# **Minishell documentation**

**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.**

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

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)

│

│ ├── 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)

│

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

│

├── Makefile

└── script.sh

| 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. |

### ✅ ****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)

### In **minishell.h,** care este “scheletul” proiectului.

### **Contine:**

Definiții de tipuri și structuri de date (cum arată datele noastre)

* Enum-uri (pentru tipuri de tokeni)
* Prototipurile funcțiilor (spune ce funcții există și unde să le găsească compilatorul)

## 🧩 În minishell, fiecare token este:

* un struct t\_token
* are un content (string-ul efectiv)
* are un type (ex: TOK\_WORD, TOK\_PIPE, TOK\_REDIR\_OUT, etc.)
* opțional: was\_quoted sau quote\_type → dacă tokenul provine dintr-un string cu ' sau "

## 🛠️ Ce face funcția tokenize\_input()?

Funcția tokenize\_input() din input\_processing.c:

* primește linia de input de la utilizator
* parcurge caracter cu caracter
* creează tokeni (adică elemente logice) în funcție de:
  + spații
  + quotes
  + caractere speciale ca |, <, >

### ✅ Tokenii sunt esențiali pentru:

| Acțiune | De ce ai nevoie de tokeni |
| --- | --- |
| Execuția comenzilor | Să știi unde e comanda și care sunt argumentele |
| Redirecții | Să știi că > cere un fișier după |
| Pipes | Să știi unde se termină o comandă și începe alta |
| Expansiuni | Să știi ce parte e $USER și dacă trebuie expandată |
| Quotes | Să știi dacă ceva a fost între ' sau " |

✅ **STRUCTURI CHEIE**

**🔹 enum e\_token\_type**

**typedef enum e\_token\_type**

**{**

**TOK\_WORD = 1,**

**TOK\_PIPE,**

**TOK\_REDIR\_IN,**

**TOK\_REDIR\_OUT,**

**TOK\_APPEND,**

**TOK\_HEREDOC**

**} t\_token\_type;**

**Folosit pentru a identifica ce tip de token avem:**

* + ls, echo, cat → TOK\_WORD
  + | → TOK\_PIPE
  + <, >, >>, << → redirecții

🟩 **Legătură cu PDF**: necesar pentru execuție, pipe, redirecții

**🔹 struct s\_token**

### typedef struct s\_token

{

t\_token\_type type;

char \*content;

bool was\_quoted;

struct s\_token \*next;

struct s\_token \*prev;

} t\_token;

**Reprezintă un token rezultat din linia de input**

* content: textul efectiv (ls, >, "Salut" etc.)
* type: tipul de token (WORD, PIPE, etc.)
* was\_quoted: dacă a fost între '...' sau "..."

**🟩 Folosit în:**

* tokenize\_input() → în input\_processing.c
* expand\_args\_globbing\_from\_tokens() → dacă was\_quoted == true, nu facem globbing

**🔹 struct s\_node**

**typedef struct s\_node**

**{**

**t\_token\_type type;**

**char \*value;**

**char \*\*args;**

**struct s\_node \*left;**

**struct s\_node \*right;**

**} t\_node;**

**Reprezintă nodul în arborele de execuție**

* value → comanda (ls)
* args → argumentele (-l, -a)
* left și right → legătura între comenzi dacă sunt cu |

🟩 **Pregătit pentru redirecții și pipe-uri** (vom folosi la exec tree)

**🔹 struct s\_env**

### **typedef struct s\_env**

**{**

**char \*key;**

**char \*value;**

**struct s\_env \*next;**

**struct s\_env \*prev;**

**} t\_env;**

**Reprezintă o variabilă de mediu (ex: USER=roxana)**

* Păstrăm toate variabilele în env\_list

🟩 Folosit pentru expansiuni $USER, $PATH, export, unset (parțial)

**🔹 struct s\_shell**

**typedef struct s\_shell**

**{**

**t\_node \*root;**

**t\_token \*tokens;**

**t\_env \*env\_list;**

**int exit\_code;**

**char \*line\_input;**

**} t\_shell;**

**Este structura principală a shell-ului:**

* tokens → lista de tokeni generați de input
* root → arborele de execuție (comanda, args, pipe etc.)
* env\_list → lista variabilelor de mediu
* exit\_code → codul de ieșire pentru $?
* line\_input → linia brută introdusă de user

**Este transmisă peste tot: execuție, parsing, expansiuni etc.**

## **🔁 FUNCȚII IMPORTANTE DECLARATE (prototypes)**

* main.c → main, banner, create\_env\_copy
* minishell\_loop.c → loop principal
* expand\_dollar.c, expand\_asterisk.c → expansiuni
* execute\_word.c, fork\_and\_execute.c → execuție
* check\_input\_errors.c → quotes neînchise, caractere interzise
* resolve\_command.c → găsirea executabilului în PATH
* init\_shell.c, signal.c → inițializare și semnale
* tokenize\_input() → tokenizer (sparge input în tokeni)

**Structura minishell.h definește tot ce ține de logica internă a shellului nostru:**

* **Tokenii** sunt bucăți logice din comanda introdusă
* **Tokenizarea** ne permite să înțelegem fiecare parte din input (comandă, argument, redirecție)
* Avem o structură principală t\_shell care centralizează tot contextul shell-ului
* Totul este pregătit modular: putem adăuga pipe, redirecții, exec tree fără să refacem structurile.

### **Rezumat:**

**✅ CERINȚE DIN PDF – IMPLEMENTARE**

### **🟩 1. **Prompt afișat pentru input****

* **Fișiere implicate**:
  + src/minishell/minishell\_loop.c → get\_prompt(), run\_shell\_loop()
  + src/minishell/input\_processing.c → readline()

### **🟩 2. **Istoric funcțional****

* **Fișiere**:
  + readline() și add\_history() sunt apelate în   
    run\_shell\_loop()

### **🟩 3. **Căutare și execuție executabil (PATH/absolut/relativ)****

* **Fișiere**:
  + parse/resolve\_command.c → find\_command\_path()
  + execute/execute\_word.c → execute\_word()
  + execute/execute\_word\_utils.c → fork\_and\_execute(), convert\_env\_to\_array()

### **🟩 4. **Doar o variabilă globală pentru semnal****

* **Detalii**: main.c definește pid\_t g\_child\_pid, conform cerinței

### 🟩 5. ****Quotes Handling****

#### 🔹 Ghilimele simple '...' blochează interpretarea

* **Fișiere**:
  + expander/expand\_protect\_single\_quotes.c → protect\_dollar\_in\_single\_quotes()
  + Integrată în expand\_line() din expand\_dollar.c

#### 🔹 Ghilimele duble "..." permit doar expansiune $

* **Fișiere**:
  + expand\_quotes\_dollar.c → remove\_double\_quotes\_preserve\_dollar()

### 🟩 6. ****Expansiune de variabile și globbing****

* **Fișiere**:
  + expand\_dollar.c → expand\_env\_variables()
  + expand\_args\_utils.c → expand\_args\_globbing\_from\_tokens()
  + expand\_asterisk.c, expand\_globbing\_utils.c, expanding\_utils.c
  + utils\_tokens\_asterisk.c → extract\_args\_from\_tokens()

### 🟩 7. ****Tokenizare și păstrarea quote flags****

### ****Fișiere****:

* + input\_processing.c → tokenize\_input(), was\_quoted flag
  + split\_args\_preserving\_quotes.c → păstrează quotes pentru parsing corect

### 🟩 8. ****Structură shell + variabile de mediu****

* **Fișiere**:
  + init\_shell.c → init\_shell(), create\_env\_list(), free\_shell()
  + utils\_env\_strjoin.c → get\_env\_value(), ft\_strjoin\_three()

### 🟩 9. ****Semnale (****Ctrl+C****,**** Ctrl+\****)****

* **Status**: ✅ Complet
* **Fișiere**:
  + signal.c → handle\_signals(), sigint\_handler()

## **✅ **Rezumat: Implementarea redirecțiilor în Minishell****

### 📁 ****Fișiere create și scopul lor****

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

* **Funcție:** int execute\_open\_input\_file(t\_node \*node);
* **Scop:** Deschide fișierul pentru redirecție <. Parcurge nodurile din node->left.

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

* **Funcții:**
* static int open\_output\_target\_file (char \*file, t\_token\_type type)
* int execute\_open\_output\_file(t\_node \*node);
* **Scop:** Deschide fișierul pentru redirecție > sau >>. Parcurge node->right.
* Funcția a fost împărțită în două pentru a respecta regula de max. 25 linii:
  + open\_output\_target\_file(...) (static helper)

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

* **Funcții:**
  + int execute\_process\_heredocs(t\_node \*node);

**Parcurge recursiv arborele de execuție (**t\_node**) și caută noduri de tip** TOK\_HEREDOC**.**

* Pentru fiecare astfel de nod, apelează execute\_handle\_heredoc() și salvează heredoc\_fd în nodul respectiv.
* Se apelează **înainte de execuția comenzii**, pentru a pregăti toate heredoc-urile.
  + **int execute\_handle\_heredoc(const char \*limiter);**

Creează un **pipe temporar** și citește linie cu linie de la utilizator folosind readline("> ").

* Scrie fiecare linie în pipe până când utilizatorul introduce limiter (ex: END).
* Returnează file descriptor-ul pipe-ului pentru a fi folosit ca **STDIN** în execuția comenzii.
* Este folosit pentru redirecția << (heredoc).

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

### • Funcții:

#### int prepare\_redirections(t\_node \*node);

* Pregătește toate redirecțiile pentru o comandă:
  + << (heredoc) — aplică dup2() pe heredoc\_fd
  + < — deschide fișierul și redirecționează în STDIN
  + > / >> — deschide fișierul și redirecționează în STDOUT
* Apelează două funcții auxiliare:
  + redirect\_input() → pentru <
  + redirect\_output() → pentru > și >>
* Toate redirecțiile sunt aplicate **prin** dup2(), iar file descriptorii sunt închiși corect după redirecționare.

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

* **Funcție:** int check\_redirection\_syntax(t\_token \*tokens);
* **Scop:** Verifică dacă tokenii de tip <, >, >>, << sunt urmați de un TOK\_WORD. Dacă nu, afișează mesaj de eroare.

#### ****🔹** execute/execute.c**

* **Funcție:** int execute\_command(t\_shell \*sh, t\_node \*node);
* **Scop:** Integrează toată execuția:
* Procesează heredoc-uri
* Pregătește redirecțiile
* Caută path-ul comenzii
* Execută comanda cu execute\_with\_redir(...)

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

* **Funcție:** int execute\_with\_redir(t\_shell \*sh, char \*path, char \*\*args, t\_node \*node);
* **Scop:** Face prepare\_redirections(node) și apoi apelează fork\_and\_execute(...)

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

* **Funcții:** Funcții ajutătoare pentru execuție și redirecții, cum ar fi:
  + Închiderea file descriptorilor
  + Utilitare pentru debug
* **Scop:** Separarea logicii auxiliare pentru redirecții și execuție, pentru a menține codul clar și scurt.

### **📁 **Fișiere suport pentru parsing****

#### 🔹 treenodes/parse\_command.c

* **Funcție:** t\_node \*parse\_command(t\_token \*\*tokens);
* **Scop:** Construiește nodul t\_node din lista de t\_token:
  + Salvează args[]
  + Leagă redirecțiile (<, >, >>, <<) în left și right

#### ****🔹** treenodes/free\_tree.c**

* **Funcție:** void free\_tree(t\_node \*node);
* **Scop:** Eliberează recursiv arborele de execuție:
  + args[], value, heredoc\_fd, left, right, nodul în sine

**Modificari in utils/init\_shell.c**

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

if (sh->root)

free\_tree(sh->root);

## **🔗 **Lanț logic al execuției cu redirecții****

1. check\_redirection\_syntax(tokens) → verifică validitatea tokenilor

2. parse\_command(&tokens) → creează arborele t\_node (args + redirecții)

3. execute\_command(sh, node): → etapa de execuție

├── execute\_process\_heredocs(node) → procesează toate heredoc-urile

├── prepare\_redirections(node) → aplică toate redirecțiile (<, >, <<, >>)

└── execute\_with\_redir(...) → face fork și execve cu STDIN/STDOUT corect setate

**🔹 Fișiere noi adăugate**

• utils/token\_utils.c – funcții pentru clasificarea tokenilor și tratarea simbolurilor speciale.  
• treenodes/tree\_parse\_args.c – logică refăcută pentru extragerea argumentelor excluzând redirecțiile.  
• execute/execute\_redir\_input.c – deschide fișierele pentru redirecții de intrare (<).  
• execute/execute\_redir\_output.c – deschide fișierele pentru redirecții de ieșire (> și >>).  
• execute/execute\_redirections.c – gestionează toate redirecțiile înainte de execuție.  
• debug\_print\_tokens.c – afișare tokeni pentru depanare (opțional).

**🔹 Implementare redirecționări – Rezumat**

• În tokenizer, tokenii precum <, >, >>, << sunt recunoscuți și marcați ca atare (TOK\_REDIR\_IN, TOK\_APPEND etc).  
• Funcția parse\_command construiește arborele de execuție, plasând redirecțiile în câmpurile left (input/heredoc) și right (output/append).  
• Înainte de execuție, prepare\_redirections() verifică dacă există redirecții și face open() + dup2().  
• Pentru heredoc, un descriptor temporar este setat anterior în node->heredoc\_fd.  
• Execuția finală se face cu fork\_and\_execute(), după ce STDIN/STDOUT au fost redirectate corect.  
• Astfel, comenzi precum `echo hello > file.txt` vor funcționa fără să scrie în terminal.

### 🧩 Funcționalitate: Implementarea pipe-urilor (|)

#### ✅ Obiectiv

Să se implementeze suportul pentru operatorul | în Minishell, astfel încât **ieșirea unei comenzi** să fie **transmisă ca intrare către următoarea**. Acest lucru permite concatenarea comenzilor în stil Unix:

ls | grep minishell | wc -l

#### 🛠️ Pașii implementării

1. **Construirea arborelui de parsare:**
   * S-a creat funcția recursivă parse\_pipeline() care:
     + Detectează tokenii de tip |.
     + Împarte lista de tokeni în stânga și dreapta față de |.
     + Creează un arbore binar (t\_node) cu noduri de tip TOK\_PIPE.
     + Subarborele stâng/drept reprezintă comenzi înainte/după |.
2. **Crearea nodului de tip pipe:**
   * S-a adăugat funcția create\_pipe\_node() care:
     + Inițializează un nod de tip TOK\_PIPE.
     + Leagă comenzile stânga/dreapta ca sub-arbori.
3. **Executarea comenzilor cu pipe:**
   * S-a implementat execute\_pipe\_node():
     + Creează o conductă (pipe) cu pipe\_fd[2].
     + Creează doi copii cu fork():
       - Primul procesează comanda din stânga și redirecționează stdout → pipe.
       - Al doilea procesează comanda din dreapta și redirecționează stdin ← pipe.
     + Se folosesc dup2() și close() pentru gestionarea corectă a descriptorilor.
     + Se așteaptă terminarea ambelor procese și se actualizează codul de ieșire.
4. **Executarea în procesele copil:**
   * În execute\_child\_process():
     + Se verifică dacă comanda este în stânga sau în dreapta pipe-ului.
     + Se aplică redirecțiile corespunzătoare.
     + Se execută comanda prin evaluate\_execution().
5. **Integrarea în bucla principală:**
   * În process\_and\_execute\_line(), apelul parse\_command() a fost înlocuit cu build\_parse\_tree().
   * Noua funcție gestionează atât comenzi simple, cât și redirecționări și pipe-uri, fără a afecta funcționalitățile deja implementate (quotes, variabile, redirecționări).

### 📂 ****Folderul src/parse/ (parsare arbori)****

| Fișier | Rol |
| --- | --- |
| parse\_pipeline.c | Parcurge lista de tokeni și construiește recursiv arborele de comenzi legate prin ` |
| create\_pipe\_node.c | Creează un nod t\_node de tip TOK\_PIPE și conține și funcția build\_parse\_tree() care pornește procesul de parsare. |

### 📂 ****Folderul**** src/execute/ ****(executare comenzi)****

| Fișier | Rol |
| --- | --- |
| execute\_pipe.c | Gestionează execuția comenzilor separate de ` |
| execute\_pipes\_utils.c | Conține funcțiile ajutătoare pentru pipe-uri: create\_pipe\_or\_exit() și close\_pipe(). |

### 📂 ****Folderul**** src/minishell/ ****(buclă principală)****

| Fișier | Rol |
| --- | --- |
| minishell\_loop.c | În process\_and\_execute\_line() a fost înlocuit parse\_command() cu build\_parse\_tree() pentru a permite suportul pipe. |

## ✅ 1. ****Handle environment variables ($VAR)****

### 🔍 Descriere:

Shell-ul trebuie să poată expanda variabile de mediu scrise sub forma $VAR.

De exemplu:

echo $USER → va afișa roxana

De asemenea:

* Variabilele din interiorul **ghilimelelor duble** sunt expandate (echo "Salut $USER" → Salut roxana).
* Variabilele din interiorul **ghilimelelor simple** **nu trebuie expandate** (echo '$USER' → $USER).

### 🗂️ Fișiere implicate:

| Fișier | Ce conține |
| --- | --- |
| expand\_dollar.c | funcția principală expand\_env\_variables() și expand\_line() care aplică expansiunea variabilelor |
| expand\_dollar\_utils.c | funcții ajutătoare pentru expansiune: handle\_variable(), handle\_char() |
| expand\_protect\_single\_quotes.c | protejează $ în ghilimele simple folosind caracterul special \x01 |
| input\_processing.c | folosește expand\_line() înainte de tokenizare |
| minishell.h | conține prototipurile funcțiilor și structurile necesare (t\_shell, t\_env) |

## ✅ 2. ****Handle**** $? ****(codul de ieșire al ultimei comenzi)****

### 🔍 Descriere:

Shell-ul trebuie să înlocuiască $? cu **valoarea de returnare** a ultimei comenzi executate (inclusiv în cazul unui pipe).  
Exemplu:

ls | grep ceva\_inexistent

echo $? # → va afișa 1

Această variabilă trebuie să reflecte corect:

* 0 dacă comanda a fost cu succes
* 1 sau 127 în caz de eroare, etc.

### 🗂️ Fișiere implicate:

| Fișier | Ce conține |
| --- | --- |
| expand\_dollar.c | folosește sh->exit\_code în funcția expand\_env\_variables() |
| expand\_dollar\_utils.c | funcția handle\_exit\_code() care returnează valoarea lui $? sub formă de string |
| expand\_protect\_single\_quotes.c | asigură că $? nu e expandat în ghilimele simple |
| input\_processing.c | folosește expand\_line() înainte de a crea tokens |
| minishell\_loop.c | actualizează sh->exit\_code după fiecare comandă executată |
| execute\_utils.c | funcția evaluate\_execution() returnează codul de ieșire pentru comenzi |

### ✅ 3. Handle Ctrl-C, Ctrl-D, and Ctrl-\ (semnale)

🔍 **Descriere:**

Shell-ul trebuie să gestioneze semnalele din modul interactiv exact ca bash:

| Semnal | Comportament așteptat | Ce trebuie să facă minishell |
| --- | --- | --- |
| Ctrl-C (SIGINT) | Afișează o linie nouă + prompt nou | ✅ minishell$ apare din nou |
| Ctrl-D (EOF) | Iese din shell dacă linia este goală | ✅ minishell se închide |
| Ctrl-\ (SIGQUIT) | Nu face nimic, ignorat complet | ✅ minishell continuă fără efect |

### 🗂️ Fișiere implicate:

| Fișier | Ce conține |
| --- | --- |
| signal.c | funcția handle\_signals() care setează handlerul pentru SIGINT și ignoră SIGQUIT |
| minishell\_loop.c | apelează readline() și verifică dacă line == NULL (Ctrl-D) pentru a ieși curat din shell |
| main.c | apelează handle\_signals() la pornirea shell-ului |

## ✅ Builtins implementate

Shell-ul implementează următoarele builtin-uri, conform cerinței proiectului:

* echo cu opțiunea -n
* cd cu o cale relativă sau absolută
* pwd fără opțiuni
* export fără opțiuni
* unset fără opțiuni
* env fără opțiuni sau argumente
* exit fără opțiuni

## 🗂️ Fisiere implicate și funcțiile implementate

### 🔹 builtins/builtin\_echo.c

* int builtin\_echo(char \*\*args, int fd\_out);
  + Implementează echo, cu suport pentru -n și scrierea în fd\_out pentru redirecționare.

### 🔹 builtins/builtin\_cd.c

* int builtin\_cd(t\_shell \*sh, char \*\*args);
  + Schimbă directorul curent (absolut/relativ) și actualizează PWD și OLDPWD.

### 🔹 builtins/builtin\_pwd.c

* int builtin\_pwd(char \*\*args, int fd\_out);
  + Afișează directorul curent în fd\_out.

### 🔹 builtins/builtin\_export.c

* int builtin\_export(t\_shell \*sh, char \*\*args, int fd\_out);
  + Adaugă/modifică variabile de mediu. Afișează lista dacă nu are argumente.

### 🔹 builtins/builtin\_unset.c

* int builtin\_unset(t\_shell \*sh, char \*\*args);
  + Elimină variabile de mediu din listă.

### 🔹 builtins/builtin\_env.c

* int builtin\_env(t\_shell \*sh, int fd\_out);
  + Afișează variabilele de mediu fără a accepta argumente.

### 🔹 builtins/builtin\_exit.c

* int builtin\_exit(t\_shell \*sh, char \*\*args);
  + Închide shell-ul cu un cod opțional. Validează argumentul și folosește sh->exit\_code.

## ⚙️ Alte fișiere implicate

### 🔸 execute/execute\_builtin\_utils.c

* int is\_builtin(const char \*cmd);
* int execute\_builtin(t\_node \*cmd\_node, t\_shell \*sh);
  + Detectează și execută builtin-uri.

### 🔸 utils/env\_utils.c

* char \*get\_env\_value(t\_env \*env\_list, const char \*key);
* int update\_env\_var(t\_env \*\*env\_list, const char \*key, const char \*value);
* int remove\_env\_var(t\_env \*\*env\_list, const char \*key);
* void print\_env\_list(t\_env \*env\_list);
* void free\_env\_list(t\_env \*env\_list);
  + Funcții ajutătoare pentru env, export, unset, cd.

## ✅ CONCLUZII GENERALE – PARTEA MANDATORY

* Proiectul minishell a fost implementat respectând toate cerințele din subiectul oficial.
* Componentele cheie ale shell-ului (prompt, input, parsing, quotes, execuție, semnale, expansiuni) sunt implementate modular, în fișiere separate, cu respectarea regulilor Norminette.
* Proiectul gestionează corect cazurile speciale precum:
  + Quotes închise și neînchise
  + Expansiuni condiționate de tipul de ghilimele
  + Caractere interzise
  + Istoric și prompt prin readline
* Execuția comenzilor externe funcționează atât cu PATH cât și cu căi relative sau absolute.

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

🔹 **1. Promptul și istoricul**

minishell$

* Promptul este afișat imediat ce minishell pornește.
* ➜ ./minishell
* minishell$ echo Hello
* rezultat: Hello
* (minishell$ ↑ ) → reapare comanda „echo Hello” din istoric
* ls => enter (apare cotinut folder) apoi => sageata in sus apare ls ( adica iti arata ultima comanda folosita)
* Nu există erori la comenzile goale sau formate din spații/tab-uri
* pwd (arata calea catre folder unde te afli) ex: /home/roxana/42-School-Luxemburg/common\_core/circle\_3/minishell\_project

**🔹 2. Quotes neînchise / caractere interzise**

**minishell$ echo "Hello**

**rezultat: unclosed double quote**

**minishell$ echo 'Hello**

**rezultat: unclosed single quote**

**minishell$ echo Hello ; ls**

**rezultat: forbidden characters (; or \) not allowed**

**minishell$ echo Hello \ World**

**rezultat: forbidden characters (; or \) not allowed**

**✅ Shell-ul detectează erorile și nu execută nimic.**

**🔹 3. Quotes corecte**

**minishell$ echo "Hello $USER"**

**rezultat: Hello roxana**

**minishell$ echo '$USER'**

**rezultat: $USER**

**minishell$ echo '\*'**

**rezultat: \***

**minishell$ echo \***

**rezultat: Makefile minishell.c src libft ...**

**✅ Shell-ul face expansiune doar în quotes duble sau fără quotes. Ghilimelele simple protejează complet.**

**🔹4. PATH și execuție**

**minishell$ /bin/ls**

**rezultat: # lista de fișiere**

**minishell$ ./script.sh  
rezultat: Hello from real script**

**rezultat: # se poate executa un script local dacă are permisiuni**

**minishell$: echo $USER**

**# ✅ Afișează: roxana (sau userul curent)**

**minishell$ echo "$USER"**

**# ✅ Afișează: roxana**

**minishell$ echo '$USER'**

**# ✅ Afișează: $USER (nu trebuie expandat)**

**minishell$ echo $NOT\_SET**

**# ✅ Afișează o linie goală (variabila nu există)**

**minishell$ echo $?**

**rezultat: 0**

**minishell$ echo '$?'**

**rezultat: $?**

**minishell$ echo "$?"**

**rezultat: 0**

**minishell$ echo $**

**rezultat: $**

**minishell$ echo hello | grep hello**

**rezultat hello**

**minishell$ echo $?**

**0 # ✅ Afișează: 0 (comanda a avut succes)**

**minishell$ echo hello | grep “not\_found”**

**minishell$ echo $?**

**1 # ✅ Afișează: 1 (grep nu a găsit nimic)**

**grep: found: No such file or directory**

**echo $?**

**2**

****test redirectii:****

**📝 Redirecționare simplă de ieșire (** > **)**

**testare: echo hello > test.txt**

**cat test.txt**

**Asteptat: Hello**

**➕ Redirecționare dublă (** >> **- append)**

**echo First >> test.txt**

**echo Second >> test.txt**

**cat test.txt**

**✅ Output:**

**#hello**

**# First**

**# Second**

**📥 Redirecționare de intrare (** < **)**

**# Creezi un fișier de input**

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

**# Rulezi un program care citește din el**

**cat < test\_input.txt**

✔️ Așteptat:

Line from file

📋 **Redirecționare cu here-document (** << **)**

**cat << EOF (enter)**

**line 1 (enter)**

**line 2 (enter)**

**EOF (enter)**

✔️ Așteptat:

line 1

line 2

**fisier inexistent:**

cat < does\_not\_exist.txt

✔️ Așteptat:

cat: does\_not\_exist.txt: No such file or directory

🚫 **Redirecționare cu fișier protejat (permission denied)**

**touch restricted.txt**

**chmod 000 restricted.txt**

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

**bash: restricted.txt: Permission denied**

**🧪 Combinație: > și <**

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

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

**cat result.txt**

**10 20 30**

**✅ Teste manuale pipe (**|**)**

### **🔹 **Test 1: Pipe simplu cu două comenzi****

echo Hello | wc

✅ **Rezultat așteptat:**

1 1 6

### 🔹 ****Test 2: Pipe cu listare și numărare linii****

ls | wc -l

✅ **Rezultat așteptat:**  
Număr de fișiere/directoare din folderul curent. De ex : 16

### 🔹 ****Test 3: Pipe cu filtrare grep****

ls | grep minishell

✅ **Rezultat așteptat:**

Afișează fișierele care conțin „minishell” în nume.

### 🔹 ****Test 4: Pipe cu redirecționare la final****

ls | grep minishell > output.txt

cat output.txt

✅ **Rezultat așteptat:**  
Fișierul output.txt conține rezultatul comenzii ls | grep minishell.

### 🔹 ****Test 5: Pipe cu heredoc****

cat << EOF | grep test

line1 (enter)

test line (enter)

line2 (enter)

EOF (enter)

✅ **Rezultat așteptat:**

test line

### 🔹 ****Test 6: Pipe cu fișier de intrare****

echo test123 > in.txt

cat < in.txt | grep test

✅ **Rezultat așteptat:**

test123

### 🔹 ****Test 7: Pipe care nu returnează nimic (filtrare fără rezultat)****

echo abc | grep zzz

✅ **Rezultat așteptat:** (linie goală)

### 🔹 ****Test 8: Pipe triplu (lanț de pipe-uri)****

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

✅ **Rezultat așteptat:**

b

### 🔹 ****Test 9: Pipe invalid – fără comanda din dreapta****

echo Hello |

✅ **Rezultat așteptat:**  
lipsă de execuție, linie goala (comanda e incompletă).

## ✅ **Teste manuale pentru builtins**

### 🔹 echo cu opțiunea -n

* echo Hello world  
  → ✅ Output: Hello world\n
* echo -n Hello  
  → ✅ Output: Hello (fără newline)
* echo -n -n Hello  
  → ✅ Output: Hello (mai multe -n sunt ignorate după primul)
* echo -n  
  → ✅ Output: (nu afișează nimic, nici newline)

### 🔹 cd cu cale relativă sau absolută

* cd /  
  → ✅ Navighează în root (/)
* cd ..  
  → ✅ Navighează în directorul părinte
* cd ./folder  
  → ✅ Navighează în subfolderul relativ
* cd  
  → ❌ Trebuie să afișeze eroare (nu este suportat fără argument)
* cd folder\_inexistent  
  → ❌ Afișează mesaj de eroare dacă folderul nu există

### 🔹 pwd fără opțiuni

* pwd  
  → ✅ Afișează calea absolută a directorului curent

### 🔹 export fără opțiuni

* export VAR=value  
  → ✅ Creează sau actualizează o variabilă
* export VAR=value VAR2=hello  
  → ✅ Setează mai multe variabile
* export 123INVALID=value  
  → ❌ Afișează eroare (nume invalid)
* export  
  → ✅ Afișează toate variabilele în formatul declare -x VAR="value"

### 🔹 unset fără opțiuni

* unset VAR  
  → ✅ Șterge variabila VAR dacă există
* unset VAR1 VAR2  
  → ✅ Șterge mai multe variabile
* unset  
  → ✅ Nu face nimic dacă nu are argumente
* unset 123INVALID  
  → ❌ Ignoră argumentele invalide

### 🔹 env fără opțiuni sau argumente

* env  
  → ✅ Afișează toate variabilele de mediu (doar cele care au valoare)
* env VAR=123  
  → ❌ Afișează eroare (nu sunt permise argumente)
* env -i  
  → ❌ Afișează eroare (opțiuni nepermise)

### 🔹 exit fără opțiuni

* exit  
  → ✅ Iese din shell cu codul sh->exit\_code
* exit 42  
  → ✅ Iese din shell cu codul 42
* exit 256  
  → ✅ Iese cu cod 0 (256 % 256)
* exit -1  
  → ✅ Iese cu cod 255 (-1 % 256)
* exit 1 2  
  → ❌ Afișează eroare: exit: too many arguments
* exit hello  
  → ❌ Afișează eroare: exit: numeric argument required, cod 255

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