejar el caso en que se ejecuta el comando cd sin argumentos, lo que generalmente significa que el usuario desea cambiar al directorio de inicio (home).

Explicación paso a paso:

1. **up_prevdir(data, NULL);**:
   - Aquí se llama a la función up_prevdir, pasándole el argumento NULL. Esto sugiere que la función up_prevdir puede estar actualizando o eliminando el valor del directorio anterior en algún tipo de estructura de datos. El uso de NULL indica que no se necesita un valor específico para este caso.
2. **home_path(data, "HOME\0");**:
   - Se llama a la función home_path, pasándole "HOME\0" como argumento.
   - Esta función probablemente está obteniendo la ruta del directorio de inicio del usuario, usando la variable de entorno HOME, que normalmente almacena la ruta del directorio home del usuario en sistemas Unix/Linux. La cadena "HOME\0" representa la variable de entorno que contiene la ruta del directorio home.
3. **chdir(data->aux);**:
   - La función chdir se llama con data->aux como argumento. Aquí, se asume que data->aux contiene la ruta del directorio home del usuario, que probablemente fue almacenada anteriormente en home_path.
   - chdir cambia el directorio de trabajo actual al que se encuentra en data->aux (el directorio de inicio del usuario).
4. **up_currdir(data);**:
   - Se llama a la función up_currdir, probablemente para actualizar el directorio actual en alguna estructura de datos o variable, reflejando el nuevo directorio de trabajo después de ejecutar chdir.
5. **data->status = 0;**:
   - Finalmente, la función establece data->status en 0, lo que generalmente indica que la operación fue exitosa. En muchas implementaciones de shells, un estado de salida de 0 indica que no hubo errores, mientras que cualquier valor distinto de 0 indica un error.

La función up_prevdir está diseñada para actualizar el valor de la variable de entorno OLDPWD, que generalmente se usa para almacenar el directorio anterior en un entorno de shell. Este tipo de funcionalidad es común en shells, donde se guarda la ruta del directorio anterior para poder volver a él cuando el usuario lo desee (por ejemplo, con el comando cd -).

t_envp *tmp;:

- Se declara un puntero tmp de tipo t_envp, que es una estructura que probablemente contiene las variables de entorno (nombre y valor). Este puntero se utilizará para iterar a través de una lista enlazada de variables de entorno.

tmp = data->envp;:

- Se inicializa tmp con el valor de data->envp. Se asume que data->envp es la cabeza de una lista enlazada de variables de entorno. tmp recorrerá esta lista para buscar la variable de entorno OLDPWD.

while (tmp):

- Se inicia un bucle while que recorre la lista enlazada de variables de entorno mientras tmp no sea NULL.

if (ft_strncmp(tmp->name, "ODLPWD", 6) == EXIT_SUCCESS):

- Dentro del bucle, se compara el nombre de la variable de entorno (tmp->name) con la cadena "ODLPWD", utilizando la función ft_strncmp. Esta función compara los primeros 6 caracteres de tmp->name con "ODLPWD". Si encuentra una coincidencia, significa que ha encontrado la variable de entorno OLDPWD.

free(tmp->content);:

- Si se encuentra la variable OLDPWD, se libera la memoria que contiene su valor anterior, es decir, tmp->content. Esto se hace para evitar fugas de memoria antes de asignar un nuevo valor.

if (!dir):

- Si el argumento dir es NULL, se obtiene el directorio actual usando la función getcwd y se guarda en tmp->content. Si dir no es NULL, se utiliza el valor de dir para actualizar tmp->content.
- **tmp->content = ft_strdup(getcwd(dir, 500));**: Si dir es NULL, se obtiene el directorio actual y se asigna a tmp->content.
- **tmp->content = ft_strdup(dir);**: Si dir no es NULL, se copia el valor de dir a tmp->content.

if (!tmp->content):

- Si la asignación de memoria para tmp->content falla (es decir, si ft_strdup devuelve NULL), la función termina y no realiza más cambios.

break;:

- Una vez que se actualiza la variable OLDPWD, se sale del bucle porque ya se ha encontrado y actualizado la variable correcta.

tmp = tmp->next;:

- Si no se encuentra la variable de entorno OLDPWD, se avanza al siguiente nodo de la lista enlazada (es decir, tmp se actualiza al siguiente elemento de la lista).

La función home_path está diseñada para buscar una variable de entorno específica, cuyo nombre se pasa como argumento en el parámetro str, y obtener su valor (que se encuentra en tmp->content). Luego, la función divide el contenido de la variable de entorno en dos partes usando = como delimitador y guarda la primera parte (la ruta de la variable) en data->aux

t_envp *tmp;:

- Se declara un puntero tmp de tipo t_envp, que probablemente representa una estructura que contiene una variable de entorno (su nombre y valor). Este puntero se utilizará para iterar a través de la lista de variables de entorno.

char **cnt;:

- Se declara un puntero a puntero de tipo char, llamado cnt, que se utilizará para almacenar el resultado de la función ft_split. La función ft_split dividirá el contenido de la variable de entorno en partes, usando el carácter = como delimitador.

tmp = data->envp;:

- Se inicializa tmp con el valor de data->envp, que apunta al comienzo de una lista enlazada de variables de entorno. Esto significa que tmp recorrerá la lista enlazada para buscar la variable de entorno que coincide con el valor de str.

while (tmp):

- Se inicia un bucle while que recorre la lista de variables de entorno mientras tmp no sea NULL. Cada iteración representa el acceso a una variable de entorno de la lista.

if (ft_strncmp(tmp->name, str, ft_strlen(str)) == EXIT_SUCCESS):

- Dentro del bucle, se compara el nombre de la variable de entorno (tmp->name) con la cadena str utilizando la función ft_strncmp. Esta función compara los primeros ft_strlen(str) caracteres de tmp->name con la cadena str. Si hay una coincidencia, significa que hemos encontrado la variable de entorno que buscamos.

cnt = ft_split(tmp->content, '=');:

- Si se encuentra la variable de entorno, se divide su contenido (tmp->content) usando el delimitador = mediante la función ft_split. Esto genera un array de cadenas (cnt) que contiene, en el primer elemento (cnt[0]), la parte antes del = (normalmente el nombre de la variable) y, en el segundo elemento (cnt[1]), la parte después del = (el valor de la variable).

data->aux = cnt[0];:

- Se guarda la primera parte del contenido de la variable de entorno (cnt[0]) en data->aux. Esto supone que la parte antes del = es relevante para lo que se quiere almacenar (por ejemplo, el nombre de la variable o la ruta asociada).

break;:

- Una vez que se ha encontrado la variable de entorno y se ha almacenado su valor en data->aux, se sale del bucle, ya que no es necesario seguir buscando.

tmp = tmp->next;:

- Si no se encuentra una coincidencia, tmp se actualiza para apuntar al siguiente elemento en la lista enlazada (tmp->next). El bucle continúa buscando en el siguiente nodo de la lista de variables de entorno.

La función up_currdir tiene como objetivo actualizar el valor de la variable de entorno PWD (Current Working Directory, directorio de trabajo actual) con la ruta del directorio en el que se encuentra el programa en el momento de la ejecución. A continuación, desglosamos la función paso a paso:

Explicación paso a paso:

1. **char currdir[500];**:
   - Se declara un arreglo de caracteres currdir de tamaño 500, que se usará para almacenar la ruta del directorio de trabajo actual que se obtiene con la función getcwd.
2. **t_envp *tmp;**:
   - Se declara un puntero tmp de tipo t_envp, que se utilizará para recorrer la lista de variables de entorno (data->envp).
3. **tmp = data->envp;**:
   - Se inicializa tmp con el valor de data->envp, que apunta al comienzo de la lista de variables de entorno. Este puntero se usará para iterar sobre cada nodo de la lista y buscar la variable de entorno PWD.
4. **while (tmp)**:
   - Se inicia un bucle while que recorre la lista de variables de entorno mientras tmp no sea NULL. Cada iteración representa el acceso a una variable de entorno en la lista.
5. **if (ft_strncmp(tmp->name, "PWD", 3) == EXIT_SUCCESS)**:
   - Dentro del bucle, se compara el nombre de la variable de entorno (tmp->name) con la cadena "PWD" utilizando ft_strncmp. Si los primeros 3 caracteres de tmp->name coinciden con "PWD", significa que hemos encontrado la variable de entorno que buscamos.
6. **free(tmp->content);**:
   - Se libera la memoria previamente asignada a tmp->content para evitar fugas de memoria. tmp->content es la cadena que almacena el valor actual de la variable de entorno PWD.

7. **tmp->content = ft_strdup(getcwd(currdir, 500));**:
    - Se actualiza tmp->content con una nueva cadena que contiene la ruta del directorio de trabajo actual, obtenida mediante la función getcwd. Esta función llena currdir con la ruta del directorio actual y la función ft_strdup duplica esa cadena para almacenarla en tmp->content.
8. **if (!tmp->content)**:
    - Se verifica si ft_strdup devolvió NULL, lo que indica que hubo un error al intentar duplicar la cadena. Si esto ocurre, la función termina de inmediato sin realizar ninguna otra acción.
9. **break;**
    - Si la variable de entorno PWD se ha encontrado y se ha actualizado correctamente, se sale del bucle usando break, ya que no es necesario seguir buscando en la lista de variables de entorno.
10. **tmp = tmp->next;**:
    - Si la variable de entorno PWD no se encuentra en el nodo actual, se avanza al siguiente nodo en la lista de variables de entorno.
11. **data->status = 0;**:
    - Finalmente, se establece el valor de data->status en 0, indicando que la operación fue exitosa.

EX_ECHO

Las tres funciones prints_echo, echo_print_args y ex_echo trabajan juntas para implementar la funcionalidad de la función echo en un entorno de shell personalizado, similar a cómo funcionaría el comando echo en un sistema operativo.

EX_ECHO

Propósito: Esta función maneja el comando echo, procesando los argumentos y decidiendo si se debe imprimir un salto de línea al final o no (en función del flag -n).

Parámetros:

- str: Un arreglo de cadenas con los argumentos pasados al comando echo.
- flag: Un indicador que controla si se debe o no imprimir un salto de línea al final.
- fd: El descriptor de archivo donde se imprimirá la salida (normalmente STDOUT).
- d: Un puntero a la estructura t_data, que contiene información de estado del programa.

Explicación:

- Primero, se cuenta cuántos argumentos hay en str con el bucle while (str[i]) i++.
- Si hay más de un argumento (i > 1):
    - Se verifica si el segundo argumento es -n (indicando que no se debe imprimir un salto de línea al final). Si es así:
        - Se establece el flag a 1 y se empieza a imprimir los argumentos desde el índice 2 (saltando el -n).
    - Si el segundo argumento no es -n, se imprime desde el primer argumento (índice 1).
- Después de imprimir todos los argumentos, si flag == 0 (es decir, no se encontró el -n), se imprime un salto de línea con ft_printf(fd, "\n");.
- Finalmente, se establece d->status = 0, indicando que la operación fue exitosa.

ECHO_PRINTS

Propósito: Esta función imprime todos los argumentos de `str`, llamando a `prints_echo` para cada uno de ellos, y añade un espacio entre los argumentos (excepto después del último).

Parámetros:

- `str`: Un arreglo de cadenas con los argumentos que se pasan al comando `echo`.
- `i`: El índice de la cadena en `str` desde donde se comenzará a imprimir.
- `fd`: El descriptor de archivo donde se imprimirá el texto.

Explicación:

- Se recorre cada cadena en `str`, comenzando desde el índice `i`.
- Para cada cadena `str[i]`, se llama a `prints_echo` para imprimirla.
- Si no es el último argumento o si el argumento es una cadena vacía (`str[i][0] == '\0'`), se imprime un espacio después de la cadena.
- El ciclo continúa hasta que se han imprimido todos los argumentos.

PRINTS_ECHO

Propósito: Esta función imprime un carácter por vez de una cadena de texto.

Parámetros:

- `str`: Es un arreglo de cadenas que contiene los argumentos pasados al comando `echo`.
- `i`: El índice del argumento dentro de `str` que se va a imprimir.
- `fd`: El descriptor de archivo donde se imprimirá el texto, normalmente `STDOUT` para imprimir en la pantalla.

Explicación:

- Se declara una variable `j` para iterar sobre los caracteres de `str[i]`.
- Se recorre cada carácter de la cadena `str[i]` e imprime uno a uno con `ft_printf`.
- Si el carácter es el final de la cadena (`'\0'`), se detiene el bucle.
- La función termina después de imprimir toda la cadena `str[i]`.

EX_EXIT

La función ex_exit implementa el comando exit en un shell, que se utiliza para finalizar la ejecución del programa o del shell. A continuación te explico paso a paso cómo funciona y qué hace cada parte de la función:

La función maneja el comando exit de un shell personalizado, que:

1. Imprime "exit" en la salida estándar.
2. Verifica si hay argumentos adicionales (y si son válidos) después del comando exit.
3. Libera los recursos relacionados con las rutas (si es necesario).
4. Termina la ejecución del programa con el código de salida adecuado.

Se imprime "exit" seguido de un salto de línea en el descriptor de archivo fd, que normalmente es STDOUT (la salida estándar). Esto muestra que el comando exit se ha ejecutado.

Aquí, se verifica si hay más de un argumento después de exit.

- Si node->all_cmd[1] y node->all_cmd[2] no son NULL, significa que el usuario ha pasado más de un argumento al comando exit (lo cual no es permitido).
- En ese caso, se imprime un mensaje de error ("error exit") y la función termina, sin realizar ninguna otra acción.

Si existe un argumento adicional después de exit (node->all_cmd[1] != NULL), la función verifica que este argumento sea un número válido.

- Si el primer carácter del argumento es un + o -, se incrementa el índice i para permitir que el número comience con un signo.
- Luego, se recorre el resto del argumento (node->all_cmd[1][i]) y se verifica que cada carácter sea un dígito usando la función ft_isdigit.
- Si se encuentra un carácter no numérico, se imprime un error ("error: exit") y se termina el bucle.
- Si el argumento no es un número válido, no se continúa con la ejecución del comando exit.

Si la variable data->path no es NULL (es decir, si se ha asignado alguna ruta o conjunto de rutas), se libera la memoria reservada para ella llamando a free_split(data->path).

Después de liberar la memoria, se establece data->path a NULL, para evitar accesos a memoria ya liberada.

Finalmente, se termina el programa o shell con la llamada a exit(data->status), donde data->status es el código de salida que se devolverá al sistema operativo.

- data->status es un valor que probablemente indica el estado de la ejecución del programa o shell. En un caso normal, este valor podría ser 0 (indicando éxito), o podría ser otro valor si hubo un error durante la ejecución.

EX_PWD

La función ex_pwd implementa el comando pwd (Print Working Directory), que se utiliza en shells para mostrar el directorio de trabajo actual. Aquí te explico paso a paso cómo funciona:

Propósito general:

La función obtiene y muestra el directorio de trabajo actual en el sistema de archivos. El resultado se imprime en el descriptor de archivo fd, que generalmente es STDOUT (la salida estándar).

La función getcwd obtiene la ruta del directorio de trabajo actual y la almacena en la variable currdir.

currdir es un arreglo de caracteres de 500 elementos, lo que significa que puede contener rutas de hasta 499 caracteres (más el terminador nulo '\0').

Si getcwd tiene éxito, la variable currdir contendrá el directorio de trabajo actual. Si ocurre un error, getcwd devuelve NULL, aunque en este caso no se maneja explícitamente el error.

Usando la función ft_printf (una versión personalizada de printf), se imprime el contenido de currdir, que es el directorio de trabajo actual.

Se usa %s para imprimir el contenido de currdir como una cadena de caracteres.

Después de la cadena, se imprime un salto de línea \n para hacer que la salida sea más legible.

Si el directorio de trabajo actual es /home/user, la función imprimirá:

/home/user

EX_UNSET

La función ex_unset implementa el comando unset de la shell, que elimina una variable de entorno de la lista de variables de entorno (almacenada en data->envp). Dado un nombre de variable, busca el nodo correspondiente en la lista y lo elimina, liberando la memoria asociada.

if (!str): Si no se pasa un nombre de variable, la función retorna sin hacer nada.

curr_node = data->envp: Se inicia el recorrido desde el primer nodo de la lista de variables de entorno (data->envp).

if (!ft_strncmp(curr_node->name, str, ft_strlen(str))): Compara el nombre de la variable de entorno (curr_node->name) con el argumento str que se pasó. Si son iguales, procede a eliminar ese nodo.

del_node = curr_node: Asigna el nodo actual a la variable del_node, que es el nodo que se eliminará.

if (prev_node): Si el nodo a eliminar no es el primer nodo, conecta el nodo anterior (prev_node) con el siguiente nodo de la lista (curr_node->next), eliminando efectivamente el nodo.

else: Si el nodo a eliminar es el primero de la lista, actualiza la cabeza de la lista (data->envp) para que apunte al siguiente nodo.

free_envp_node(del_node, data): Llama a la función free_envp_node para liberar la memoria del nodo eliminado.

check_path(data): Si no se encuentra la variable de entorno a eliminar, la función check_path es llamada para verificar o actualizar las variables de entorno, específicamente el PATH.

data->status = 1: Si no se encontró el nodo para eliminar, se establece el estado de data a 1, posiblemente indicando un error en la operación.

La función free_envp_node libera la memoria asociada a un nodo de la lista enlazada de variables de entorno (t_envp). Este nodo se pasa como argumento, y la función libera la memoria utilizada para el nombre, el contenido, y finalmente el nodo en sí.

free(remove->name): Libera el nombre de la variable de entorno (almacenado en name del nodo remove).

free(remove->content): Si existe contenido en el nodo (content no es NULL), se libera esa memoria también.

free(remove): Finalmente, se libera la memoria ocupada por el nodo en sí.

remove = NULL: Aunque esto no afecta el funcionamiento del programa en este punto, se asegura de que el puntero remove ya no apunte a una ubicación de memoria liberada.

data->status = 0: Actualiza el estado de data a 0, posiblemente para indicar que la operación se completó con éxito.

EX_ENVP

La función ex_envp imprime todas las variables de entorno almacenadas en la lista enlazada data->envp en el archivo de salida especificado por fd. Excluye de la impresión las variables de entorno cuyo contenido esté rodeado por comillas ("), verificando si el segundo y tercer carácter de content son comillas.

t_envp *tmp = data->envp;:

- Se inicializa un puntero tmp para recorrer la lista de variables de entorno, comenzando desde el primer nodo de data->envp.

while (tmp != NULL):

- Se utiliza un bucle while para recorrer la lista de variables de entorno. El bucle continuará hasta que tmp sea NULL, lo que indica el final de la lista.

Condicional if ((tmp->content[1] != "") && (tmp->content[2] != "")):

- Este condicional verifica si los caracteres en las posiciones 2 y 3 de tmp->content **no son comillas dobles**. Si son comillas, significa que el valor de la variable de entorno está entre comillas, por lo que la variable no se imprimirá.
- Si no son comillas dobles en las posiciones 2 y 3, la variable será impresa.

ft_printf(fd, "%s%s\n", tmp->name, tmp->content);:

- Si el condicional es verdadero, se imprime la variable de entorno usando ft_printf. Imprime primero el nombre de la variable (tmp->name), seguido del contenido de la variable (tmp->content).
- La salida se envía al descriptor de archivo fd, lo que puede ser un archivo o la salida estándar (dependiendo del valor de fd).

tmp = tmp->next;:

- Se avanza al siguiente nodo en la lista, es decir, tmp se actualiza para apuntar al siguiente nodo de la lista de variables de entorno.

HERE_DOC

La función ft_heredoc simula una operación de *heredoc* en un shell, que consiste en leer líneas de entrada hasta que se encuentra una línea de terminación específica, guardando esas líneas en un archivo temporal. Luego, ese archivo se abre para ser utilizado como entrada estándar en el futuro.

Esta función lee líneas de entrada de stdin (como un comando de shell en el que el usuario proporciona datos) y las guarda en un archivo temporal llamado "here_doc.tmp". Cuando el contenido de una línea coincide con un valor de terminación definido, el proceso se detiene. Finalmente, el archivo se cierra y se reabre para ser utilizado más adelante.

Se abre el archivo "here_doc.tmp" con los permisos O_WRONLY | O_CREAT | O_TRUNC. Esto significa que el archivo será abierto en modo solo escritura, se creará si no existe y se truncará (se borrará) si ya existe.

Los permisos del archivo son 0644, lo que da permisos de lectura y escritura al propietario, y solo lectura a los demás usuarios.

Si la operación de apertura falla, se sale de la función (return).

Se entra en un bucle infinito (while (1)) para seguir leyendo las líneas.

write(1, "heredoc> ", 9) muestra un mensaje para indicar al usuario que debe ingresar una línea.

Se llama a get_next_line(data->dup_stdin) para obtener la siguiente línea de la entrada estándar (originalmente desde el terminal).

Si get_next_line devuelve NULL (lo que indica un error o el fin del archivo), se llama a error_msg para manejar el error.

Se compara la línea leída (str) con el valor de data->filein. ft_strncmp se usa para comparar las primeras ft_strlen(data->filein) caracteres de ambas cadenas.

Si la línea coincide con el valor de terminación, se libera la memoria de str, se cierra el descriptor de entrada estándar (data->dup_stdin) y se sale del bucle con break.

Si la línea no es la de terminación, se escribe en el archivo temporal "here_doc.tmp" usando ft_putstr_fd(str, (*node)->filein).

Luego, se libera la memoria de str.

Una vez que se ha terminado de escribir las líneas en el archivo temporal, se cierra el archivo con close((*node)->filein).

Luego, el archivo "here_doc.tmp" se vuelve a abrir en modo solo lectura (O_RDONLY).

data->dup_stdin = dup(STDIN_FILENO) guarda una copia del descriptor de archivo estándar de entrada, lo que permite restaurar la entrada estándar más adelante.

Si la operación de apertura falla, se llama a error_msg para manejar el error.