# **Minishell**

**Ce inseamna un Shell?**

Un **shell** este un program care acționează ca interfață între utilizator și sistemul de operare.   
Mai simplu spus:

🔹 **Shell-ul interpretează comenzile introduse de utilizator și le execută.**

### **🧠 Ce face un shell mai exact?**

1. **Primește input-ul de la utilizator** – ex: ls -l
2. **Parsează** comanda – înțelege ce vrei să faci
3. **Caută comanda** în sistem (în PATH sau după path absolut)
4. **Creează un proces nou** pentru a executa comanda
5. **Execută comanda** și afișează rezultatul în terminal
6. **Așteaptă alte comenzi** – se întoarce în bucla principală

### 🖥️ Exemple de shell-uri:

* bash (Bourne Again Shell) – cel mai comun pe Linux
* zsh, fish – alternative moderne
* sh – shell-ul clasic
* **minishell** – o versiune simplificată construita dupa cerintele minishell project in cadrul scolii 42 (Minishell - As beautiful as a shell)

### 🔧 De ce este important shell-ul?

* Îți permite să rulezi programe și să automatizezi sarcini
* Poți scrie scripturi (fisiere .sh) pentru operații repetitive
* Este esențial pentru administrarea sistemelor Linux/Unix

### ✅ Câte variabile globale sunt folosite? De ce?

În proiectul nostru **folosim o singură variabilă globală**:

pid\_t g\_child\_pid = 0;

### 🔍 De ce este necesară această variabilă?

Această variabilă este utilizată pentru a **reține PID-ul(process id) procesului copil curent** care este creat atunci când se execută o comandă. Este **esențială pentru tratarea corectă a semnalelor**, în special atunci când utilizatorul trimite un semnal de întrerupere, cum ar fi Ctrl+C.

Dacă utilizatorul apasă Ctrl+C în timpul execuției, shell-ul trebuie să știe **care este PID-ul procesului** sleep pentru a-l putea opri.

Fără această variabilă globală, shell-ul **nu ar ști care proces trebuie întrerupt**, și ar putea ignora semnalul sau în cel mai rău caz ar opri shell-ul în sine.

### 🧠 De ce este logic și necesar?

Este **logic** să fie globală pentru că trebuie accesată atât de funcția care creează procesul copil (fork()), cât și de funcția care tratează semnalele (handler-ul de SIGINT). Dacă nu ar fi globală, ar trebui să transmitem PID-ul între multe funcții, lucru care ar complica design-ul și ar încălca cerințele proiectului (care permit o singură variabilă globală).

Să presupunem că utilizatorul scrie în shell:

minishell$ sleep 10

În acel moment, shell-ul creează un **proces copil** pentru a executa comanda sleep. Dacă utilizatorul apasă Ctrl+C înainte ca sleep să termine, shell-ul trebuie să știe **ce proces să termine**.

Fără variabila globală **g\_child\_pid,** handler-ul pentru SIGINT **nu ar ști ce PID să oprească** și comanda sleep ar continua să ruleze.

**Deci, g\_child\_pid este necesară pentru ca shell-ul să se comporte corect, ca un shell real.**

### ✅ ****De ce am ales să implementez Minishell folosind structuri de date și tokenizare****

Am ales această metodă deoarece:

1. **Modularitate și claritate**
   * Împărțind inputul în tokeni (cuvinte, operatori, redirecționări, pipe-uri etc.), fiecare parte a liniei de comandă poate fi tratată separat și logic.
   * Acest lucru permite o implementare clară, unde fiecare pas este responsabil de o etapă specifică: **tokenizare**, **expansiune**, **parsing**, **execuție**.
2. **Scalabilitate**
   * Folosind structuri de date dinamice (liste, arbori), shell-ul devine extensibil:
     + Pot adăuga mai ușor suport pentru redirecționări complexe, pipe-uri multiple, subshells sau and/or (&&, ||).
     + Pot construi un **arbore sintactic (parse tree)** pentru a reflecta structura logică a comenzilor.
3. **Gestionare precisă a contextului**
   * Structura t\_token păstrează informații utile despre fiecare cuvânt:
     + tipul de token (comandă, argument, redirecționare etc.)
     + dacă este sau nu între ghilimele
   * Structura t\_node păstrează ierarhia comenzilor (ex: în cazul ls | grep txt > out.txt), ceea ce permite execuția corectă în funcție de ordine și priorități.
4. **Respectarea cerințelor proiectului**
   * Proiectul impune tratarea corectă a ghilimelelor, variabilelor, pipe-urilor și redirecționărilor. Acestea nu pot fi rezolvate eficient doar cu split, ci necesită analiză sintactică reală — pe care o fac prin **tokenizare și parsing**.
5. **Gestionarea erorilor**
   * Folosind structuri de date, pot verifica ușor:
     + ghilimele neînchise
     + sintaxă invalidă (ex: | ls, cat <)
     + lipsa argumentelor
   * Astfel, shell-ul devine mai robust și poate oferi feedback clar utilizatorului.

## 📚 Rezumat scurt:

## ****Tokenizare**** = procesul de a împărți o linie de input în bucăți logice (tokeni), fiecare având un înțeles pentru shell.

Structura t\_token **nu este o structură standard** dintr-o bibliotecă C sau POSIX – **nu există o "librărie oficială" care să o definească**. Este o convenție folosită frecvent de dezvoltatori atunci când implementează interpretoare (interpreters), compilatoare sau shell-uri.

### 🔹 De unde provine t\_token?

1. **Nu este parte din vreo bibliotecă standard**  
   t\_token este **o structură de date definită de programator**, nu o structură predefinită în C. Numele token provine din teoria limbajelor formale și a compilatoarelor, unde:

Un **token** este o unitate elementară de sens (ex: un cuvânt cheie, o variabilă, un operator, un literal).

1. **Folosit în parsere și interpretoare**  
   În shell-uri sau compilatoare, linia introdusă de utilizator este împărțită în **tokens** pentru a fi analizată.  
   De exemplu, comanda:

echo "$USER" > out.txt

este împărțită în tokens:

* echo (command)
* "$USER" (argument cu expansiune)
* > (operator de redirecționare)
* out.txt (fișier de output

**De unde am învățat de tokens și** t\_token**?**

* Din documentații și articole despre **parsing** și **interpretoare**.
* **[https://en.wikipedia.org/wiki/Lexical\_analysis#Token](https://en.wikipedia.org/wiki/Lexical_analysis" \l "Token)**

## **🎯 **Obiectiv general****

Proiectul minishell implementează un shell minimal care reproduce comportamentul de bază al bash, incluzând:

* execuție comenzi simple,
* redirecționări și pipe-uri,
* variabile de mediu și exit status,
* builtin-uri (cd, echo, pwd, export, unset, env, exit),
* manipulare semnale în mod interactiv.

## ⚙️ ****Funcționalități implementate****

### ✅ 1. Prompt și execuție comenzi

* Afișează prompt personalizat cu readline().
* Stochează istoric comenzi.
* Comenzile sunt executate fie ca builtin, fie cu execve(), folosind path-ul din env.

📂 Fișiere: main.c, minishell\_loop.c, resolve\_command.c, execute\_word.c

### ✅ 2. Redirecționări: <, >, >>, <<

* Tokenizare redirecții speciale.
* Construirea arborelui execuției cu redirecții.
* dup2() și open()/close() pentru manipularea STDIN/STDOUT.
* Heredoc funcțional cu readline("> ").

📂 Fișiere:  
token\_utils.c, parse\_command.c,  
execute\_redirections.c, execute\_redir\_input/output/heredoc.c,  
execute\_word\_utils.c, free\_tree.c

### ✅ 3. Pipe-uri |

* Construiește arbori de execuție pentru comenzi legate prin |.
* Procesele sunt conectate prin pipe() și dup2() pe STDIN/STDOUT.
* Se așteaptă corect terminarea tuturor proceselor.

📂 Fișiere:  
parse\_pipeline.c, create\_pipe\_node.c,  
execute\_pipe.c, execute\_pipes\_utils.c

### ✅ 4. Expansiuni: $VAR, $?

* Expandare variabile de mediu din env\_list.
* $? înlocuit cu sh->exit\_code.
* $ nu este expandat în ghilimele simple ('...').
* Quotes sunt procesate corect: "..." permite $, '...' nu.

📂 Fișiere:  
expand\_dollar.c, expand\_dollar\_utils.c,  
expand\_protect\_single\_quotes.c, expand\_quotes\_dollar.c,  
input\_processing.c

### ✅ 5. Globbing \*

* Detectează și expandează \* în argumente dacă nu sunt quoted.
* Listează fișierele din directorul curent folosind opendir().

📂 Fișiere:  
expand\_asterisk.c, expand\_globbing\_utils.c, utils\_tokens\_asterisk.c,  
expand\_args\_utils.c, expanding\_utils.c

### ✅ 6. Gestionare semnale

| Semnal | Comportament |
| --- | --- |
| Ctrl-C | Prompt nou, fără ieșire |
| Ctrl-D | Iese din shell dacă linia e goală |
| Ctrl-\ | Ignorat complet |

📂 Fișiere:  
signal.c, minishell\_loop.c, main.c

### ✅ 7. Builtin-uri

| Comandă | Fișier | Descriere |
| --- | --- | --- |
| echo | builtin\_echo.c | cu -n |
| cd | builtin\_cd.c | update PWD |
| pwd | builtin\_pwd.c | print dir |
| export | builtin\_export.c, handle\_export.c, process\_for\_export.c | adaugă variabile |
| unset | builtin\_unset.c | șterge din env |
| env | builtin\_env.c | afișează toate |
| exit | builtin\_exit.c | termină shell-ul |

✔️ export este tratat **separat** pentru a evita interpretarea greșită ($, \* etc.)  
✔️ Toate builtin-urile suportă fd\_out pentru redirecționare.

📂 + execute\_builtin\_utils.c, execute\_builtin\_handler.c, env\_utils.c

**Schita a Minishell\_project**

minishell\_project/

├── 🗂️ inc/

│ └── minishell.h

│

├── 🗂️ libft/

│ └── (toate funcțiile libft - implementarea ta)

│

├── 🗂️ scripts/

│ └── install\_readline\_local.sh (se instaleaza readline)

│

├── 🗂️ src/

│ │── execute/ (execute\_word.c, execute\_word\_utils.c, execute.c, execute\_redirections.c, execute\_redir\_errors.c , execute\_redir\_heredoc.c, execute\_redir\_input.c, execute\_redir\_output.c, execute\_utils.c, execute\_builtin\_utils.c, execute\_pipe, execute\_pipe-utils.c, execute\_builtin\_handler.c)

│

│ ├──🗂️ expander/ (expand\_args\_utils.c, expand\_asterisk.c, expand\_dollar.c, expand\_dollar\_utils.c, expand\_globbing\_utils.c, expanding\_utils.c, expand\_protect\_single\_quotes.c, expand\_quotes\_dollar.c)

│

│ ├── 🗂️ lexer/ (check\_input\_errors.c, handle\_quotes.c)

│

│ ├──🗂️ minishell/ (cleanup\_readline.c, input\_processing.c, main.c, minishell\_loop.c, handle\_export.c, process\_for\_export.c)

│

│ ├── 🗂️ builtins (builtin\_echo.c, builtin\_cd.c, builtin\_pwd.c, builtin\_env.c, builtin\_export.c, builtin\_unset.c, builtin\_exit.c)

│

│ ├── 🗂️ parse/ (resolve\_command.c, parse\_pipeline.c)

│

│ ├── 🗂️ treenodes/ (create\_pipe\_node.c, free\_tree.c, parse\_command.c, treee\_parse\_args.c)

│

│ └── 🗂️ utils/ (debug\_print\_tokens.c, env\_utils.c, free\_str\_array.c, init\_shell.c, signal.c, split\_args\_preserving\_quotes.c, token\_utils.c, utils\_env\_strjoin.c,

│ utils\_is\_quoted.c, utils\_is\_valid\_varname.c, utils\_remove\_quotes.c,

│ utils\_tokens\_asterisk.c, utils\_tokens\_free.c)

│

│── Makefile

**└── script.sh**

## **📘 Descriere pe scurt a folderelor**

| Folder/Fisier | Descriere |
| --- | --- |
| inc/ | Conține headerul principal minishell.h, care declară toate structurile, tipurile și prototipurile de funcții. |
| libft/ | Librărie personală (funcții standard + utile). Se folosește în tot proiectul. |
| minishell\_test/ | Script de testare pentru funcționalități de bază (tests\_simple.sh). |
| scripts/ | Script pentru instalarea locală a readline dacă lipsește (install\_readline\_local.sh). |
| src/ | Codul sursă principal, împărțit pe module/folder logic. |
| execute/ | Conține logica pentru execuția comenzilor (cu fork, execve). |
| expander/ | Se ocupă de expansiuni: variabile ($), globbing (\*), quotes etc. |
| lexer/ | Conține logica de verificare a inputului: quotes neînchise, caractere interzise. |
| minishell/ | Punctul central: main.c, loop-ul shellului, input etc. |
| builtins | Builtin-uti implementate |
| parse/ | Conține funcții de parsing, de ex. pentru rezolvarea comenzilor în PATH. |
| treenodes/ | (Gol deocamdată) – va fi folosit pentru construirea arborelui de execuție. |
| utils/ | Funcții ajutătoare pentru shell, env, liste, tokens, etc. |
| Makefile | Fisierul de compilare – construiește binarul shell-ului. |
| script.sh | Script suplimentar – posibil folosit pentru execuții automate/debug/test. |

# **minishell.h — scheletul proiectului nostru**

# Fișierul minishell.h este coloana vertebrală a proiectului Minishell. Conține:

* **Definiții de tipuri și structuri de date** (cum arată datele noastre)
* **Enum-uri** (tipuri de tokeni, tipuri de ghilimele)
* **Prototipuri de funcții** (spune ce funcții există și unde să le găsească compilatorul)

## **✅ STRUCTURI CHEIE**

### 🔹 Enum-uri: e\_token\_type Reprezintă tipurile de tokeni identificați în input:

typedef enum e\_token\_type

{

TOK\_WORD = 1,  
 TOK\_PIPE,  
 TOK\_REDIR\_IN,  
 TOK\_REDIR\_OUT,  
 TOK\_APPEND,  
 TOK\_HEREDOC,  
 TOK\_INVALID

} t\_token\_type;

* TOK\_WORD: cuvinte normale (comenzi sau argumente)
* TOK\_PIPE: |
* TOK\_REDIR\_IN, TOK\_REDIR\_OUT, TOK\_APPEND, TOK\_HEREDOC: pentru redirecții
* TOK\_INVALID: token invalid detectat în input

🟩 **Folosit în**: Tokenizare, parsing, execuție.

e\_quote\_type  
  
Reprezintă tipul de ghilimele care înconjoară un token:

typedef enum e\_quote\_type

{  
 QUOTE\_NONE,  
 QUOTE\_SINGLE,  
 QUOTE\_DOUBLE  
 } t\_quote\_type;

* QUOTE\_NONE: fără ghilimele
* QUOTE\_SINGLE: între '…'
* QUOTE\_DOUBLE: între "..."

🟩 **Folosit în**: Gestionarea expansiunii variabilelor și a globbing-ului (\*).

🔹 Structuri principale

s\_token — un token din input:

typedef struct s\_token  
 {  
 t\_token\_type type;  
 char \*content;  
 bool was\_quoted;  
 t\_quote\_type quote\_type;  
 struct s\_token \*next;  
 struct s\_token \*prev;  
 } t\_token;

* content: textul tokenului
* type: tipul lui (WORD, PIPE, REDIR, etc.)
* was\_quoted: dacă era între ghilimele
* quote\_type: tipul de ghilimele (' sau ")

🟩 **Folosit în**: tokenize\_input(), expand\_args\_globbing\_from\_tokens(), execuție.

**s\_node** — un nod din arborele de execuție:

typedef struct s\_node  
 {  
 t\_token\_type type;  
 char \*value;  
 char \*\*args;  
 int heredoc\_fd;  
 struct s\_node \*left;  
 struct s\_node \*right;  
 } t\_node;

* value: comanda
* args: argumentele
* heredoc\_fd: file descriptor pentru heredoc, dacă există
* left și right: pentru pipe-uri (|)

🟩 **Folosit în**: Parsing și execuție.

**s\_env** — variabilă de mediu:

typedef struct s\_env  
 {  
 char \*key;  
 char \*value;  
 struct s\_env \*next;  
 struct s\_env \*prev;  
 } t\_env;

* key: numele variabilei
* value: valoarea variabilei

🟩 **Folosit în**: Expansiunea $VAR, builtins export, unset, env.

**s\_shell** — structura principală a shellului:

typedef struct s\_shell  
 {  
 t\_node \*root;  
 t\_token \*tokens;  
 t\_env \*env\_list;  
 int exit\_code;  
 char \*line\_input;  
 } t\_shell;

* root: arborele de execuție
* tokens: lista tokenilor generați
* env\_list: lista de variabile de mediu
* exit\_code: codul ultimei comenzi executate ($?)
* line\_input: linia brută citită de la user

🟩 **Folosit în**: Aproape toate funcțiile shell-ului.

**Structuri auxiliare:**

* s\_arg\_info  
  Păstrează un argument și informația dacă era quoted.
* s\_pipe\_data  
  Gestionarea execuției între procese prin pipe-uri.
* s\_quote\_flags  
  Starea curentă în timpul procesării ghilimelelor (dacă suntem într-un string cu ' sau ").

## ✅ FUNCȚII IMPORTANTE

În minishell.h sunt declarate toate prototipurile funcțiilor organizate după foldere.

## ✅ Rolul fiecărei părți

* **Tokenizarea** → Spargem inputul în tokeni ușor de procesat
* **Parsing** → Construim arborele de execuție
* **Execuție** → Rulăm comenzile în procese, redirecționăm, facem pipe
* **Expansiune** → Expandăm variabile ($USER) și pattern-uri (\*)
* **Builtin-uri** → Tratamente speciale pentru comenzi interne ca cd, exit
* **Utils** → Inițializăm shell-ul, gestionăm memoria, debug.

📚 **DOCUMENTAȚIE – Implementarea Cerințelor Inițiale Minishell**

## 1. ****Display a prompt when waiting for a new command****

### 🔹 Cerință:

* Afișează un prompt (minishell$) și așteaptă input de la utilizator.

### 🔹 Fisiere implicate:

* minishell/main.c
* minishell/minishell\_loop.c

### 🔹 Funcții implicate:

* minishell\_banner(void)  
  → Afișează banner-ul colorat la pornirea shell-ului.
* run\_shell\_loop(t\_shell \*sh)  
  → Creează promptul (get\_prompt()), apelează readline(), și așteaptă input-ul utilizatorului.
* get\_prompt(void) (statică în minishell\_loop.c)  
  → Generează stringul "minishell$ ".

### 🔹 Logica:

* La pornire, shell-ul afișează un bannerul.
* În loop-ul principal (run\_shell\_loop()), se afișează promptul și se citește input-ul folosind readline().
* Se gestionează corect terminarea (CTRL+D) și memorarea istoriei.

## 2. ****Have a working history****

### 🔹 Cerință:

* Comenzile introduse sunt salvate și pot fi accesate cu săgețile ↑ și ↓.

### 🔹 Fisiere implicate:

* minishell/minishell\_loop.c
* minishell/cleanup\_readline.c

### 🔹 Funcții implicate:

* run\_shell\_loop(t\_shell \*sh)  
  → Adaugă fiecare linie de input validă în istorie cu add\_history(line).
* cleanup\_readline(void)  
  → Curăță memoria și istoria la închiderea shell-ului (rl\_clear\_history(), rl\_free\_line\_state(), rl\_deprep\_terminal()).

### 🔹 Logica:

* readline() gestionează automat istoria.
* După fiecare comandă introdusă, add\_history(line) o salvează.
* La ieșire (exit sau CTRL+D), apelăm cleanup\_readline() ca să curățăm memoria readline.

## 3. ****Search and launch the right executable****

### 🔹 Cerință:

* Shell-ul trebuie să găsească executabilul corect:
  + dacă este în PATH
  + dacă este o cale relativă (ex: ./script.sh)
  + dacă este o cale absolută (ex: /bin/ls)

### 🔹 Fisiere implicate:

* parse/resolve\_command.c
* execute/execute\_word.c
* execute/execute\_word\_utils.c
* utils/utils\_env\_strjoin.c

### 🔹 Funcții implicate:

* find\_command\_path(t\_shell \*sh, char \*cmd)  
  → Caută executabilul în PATH sau verifică dacă este cale relativă/absolută.
* execute\_word(t\_shell \*sh, t\_node \*node)  
  → Caută path-ul comenzii și începe execuția.
* fork\_and\_execute(t\_shell \*sh, char \*path, char \*\*args)  
  → Creează un proces copil și execută comanda cu execve.
* convert\_env\_to\_array(t\_env \*env)  
  → Transformă lista de environment variables în formatul cerut de execve.

### 🔹 Logica:

* Dacă comanda are / sau access(cmd, X\_OK) -> e relativă/absolută.
* Dacă nu, căutăm în toate directoarele din PATH.
* Dacă găsim executabilul, facem fork() + execve() în proces copil.
* La eroare (command not found), afișăm mesaj și setăm exit code 127.

## 4. ****Avoid using more than one global variable to indicate a received signal****

### 🔹 Cerință:

* Folosim maximum **o singură** variabilă globală pentru a indica dacă am primit semnal (ex: SIGINT).

### 🔹 Fisiere implicate:

* minishell/main.c
* utils/signal.c

### 🔹 Variabilă globală:

pid\_t g\_child\_pid = 0;

* Definita în main.c.

### 🔹 Funcții implicate:

* handle\_signals(void)  
  → Setează signal handler pentru SIGINT și SIGQUIT.
* Signal handlers vor folosi g\_child\_pid să determine:
  + dacă suntem în shell principal (prompt)
  + sau într-un proces copil (în execuție).

### 🔹 Logica:

* Dacă suntem în shell principal (g\_child\_pid == 0) și primim CTRL+C (SIGINT), curățăm promptul fără să terminăm shell-ul.
* Dacă suntem într-un copil (g\_child\_pid != 0), semnalul termină copilul.

# 📚 Detectare ghilimele neînchise și caractere interzise + Gestionare ghilimele simple

## 1. ****Not interpret unclosed quotes or special characters like \ (backslash) or ; (semicolon)****

### 🔹 Cerință:

* Dacă inputul are ghilimele simple ' sau duble " neînchise → trebuie detectat și refuzat.
* Dacă inputul conține caractere interzise (\ sau ;) → trebuie detectat și refuzat.
* **Shell-ul nu trebuie să încerce să ruleze comanda invalidă!**

### 🔹 Fisiere implicate:

* lexer/handle\_quotes.c
* lexer/check\_input\_errors.c
* minishell/minishell\_loop.c

### 🔹 Funcții implicate:

* has\_unclosed\_single\_quotes(const char \*input) (handle\_quotes.c)  
  → Verifică dacă sunt ghilimele simple ' neînchise.
* has\_unclosed\_double\_quotes(const char \*input) (check\_input\_errors.c)  
  → Verifică dacă sunt ghilimele duble " neînchise.
* has\_forbidden\_chars(const char \*input) (check\_input\_errors.c)  
  → Detectează caractere interzise (;, \) care nu sunt în interiorul ghilimelelor.
* has\_input\_errors(const char \*line) (minishell\_loop.c)  
  → Apelează funcțiile de mai sus înainte de procesarea inputului.

### 🔹 Logica abordată:

* La fiecare input citit, shell-ul apelează has\_input\_errors(line).
* Dacă inputul are erori (quotes neînchise sau caractere interzise):
  + Se afișează un mesaj de eroare ("minishell: unclosed single quote", "forbidden characters" etc.).
  + Linia NU este procesată mai departe.
* Astfel, se previne execuția comenzii greșite și shell-ul rămâne stabil.

## 2. ****Handle ' (single quote) which should prevent the shell from interpreting meta-characters****

### 🔹 Cerință:

* Dacă inputul este între ghilimele simple '...', shell-ul trebuie să trateze totul ca text brut.
* Nu se face **nici expansiune de variabile** ($VAR), **nici interpretare de caractere speciale**.

### 🔹 Fisiere implicate:

* expander/expand\_protect\_single\_quotes.c
* expander/expand\_dollar.c
* lexer/input\_processing.c

### 🔹 Funcții implicate:

* protect\_dollar\_in\_single\_quotes(const char \*input) (expand\_protect\_single\_quotes.c)  
  → Înlocuiește toate caracterele $ din interiorul '...' cu un caracter special (\x01).
* expand\_line(t\_shell \*sh, const char \*input) (expand\_dollar.c)  
  → Apelează protect\_dollar\_in\_single\_quotes() înainte de expansiunea variabilelor.
* tokenize\_input(const char \*input) (input\_processing.c)  
  → La tokenizare setează **was\_quoted și quote\_type** pentru fiecare token.

### 🔹 Logica abordată:

* La începutul expansiunii (expand\_line()), apelăm protect\_dollar\_in\_single\_quotes(input).
* Aceasta:
  + Detectează dacă ne aflăm într-un șir cuprins între '...'.
  + Protejează orice $ găsit acolo, astfel încât **expanderul să nu le trateze** ca variabile.
* Mai departe, în expand\_env\_variables(), dacă întâlnim \x01, înlocuim cu caracterul $ fără expansiune.
* Astfel, secvențele între ghilimele simple sunt păstrate literal, conform cerinței din subiect.

# 📚 DOCUMENTAȚIE – Gestionarea "double quotes" și globbing (\*)

## 1. ****Handle " (double quotes) — permite doar expansiune $****

### 🔹 Cerință:

* În interiorul "...":
  + **se expandează** doar variabilele ($VAR);
  + **nu se interpretează** alte caractere speciale (\*, |, ;, etc.).

### 🔹 Fisiere implicate:

* expander/expand\_quotes\_dollar.c
* expander/expand\_dollar.c
* lexer/input\_processing.c

### 🔹 Funcții implicate:

* remove\_double\_quotes\_preserve\_dollar(const char \*input)  
  (expand\_quotes\_dollar.c)  
  → Elimină ghilimelele duble "..." din input, păstrând $ pentru expansiune.
* expand\_line(t\_shell \*sh, const char \*input)  
  (expand\_dollar.c)  
  → Face pre-procesarea inputului:
  + Protejează $ în ghilimele simple '...'
  + Elimină "..." păstrând $
  + Apelează expand\_env\_variables().
* tokenize\_input(const char \*input)   
  (input\_processing.c)  
  → Creează lista de tokeni și marchează tokenii quoted (was\_quoted = true) pentru controlul ulterior al globbing-ului.

### 🔹 Logica abordată:

1. La primirea inputului:
   * tokenize\_input() sparge în tokeni și setează was\_quoted.
2. În expand\_line():
   * Dacă inputul conține "...", remove\_double\_quotes\_preserve\_dollar() elimină doar ghilimelele, păstrând $VAR intact pentru expansiune.
3. Apoi:
   * expand\_env\_variables() detectează $ și îl înlocuiește cu valoarea din environment.
4. Rezultatul:
   * "Hello $USER" → Hello roxana dacă $USER=roxana.
   * "\*" → rămâne \*, fără expandare.

## 2. ****Handle globbing (****\*****) — doar dacă nu este quoted****

### 🔹 Cerință:

* Expandăm caracterul **\*** în lista fișierelor existente, **dar numai** dacă:
  + **\*** NU este într-un token quoted ('...' sau "...").

### 🔹 Fisiere implicate:

* expander/expand\_args\_utils.c
* expander/expand\_asterisk.c
* expander/expand\_globbing\_utils.c
* utils/utils\_tokens\_asterisk.c
* expander/expanding\_utils.c

### 🔹 Funcții implicate:

* expand\_args\_globbing\_from\_tokens(t\_token \*tokens) (expand\_args\_utils.c)  
  → Creează un nou array de argumente (char \*\*) aplicând globbing doar pe tokeni **nequoted** care conțin **\*.**
* expand\_asterisk(void) - (expand\_asterisk.c)  
  → Returnează lista de fișiere din directorul curent care nu încep cu . (ascunse).
* has\_unquoted\_asterisk(const char \*str) - (expand\_globbing\_utils.c)  
  → Verifică dacă există **\*** în afara ghilimelelor.
* replace\_asterisk\_with\_files(const char \*input) - (expand\_globbing\_utils.c)  
  → Înlocuiește **\*** cu lista de fișiere existente.
* extract\_args\_from\_tokens(t\_token \*tokens) - (utils\_tokens\_asterisk.c)  
  → Pregătește argumentele pentru execuție, aplicând globbing unde este necesar.

### 🔹 Logica abordată:

1. La input:
   * tokenize\_input() creează tokeni și marchează dacă un token era quoted (was\_quoted).
2. În timpul procesării:
   * expand\_args\_globbing\_from\_tokens() parcurge tokenii:
     + Dacă tokenul **nu era quoted** și conține \* → aplică globbing (expand\_asterisk()).
     + Dacă era quoted → păstrează \* ca text simplu.
3. Astfel:
   * **echo \*** → afișează toate fișierele.
   * **echo "\*"** → afișează pur și simplu **\*** .

# ✅ Rezumat: Implementarea redirecțiilor în Minishell

În Minishell, redirecțiile <, >, <<, >> sunt complet funcționale și respectă cerințele subiectului.  
Execuția lor este împărțită în parsing, validare, pregătire redirecții și execuție efectivă a comenzilor.

# 📁 Fișiere și scopul lor

## 🔹 ****utils/token\_utils.c****

* **Funcții:**
  + handle\_special\_token() — recunoaște >, >>, <, <<.
  + set\_token\_type() — setează tipurile de tokeni (TOK\_REDIR\_IN, TOK\_APPEND etc).
* **Scop:**  
  Recunoaște și marchează tokenii speciali de redirecție încă din tokenizer.

## 🔹 ****treenodes/parse\_command.c****

* **Funcții:**
  + parse\_command(t\_token \*\*tokens) — creează un arbore t\_node.
  + parse\_redirections(t\_node \*cmd\_node, t\_token \*tokens) — separă redirecțiile <, >, <<, >>.
* **Scop:**  
  Creează arborele de execuție, cu:
  + args[] pentru argumente normale,
  + left pentru redirecții de input/heredoc,
  + right pentru redirecții de output/append.

## 🔹 ****treenodes/tree\_parse\_args.c****

* **Funcții:**
  + parse\_args(t\_node \*cmd\_node, t\_token \*tokens)
* **Scop:**  
  Extrage doar argumentele valide (WORD), ignorând redirecțiile.

## 🔹 ****execute/execute\_redir\_errors.c****

* **Funcții:**
  + check\_redirection\_syntax(t\_token \*tokens)
* **Scop:**  
  Verifică dacă fiecare <, >, <<, >> este urmat corect de un token de tip WORD.  
  Dacă nu, afișează mesaj de eroare și oprește execuția.

## 🔹 ****execute/execute\_redir\_input.c****

* **Funcții:**
  + int execute\_open\_input\_file(t\_node \*node)
* **Scop:**  
  Deschide fișierul pentru redirecția <.  
  Parcurge nodurile node->left pentru a căuta inputurile multiple.

## 🔹 ****execute/execute\_redir\_output.c****

* **Funcții:**
  + int execute\_open\_output\_file(t\_node \*node)
  + static int open\_output\_target\_file(char \*file, t\_token\_type type)
* **Scop:**  
  Deschide fișierul pentru redirecțiile > (scriere) sau >> (append).  
  Parcurge nodurile node->right.

## 🔹 ****execute/execute\_redir\_heredoc.c****

* **Funcții:**
  + int execute\_process\_heredocs(t\_node \*node)
  + int execute\_handle\_heredoc(const char \*limiter)
* **Scop:**  
  Pregătește toate heredoc-urile înainte de execuție.  
  Creează pipe-uri temporare în care salvează inputul utilizatorului până la delimitator.

## 🔹 ****execute/execute\_redirections.c****

* **Funcții:**
  + prepare\_redirections(t\_node \*node)
  + redirect\_input(t\_node \*node)
  + redirect\_output(t\_node \*node)
* **Scop:**  
  Aplică toate redirecțiile:
  + << (heredoc) → dup2() pe heredoc\_fd.
  + < → dup2() după deschiderea fișierului de input.
  + > și >> → dup2() după deschiderea fișierului de output.

Închide descriptorii după aplicarea redirecțiilor pentru a preveni leaks.

## 🔹 ****execute/execute\_word\_utils.c****

* **Funcții:**
  + int execute\_with\_redir(t\_shell \*sh, char \*path, char \*\*args, t\_node \*node)
* **Scop:**  
  Salvează STDIN/STDOUT, aplică prepare\_redirections(node), apoi execută comanda cu execve().  
  După execuție, restaurează descriptorii originali.

## 🔹 ****treenodes/free\_tree.c****

* **Funcții:**
  + void free\_tree(t\_node \*node)
* **Scop:**  
  Eliberează memoria alocată pentru arborele de execuție, inclusiv args[], value, heredoc\_fd, left, right.

## 🔹 ****utils/init\_shell.c****

* **Modificare:**
  + În void free\_shell (t\_shell \*sh) a fost adăugat:

if (sh→root)  
 free\_tree(sh->root);

* **Scop:**  
  Eliberează corect memoria asociată arborelui la terminarea shell-ului.

**În** prepare\_redirections():

Întâi sunt procesate toate << (heredocs),

Abia apoi se aplică <, >, >> normale.

**În** execute\_process\_heredocs():

* Se creează pipe pentru fiecare <<.
* Se scrie input-ul dat de utilizator linie cu linie în pipe.

**În** execute\_open\_output\_file():

* Se diferențiază > (creare + suprascriere) de >> (append) prin O\_TRUNC vs O\_APPEND.
* **Toate erorile** sunt tratate corect:
* La eroare în redirecționare, execuția este oprită înainte de fork.

**📚 DOCUMENTAȚIE FINALĂ ACTUALIZATĂ – Pipe-uri |, Expansiuni $, Semnale**

# **✅ 1. Implementarea Pipe-urilor (|)**

## 📌 Obiectiv:

Implementarea suportului pentru operatorul | astfel încât output-ul unei comenzi să fie input-ul următoarei.

## 📁 Fișiere și scopul lor

| Folder | Fișier | Scop |
| --- | --- | --- |
| src/parse/ | parse\_pipeline.c | Detectează ` |
| src/parse/ | create\_pipe\_node.c | Creează noduri t\_node pentru pipe-uri (TOK\_PIPE). |
| src/execute/ | execute\_pipe.c | Gestionează execuția pipe-urilor: creează pipe(), fork()-uri și redirecționează STDIN/STDOUT. |
| src/execute/ | execute\_pipes\_utils.c | Funcții utilitare: create\_pipe\_or\_exit() și close\_pipe(). |
| src/minishell/ | minishell\_loop.c | Apelează build\_parse\_tree() pentru a construi arborele corect. |

## 🔹 Funcții principale implicate:

* parse\_pipeline(t\_token \*\*tokens) - (parse\_pipeline.c)  
  → Caută |, împarte lista de tokeni, creează subarbori stânga și dreapta.
* handle\_pipe\_split() - (parse\_pipeline.c)  
  → Rupe lista de tokeni la |, creează t\_node de tip TOK\_PIPE conectând cele două subcomenzi.
* create\_pipe\_node(t\_node \*left, t\_node \*right) - (create\_pipe\_node.c)  
  → Creează un nod TOK\_PIPE care leagă stânga/dreapta.
* execute\_pipe\_node(t\_shell \*sh, t\_node \*node) - (execute\_pipe.c)  
  → Creează pipe(), fork()-ează două procese, redirecționează STDOUT și STDIN, așteaptă finalizarea proceselor.
* create\_pipe\_or\_exit(int pipe\_fd[2]) - (execute\_pipes\_utils.c)  
  → Creează un pipe sau termină cu eroare la eșec.
* close\_pipe(int pipe\_fd[2]) - (execute\_pipes\_utils.c)  
  → Închide ambele capete ale unui pipe.
* process\_and\_execute\_line(t\_shell \*sh, const char \*input) (minishell\_loop.c)  
  → Apelează build\_parse\_tree() în loc de parse\_command() pentru a suporta pipe-uri.

## 🔹 Logica execuției:

1. **Parsing:**
   * La întâlnirea unui |, inputul se împarte în două subcomenzi.
   * Se construiește un nod TOK\_PIPE.
2. **Execuție:**
   * Se creează pipe().
   * Se fork() două procese:
     + Primul procesează comanda din stânga → redirecționează STDOUT în pipe[1].
     + Al doilea procesează comanda din dreapta → citește STDIN din pipe[0].
   * Se închid capetele nepotrivite ale pipe-ului.
   * Shell-ul așteaptă terminarea ambelor procese (waitpid()).

# ✅ 2. Handle Environment Variables ($VAR) și Exit Code ($?)

## 📌 Obiectiv:

Expandarea corectă a $VAR și $? în input.

## 📁 Fișiere și scopul lor

| Folder | Fișier | Scop |
| --- | --- | --- |
| src/expander/ | expand\_dollar.c | Expandarea variabilelor de mediu și a codului de ieșire. |
| src/expander/ | expand\_dollar\_utils.c | Funcții ajutătoare pentru expandarea $VAR și $?. |
| src/expander/ | expand\_protect\_single\_quotes.c | Previne expandarea în '...'. |
| src/lexer/ | input\_processing.c | Aplică expand\_line() înainte de tokenizare. |
| src/minishell/ | minishell\_loop.c | Actualizează sh->exit\_code. |

## 🔹 Funcții principale implicate:

* expand\_line(t\_shell \*sh, const char \*input)  
  → Protejează '...', aplică expansiunea $VAR și $?.
* expand\_env\_variables(t\_shell \*sh, const char \*input)  
  → Caută $, înlocuiește cu valoarea din env\_list sau exit\_code.
* handle\_variable(t\_shell \*sh, const char \*str, int \*i)  
  → Expandează variabile standard $VAR.
* handle\_exit\_code(t\_shell \*sh, int \*i)  
  → Expandează codul de exit $?.
* protect\_dollar\_in\_single\_quotes(const char \*input)  
  → Evită expansiunea $ în ghilimele simple '...'.

## 🔹 Logica execuției:

1. Înainte de tokenizare, linia este trecută prin expand\_line().
2. Se expandează variabilele de mediu sau codul de exit ($?).
3. Expansiunea $ este ignorată în ghilimele simple ('...').

# ✅ 3. Gestionarea Semnalelor (Ctrl-C, Ctrl-D, Ctrl-)

## 📌 Obiectiv:

Comportament interactiv corect la semnale din terminal:

| Semnal | Comportament |
| --- | --- |
| **Ctrl-C** (SIGINT) | Afișează prompt nou, linie goală. |
| **Ctrl-D** (EOF) | Iese din shell dacă linia este goală. |
| **Ctrl-\** (SIGQUIT) | Ignorat complet, nu face nimic. |

## 📁 Fișiere și scopul lor

| Folder | Fișier | Scop |
| --- | --- | --- |
| src/minishell/ | signal.c | Gestionează semnalele corect. |
| src/minishell/ | minishell\_loop.c | Detectează readline() == NULL pentru Ctrl-D. |
| src/minishell/ | main.c | Inițializează handlers la pornire. |

## 🔹 Funcții principale implicate:

* handle\_signals() - (signal.c)  
  → Setează handler personalizat pentru SIGINT și ignoră SIGQUIT.
* sigint\_handler(int sig) - (signal.c)  
  → Pe Ctrl-C: linie nouă și prompt nou.
* run\_shell\_loop() **și** process\_and\_execute\_line() - (minishell\_loop.c)  
  → Detectează Ctrl-D (line == NULL) și închide shell-ul.
* main()  
  → Apelează handle\_signals() la început.

# 🔥 Sumar general

| Funcționalitate | Fișiere principale | Funcții principale |
| --- | --- | --- |
| Pipe-uri ` | ` | parse\_pipeline.c, execute\_pipe.c, execute\_pipes\_utils.c, minishell\_loop.c |
| Expansiuni $VAR, $? | expand\_dollar.c, expand\_dollar\_utils.c, expand\_protect\_single\_quotes.c, input\_processing.c | expand\_line(), expand\_env\_variables(), handle\_variable(), handle\_exit\_code() |
| Semnale | signal.c, minishell\_loop.c, main.c | handle\_signals(), sigint\_handler(), run\_shell\_loop() |

# **📚 Builtin-uri (cu tratament special pentru export)**

## 📁 Fișiere relevante și funcționalități

### 🔹 builtins/builtin\_export.c

* **Funcție:** int builtin\_export(t\_shell \*sh, char \*\*args, int fd\_out)
* **Scop:**
  + Dacă nu are argumente → afișează env cu builtin\_env().
  + Cu argumente → adaugă/înlocuiește perechi key=value.
  + Folosește remove\_surrounding\_quotes() pentru a curăța valorile.

### 🔹 src/minishell/handle\_export.c​

* **Funcții:**
  + bool is\_export\_command(const char \*input)  
    → Verifică dacă linia este un export direct.
  + void process\_export\_directly(t\_shell \*sh, const char \*input)  
    → Sparge inputul în args[], apelează builtin\_export() direct fără tokenizare.

### 🔹 src/minishell/minishell\_loop.c​ În process\_and\_execute\_line():

if (is\_export\_command(input))

{

process\_export\_directly(sh, input);

return;

}

* + Detectează export din start și îl tratează separat.
  + Evită expandare de variabile, globbing sau quotes.

### 🔹 src/minishell/process\_for\_export.c​

* **Funcție:** void process\_non\_export(t\_shell \*sh, const char \*input)
* **Scop:**
  + Procesează orice linie care **nu** este export.
  + Include:
    - expand\_line()
    - tokenize\_input()
    - expand\_args\_globbing()
    - build\_parse\_tree() → execute\_parsed\_tree()

## 🧠 De ce este export tratat separat?

* Pentru a respecta regula că export VAR=value:
  + **nu trebuie** să fie afectat de expansiunea $VAR sau globbing (\*).

## 🔧 Conectare cu restul proiectului

| Fază | Builtin export |
| --- | --- |
| Detectare | is\_export\_command() |
| Execuție directă | process\_export\_directly() |
| Fără tokenizare/globbing | Da |
| Restul comenzilor | merg prin process\_non\_export() |

**✅ Manual Testing – Documentație (Partea Mandatory implementată)**

### ****✅ 1. Compile****

(implicit testat – shell-ul pornește cu ./minishell fără erori)

**✅ 2. Simple Command & global variables**

**minishell$ echo Hello**

**raspuns: Hello**

**minishell$ ls**

**raspuns: conținutul directorului curent**

**minishell$ pwd**

**raspuns: /calea/curentă**

**# Test global: minishell$ ↑ (săgeată sus) → reapare ultima comandă (ex: echo Hello)**

**✅ 3. Arguments**

**minishell$ echo Hello world**

**raspuns: Hello world**

**minishell$ echo -n Hello**

**raspuns: Hello (fără newline)**

**✅ 4. echo**

**echo Hello world**

**→ ✅ Output: Hello world\n**

**echo -n Hello**

**→ ✅ Output: Hello (fără newline)**

**echo -n -n Hello**

**→ ✅ Output: Hello**

**echo -n**

**→ ✅ Output: (fără output, nici newline)**

**echo \***

**→ ✅ Output: Makefile minishell.c src libft …**

**✅ 5. exit**

**exit**

**→ ✅ Iese din shell cu codul sh->exit\_code**

**exit 42**

**dupa exit se face in bash comanda - echo $?**

**→ ✅ 42**

**exit 256**

**dupa exit se face in bash comanda - echo $?**

**→ ✅ 0 (256 % 256)**

**exit -1**

**dupa exit se face in bash comanda - echo $?**

**→ ✅ 255 (-1 % 256)**

**exit 1 2**

**→ ❌ exit: too many arguments**

**exit hello**

**→ ❌ exit: numeric argument required (exit code 255)**

**✅ 6. Return value of a process**

**echo hello | grep hello**

**echo $?**

**→ ✅ 0**

**echo hello | grep "not\_found"**

**echo $?**

**→ ✅ 1**

### **✅ **7. Signals (Ctrl+C, Ctrl+D, Ctrl+)****

* **Ctrl+C**: se afișează un prompt nou, fără ieșire din shell.
* **Ctrl+D**: închide shell-ul dacă linia este goală.
* **Ctrl+\**: ignorat complet, shell-ul continuă.

✅ **8. Double Quotes**

**echo "Hello $USER"**

**→ ✅ Hello roxana**

**echo "\*"**

**→ ✅ \***

**✅ 9. Single Quotes**

**echo '$USER'**

**→ ✅ $USER**

**echo '$?'**

**→ ✅ $?**

**echo 'Hello**

**→ ❌ unclosed single quote**

**echo "Hello**

**→ ❌ unclosed double quote**

**✅ 10. env**

**export A=1 B=2**

**si apoi se scrie -env (enter)**

**→ ✅ Afișează A=1 și B=2**

**unset A**

**si apoi se scrie -env (enter)**

**→ ✅ Nu mai apare A**

**env VAR=123**

**→ ❌ env: too many arguments**

**env -i**

**→ ❌ env: too many arguments**

**✅ 11. export**

**export TEST\_VAR=hello**

**env | grep TEST\_VAR**

**→ ✅ TEST\_VAR=hello**

**export VAR1="Hello"**

**export VAR2="World"**

**env | grep VAR**

**→ ✅ VAR1=Hello, VAR2=World**

**export MY\_NAME="roxana"**

**env | grep MY\_NAME**

**→ ✅ MY\_NAME=roxana**

**export MY\_CITY='Trier'**

**env | grep MY\_CITY**

**→ ✅ MY\_CITY=Trier**

**export TEST\_VAR="new value"**

**→ ✅ valoarea e înlocuită**

**export EMPTY\_VAR=**

**→ ✅ EMPTY\_VAR= (valoare goală)**

**export GREETING="Hello World"**

**→ ✅ GREETING=Hello World**

**export 42INVALID=value**

**→ ❌ not a valid identifier**

**export VAR1=one VAR2=two VAR3=three**

**→ ✅ toate sunt setate**

**export VAR1="first"**

**export VAR1="second" VAR4="newone"**

**→ ✅ VAR1=second, VAR4=newone**

**export**

**→ ✅ Afișează toți, sortat sau nu**

**✅ 12. unset**

**export VARTEST=hello**

**unset VARTEST**

**env | grep VARTEST**

**→ ✅ Nu mai există**

**export VAR1=one VAR2=two**

**unset VAR1 VAR2**

**env | grep VAR**

**→ ✅ Nu mai apar**

**unset VARINEXISTENT**

**→ ✅ Nu dă eroare**

**unset**

**→ ✅ Nu face nimic**

**unset 123ABC**

**→ ❌ Ignorat**

**unset A=B**

**→ ❌ Ignorat**

**✅ 13. cd**

**cd /**

**→ ✅ Navighează în root**

**cd ..**

**→ ✅ Navighează în părinte**

**cd ./folder**

**→ ✅ Navighează relativ**

**cd**

**→ ❌ Nu e suportat fără argument**

**cd folder\_inexistent**

**→ ❌ Afișează eroare**

**✅ 14. pwd**

**pwd**

**→ ✅ Afișează directorul curent**

**✅ 15. Relative Path**

**cd ..**

**→ ✅ Mergi în părinte**

**cd ./folder**

**→ ✅ Mergi relativ**

**✅ 16. Environment Path**

**/bin/ls**

**→ ✅ execută comanda**

**./script.sh**

**→ ✅ Hello from real script**

**echo $USER**

**→ ✅ roxana**

**echo "$USER"**

**→ ✅ roxana**

**echo '$USER'**

**→ ✅ $USER**

**echo $NOT\_SET**

**→ ✅ (linie goală)**

**echo $**

**→ ✅ $**

**✅ 17. Redirection**

**echo hello > test.txt**

**cat test.txt**

**→ ✅ hello**

**echo First >> test.txt**

**echo Second >> test.txt**

**cat test.txt**

**→ ✅ hello\nFirst\nSecond**

**echo "Line from file" > test\_input.txt**

**cat < test\_input.txt**

**→ ✅ Line from file**

**cat << EOF**

**line 1**

**line 2**

**EOF**

**→ ✅ line 1\nline 2**

**cat < does\_not\_exist.txt**

**→ ❌ does\_not\_exist.txt: No such file or directory**

**touch restricted.txt**

**chmod 000 restricted.txt**

**echo "test" > restricted.txt**

**→ ❌ Permission denied**

**echo "10 20 30" > nums.txt**

**cat < nums.txt > result.txt**

**cat result.txt**

**→ ✅ 10 20 30**

**✅ 18. Pipes**

**echo Hello | wc**

**→ ✅ 1 1 6**

**ls | wc -l**

**→ ✅ număr fișiere**

**ls | grep minishell**

**→ ✅ fișiere ce conțin minishell**

**ls | grep minishell > output.txt**

**cat output.txt**

**→ ✅ rezultatul scris în fișier**

**cat << EOF | grep test**

**line1**

**test line**

**line2**

**EOF**

**→ ✅ test line**

**echo test123 > in.txt**

**cat < in.txt | grep test**

**→ ✅ test123**

**echo abc | grep zzz**

**→ ✅ (linie goală)**

**echo "a b c" | tr ' ' '\n' | grep b**

**→ ✅ b**

**echo Hello |**

**→ ✅ (comandă incompletă, nu face nimic)**

**✅ 19. Go Crazy and history**

**minishell$**

**→ promptul apare corect**

**comenzi goale → nu dau erori**

**space/tab → ignorate**

**săgeată sus → ultima comandă reapare**

****valgrind --leak-check=full --track-origins=yes --show-leak-kinds=all ./minishell****