Contents

1 Documentation Complète du Projet Minishell

1.1 Table des Matières

1.2 Vue d'ensemble du Projet

Minishell est une implémentation minimaliste d'un shell Unix écrit en C. Il reproduit les fonctionnalités de base de bash, incluant :

- Exécution de commandes simples et complexes
- Gestion des pipes (`|`)
- Opérateurs logiques (`&&`, `||`)
- Redirections (`>`, `>>`, `<`, `<<`)
- Variables d'environnement et expansion
- Commandes intégrées (builtins)
- Gestion des signaux
- Historique des commandes
- Expansion des wildcards

1.2.1 Objectifs Pédagogiques

Ce projet fait partie du cursus 42 et vise à approfondir : - La programmation système Unix - La gestion des processus et signaux - Le parsing et l'analyse syntaxique - La gestion de la mémoire - L'architecture modulaire

---

1.3 Architecture Générale

Le projet suit une architecture modulaire avec séparation claire des responsabilités :

```
          MAIN LOOP
(Lecture, Parsing, Exécution)
```

```
PARSING                    EXECUTION

Tokenize           AST Exec
Parse                      Builtins
AST                        Commands
```

```
        UTILITY MODULES
```

- Environment Management
- Memory Management
- Signal Handling
- String Operations
- File Operations

1.3.1 Flux d'Exécution Principal

1. **Initialisation** : Configuration de l'environnement et des signaux
2. **Boucle Principale** : Lecture de l'input utilisateur avec readline
3. **Tokenisation** : Découpage de l'input en tokens
4. **Parsing** : Construction de l'AST (Abstract Syntax Tree)
5. **Exécution** : Parcours et exécution de l'AST
6. **Nettoyage** : Libération de la mémoire

1.4 Structures de Données

1.4.1 Structure Principale : `t_user_info`

```c
typedef struct s_user_info
{
    char    **env;           // Variables d'environnement
    int     last_exit_code;  // Code de sortie de la dernière commande
    pid_t   shell_pid;       // PID du shell principal
    // Autres champs de gestion d'état
} t_user_info;
```

1.4.2 AST (Abstract Syntax Tree) : `t_ast`

```c
typedef struct s_ast
{
    t_enum_cmd      type;                   // Type de nœud (SIMPLE, PIPE, AND, OR, etc.)
    char            **argv;                 // Arguments de la commande
```

```
    char            *infile;                // Fichier d'entrée
    char            *outfile;               // Fichier de sortie
    int             append_mode;            // Mode append pour redirections
    int             stderr_to_devnull;      // Redirection stderr vers /dev/null
    char            *heredoc_delimiter;     // Délimiteur pour heredoc
    pid_t           child;                  // PID du processus enfant
    char            **full_cmd;             // Commande complète
    char            *ful_path;              // Chemin complet de l'exécutable
    struct s_ast    *left;                  // Nœud enfant gauche
    struct s_ast    *right;                 // Nœud enfant droit
} t_ast;
```

1.4.3 Types de Commandes : `t_enum_cmd`

```
typedef enum e_enum_cmd
{
    SIMPLE,         // Commande simple
    WORD,           // Mot/argument
    PIPE,           // Pipe |
    AND,            // Opérateur logique &&
    OR,             // Opérateur logique ||
    REDIR_IN,       // Redirection d'entrée <
    REDIR_OUT,      // Redirection de sortie >
    REDIR_APPEND,   // Redirection append >>
    HEREDOC,        // Here document <<
    LPAREN,         // Parenthèse ouvrante
    RPAREN,         // Parenthèse fermante
    SUBSHELL,       // Sous-shell
    CMD_INVALID     // Commande invalide
} t_enum_cmd;
```

1.4.4 Token : `t_token`

```
typedef struct s_token
{
    int     type;    // Type du token
    char    *value;  // Valeur du token
} t_token;
```

1.4.5 Structure de Liste : `t_list`

```
typedef struct s_list
{
    void            *content;   // Contenu du nœud
    struct s_list   *next;      // Pointeur vers le nœud suivant
} t_list;
```

1.5 Modules et Headers

1.5.1 1. minishell.h - Header Principal

Rôle : Inclusion de toutes les dépendances système et des headers du projet.

Inclusions : - Headers système : `<stdio.h>`, `<stdlib.h>`, `<unistd.h>`, `<readline/readline.h>`, etc. - Headers du projet : `ast.h`, `command.h`, `parsing.h`, `execution.h`, etc.

Définitions globales : - `PROMPT "minishell> "` : Prompt par défaut - `PATH_MAX 4096` : Taille maximale des chemins - `g_received_signal` : Variable globale pour la gestion des signaux

1.5.2 2. ast.h - Structures AST

Rôle : Définit les structures pour l'arbre syntaxique abstrait.

Structures principales : - `t_ast` : Nœud de l'AST - `t_expand_ctx` : Contexte d'expansion des variables - `t_parse_word_ctx` : Contexte de parsing des mots - `t_match_ctx` : Contexte de matching pour les wildcards

1.5.3 3. token.h - Gestion des Tokens

Rôle : Structures pour la tokenisation.

Structures : - `t_token` : Token individuel - `t_tokenize_context` : Contexte de tokenisation

1.5.4 4. command.h - Types de Commandes

Rôle : Énumérations et structures pour les types de commandes.

Définitions : - `t_enum_cmd` : Types de commandes - `t_builtin_func` : Pointeur de fonction pour builtins - `t_builtin` : Structure pour builtins

1.5.5 5. parsing.h - Module de Parsing

Rôle : Déclarations pour toutes les fonctions de parsing.

Fonctions principales : - Tokenisation : `tokenize()`, `process_token()` - Parsing AST : `parse_command()`, `parse_logical_command()` - Expansion : `expand_variable()`, `remove_quotes_and_expand()` - Gestion quotes : `handle_single_quote()`, `handle_double_quote()`

1.5.6 6. execution.h - Module d'Exécution

Rôle : Déclarations pour l'exécution des commandes.

Fonctions principales : - Exécution : `execute_ast()`, `execute_simple()`, `execute_pipe()` - Résolution paths : `find_command_path()`, `resolve_command_path()` - Gestion processus : `fork_and_exec_left()`, `wait_for_child_sim()`

1.5.7 7. builtin.h - Commandes Intégrées

Rôle : Déclarations des commandes builtin.

Builtins supportés : - `echo` : Affichage de texte - `cd` : Changement de répertoire - `pwd` : Répertoire courant - `export` : Export de variables - `unset` : Suppression de variables - `env` : Affichage environnement - `exit` : Sortie du shell

1.5.8 8. env.h - Gestion Environnement

Rôle : Gestion des variables d'environnement.

Fonctions : - `ms_getenv()` : Récupération variable - `ms_setenv()` : Définition variable - `ms_unsetenv()` : Suppression variable - `ms_env_init()` : Initialisation environnement

1.5.9 9. utils.h - Utilitaires

Rôle : Fonctions utilitaires diverses.

1.5.10 10. libft.h - Bibliothèque Custom

Rôle : Fonctions de base réimplémentées.

Fonctions principales : - Manipulation strings : `ft_strlen()`, `ft_strdup()`, `ft_strjoin()` - Conversion : `ft_itoa()`, `ft_atol()` - Listes : `ft_lstadd_back()`, `ft_lstclear()` - Matrices : `ft_split()`, `ft_matrix_len()`

1.6 Fonctions par Module

1.6.1 Module Principal (`main/`)

```c
int main(int argc, char **argv, char **envp)
```

1.6.1.1 `main.c`

- Point d'entrée principal
- Gère les arguments en ligne de commande
- Lance la boucle principale ou exécute une commande unique avec `-c`

```c
int minishell_loop(char **envp)
```

1.6.1.2 `main_loop.c`

- Boucle principale du shell
- Gère readline, parsing et exécution

```c
void init_user_info_m(t_user_info *user, char **envp)
```

- Initialise la structure utilisateur

```c
int process_input_m(const char *input, t_user_info *user, int *exiter)
```

1.6.1.3 `main_input.c`

- Traite une ligne d'entrée complète
- Tokenise, parse et exécute

1.6.2 Module Parsing (parsing/)

```
int tokenize(const char *input, t_list **tokens, char **env, int exit_status)
```

1.6.2.1 Tokenisation (token/)

- Découpe l'input en tokens
- Gère les quotes, opérateurs, et espaces

```
int process_token(char *cleaned, int i, t_tokenize_context *ctx, int *new_index)
```

- Traite un token individuel selon son type

```
t_ast *parse_command(t_list *tokens)
```

1.6.2.2 Parsing AST (parser/)

- Point d'entrée du parsing
- Construit l'AST à partir des tokens

```
t_ast *parse_logical_command(t_list *tokens)
```

- Parse les opérateurs logiques (&&, ||)

```
t_ast *parse_simple_command(t_list **tokens)
```

- Parse une commande simple avec ses arguments

```
char *expand_variable(const char *str, int *i, char **env, int exit_status)
```

1.6.2.3 Expansion de Variables

- Expanse les variables d'environnement ($VAR)

```
char *remove_quotes_and_expand(const char *input, char **env)
```

- Supprime les quotes et expanse les variables

```
int handle_single_quote(const char *s, int *i, char **value)
```

1.6.2.4 Gestion des Quotes (quote/)

- Traite le contenu entre quotes simples (pas d'expansion)

```
int handle_double_quote(const char *s, int *i, char **value, t_expand_ctx *ctx)
```

- Traite le contenu entre quotes doubles (avec expansion)

1.6.3 Module Exécution (exec/)

```
int execute_ast(t_ast *node, t_user_info *user, int *exiter)
```

1.6.3.1 Exécution AST (ast_exec/)

- Exécute récursivement l'AST
- Délègue selon le type de nœud

```
int execute_simple(t_ast *node, t_user_info *user, int *exiter)
```

1.6.3.2 Exécution Simple (`simple_exec/`)

- Exécute une commande simple
- Gère builtins vs commandes externes

```
int handle_child_process_s(t_ast *node, t_user_info *user)
```

- Processus enfant pour commande externe

```
int execute_pipe(t_ast *node, t_user_info *user, int *exiter)
```

1.6.3.3 Exécution Pipes (`pipe_exec/`)

- Exécute un pipeline
- Crée les pipes et processus enfants

```
pid_t fork_and_exec_left(t_ast *left, int pipefd[2], t_user_info *user, int *exiter)
```

- Fork et exécute la partie gauche du pipe

```
char *resolve_command_path(const char *cmd, char **env)
```

1.6.3.4 Résolution de Chemin (`path/`)

- Résout le chemin complet d'une commande
- Cherche dans PATH ou utilise chemin absolu/relatif

```
char *find_command_in_path(const char *cmd, char **env)
```

- Cherche une commande dans les répertoires PATH

```
int setup_redirections(t_ast *node)
```

1.6.3.5 Redirections (`redirection/`)

- Configure les redirections d'entrée/sortie
- Gère <, >, >>, <<

```
int handle_heredoc(const char *delimiter, char **env)
```

1.6.3.6 Heredoc (`heredoc/`)

- Implémente la fonctionnalité heredoc
- Lit jusqu'au délimiteur spécifié

1.6.4 Module Builtins (`builtin/`)

```
int ms_echo(t_list *cmd)
```

1.6.4.1 `echo/`

- Implémente la commande echo
- Gère l'option -n

```
int ms_cd(char **args, char ***env)
```

1.6.4.2 `cd/`

- Implémente la commande cd
- Met à jour PWD et OLDPWD

```
char *get_target_directory(char **args, char **env)
```

- Détermine le répertoire cible

```
int ms_pwd(void)
```

1.6.4.3 `pwd/`

- Affiche le répertoire courant

```
int ms_env(char **env)
```

1.6.4.4 `env/`

- Affiche les variables d'environnement

```
int handle_builtin_export(t_user_info *prompt, char **args)
```

1.6.4.5 `export/`

- Exporte des variables d'environnement
- Gère la syntaxe VAR=value

```
int handle_builtin_unset(t_user_info *prompt, char **args)
```

1.6.4.6 `unset/`

- Supprime des variables d'environnement

```
int ms_exit(int argc, char **argv, char **envp)
```

1.6.4.7 `exit/`

- Quitte le shell avec code de sortie optionnel

1.6.5 Module Environnement (`env/`)

```
char *ms_getenv(char **env, const char *key, int n)
```

1.6.5.1 Gestion Variables

- Récupère une variable d'environnement

```
char **ms_setenv(char **env, const char *key, const char *value)
```

- Définit une variable d'environnement

```
char **ms_unsetenv(char **env, const char *key)
```

- Supprime une variable d'environnement

```
char **ms_env_init(char **envp)
```

1.6.5.2 Initialisation

- Initialise l'environnement à partir d'envp

1.6.6 Module Signaux (`signal/`)

```
void setup_signals(void)
```

- Configure les gestionnaires de signaux
- Gère SIGINT, SIGQUIT

```
void handle_sigint(int sig)
```

- Gestionnaire pour Ctrl+C

1.6.7 Module Wildcards (`wildcard/`)

```
char **expand_wildcards(const char *pattern)
```

- Expanse les patterns avec * et ?
- Retourne liste des fichiers correspondants

1.6.8 Module Libft (`libft/`)

```
size_t ft_strlen(const char *s)
char *ft_strdup(const char *s1)
char *ft_strjoin(char const *s1, char const *s2)
int ft_strcmp(const char *s1, const char *s2)
char *ft_substr(char const *s, unsigned int start, size_t len)
```

1.6.8.1 Manipulation Strings

```
char *ft_itoa(int n)
long ft_atol(const char *str)
```

1.6.8.2 Conversion

```
void ft_lstadd_back(t_list **lst, t_list *new)
void ft_lstclear(t_list **lst, void (*del)(void*))
int ft_lstsize(t_list *lst)
```

1.6.8.3 Listes

```
char **ft_split(char const *s, char c)
int ft_matrix_len(char **matrix)
```

1.6.8.4 Matrices

1.6.9 Module Nettoyage (`free/`)

```
void free_ast(t_ast *ast)
```

- Libère récursivement un AST

```
void free_tokens(t_list *tokens)
```

- Libère une liste de tokens

```
void cleanup_user_info_m(t_user_info *user)
```

- Libère toutes les ressources utilisateur

1.7 Diagramme de Flux

1.7.1 Flux d'Exécution Principal

```
graph TB
    A[Démarrage] --> B[Initialisation]
    B --> C[Configuration Signaux]
    C --> D[Boucle Principale]

    D --> E[Lecture Input readline]
    E --> F{Input vide?}
    F -->|Oui| D
    F -->|Non| G[Ajout Historique]

    G --> H[Tokenisation]
    H --> I{Erreur Syntaxe?}
    I -->|Oui| J[Affichage Erreur]
    J --> D
```

```
    I -->|Non| K[Parsing AST]

    K --> L{AST Valide?}
    L -->|Non| J
    L -->|Oui| M[Exécution AST]

    M --> N[Nettoyage Mémoire]
    N --> D

    D --> O{Commande Exit?}
    O -->|Non| D
    O -->|Oui| P[Nettoyage Final]
    P --> Q[Fin]
```

1.7.2 Flux de Tokenisation

```
graph TB
    A[Input String] --> B[Caractère Courant]
    B --> C{Type Caractère}

    C -->|Espace| D[Skip Espaces]
    C -->|Quote| E[Traiter Quote]
    C -->|Opérateur| F[Parser Opérateur]
    C -->|Caractère Normal| G[Construire Mot]

    D --> H[Caractère Suivant]
    E --> I[Gérer Contenu Quote]
    F --> J[Créer Token Opérateur]
    G --> K[Continuer Mot]

    I --> L[Fermer Quote]
    J --> H
    K --> M{Fin Mot?}
    L --> N[Créer Token Mot]

    M -->|Non| K
    M -->|Oui| N
    N --> H

    H --> O{Fin Input?}
    O -->|Non| B
    O -->|Oui| P[Retourner Tokens]
```

1.7.3 Flux d'Exécution AST

```
graph TB
    A[Nœud AST] --> B{Type Nœud}

    B -->|SIMPLE| C[Execute Simple]
```

```
    B -->|PIPE| D[Execute Pipe]
    B -->|AND| E[Execute AND]
    B -->|OR| F[Execute OR]

    C --> G[Résoudre Chemin]
    G --> H{Builtin?}
    H -->|Oui| I[Exécuter Builtin]
    H -->|Non| J[Fork + Exec]

    D --> K[Créer Pipe]
    K --> L[Fork Gauche]
    L --> M[Fork Droite]
    M --> N[Attendre Enfants]

    E --> O[Exécuter Gauche]
    O --> P{Succès?}
    P -->|Oui| Q[Exécuter Droite]
    P -->|Non| R[Skip Droite]

    F --> S[Exécuter Gauche]
    S --> T{Échec?}
    T -->|Oui| U[Exécuter Droite]
    T -->|Non| V[Skip Droite]

    I --> W[Retourner Code]
    J --> X[Attendre Enfant]
    N --> W
    Q --> W
    R --> W
    U --> W
    V --> W
    X --> W
```

1.8 Compilation et Utilisation

1.8.1 Makefile

Le projet utilise un Makefile sophistiqué avec :

Variables principales : - `NAME := minishell` : Nom de l'exécutable - `SOURCES := ./source` : Répertoire des sources - `INCLUDE := ./include` : Répertoire des headers - `OBJECTS := ./bin` : Répertoire des objets

Flags de compilation : - `-Wall -Wextra -Werror` : Warnings strictes - `-g3 -fno-omit-frame-pointer` : Debug info - `-O1` : Optimisation légère - `-I$(INCLUDE)` : Include path

Dépendances : - `libreadline` : Pour l'interface en ligne de commande

Règles principales : - `make` ou `make all` : Compilation complète - `make clean` : Supprime objets - `make fclean` : Supprime objets et exécutable - `make re` : Recompilation complète

1.8.2 Compilation

```
# Compilation standard
make

# Compilation avec AddressSanitizer (décommenter dans Makefile)
make clean
# Éditer Makefile pour activer ASAN
make

# Installation de readline sur différents systèmes
# Ubuntu/Debian
sudo apt-get install libreadline-dev

# macOS avec Homebrew
brew install readline
```

1.8.3 Utilisation

```
# Lancement interactif
./minishell

# Exécution d'une commande unique
./minishell -c "ls -la | grep minishell"

# Avec suppression Valgrind
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --suppressions=readline.supp
```

1.8.4 Exemples d'Utilisation

```
# Commandes simples
minishell> ls -la
minishell> echo "Hello World"
minishell> pwd

# Pipes
minishell> ls | grep .c | wc -l
minishell> cat file.txt | sort | uniq

# Redirections
minishell> echo "test" > output.txt
minishell> cat < input.txt
minishell> ls >> log.txt

# Heredoc
minishell> cat << EOF
> Ligne 1
> Ligne 2
> EOF
```

```
# Opérateurs logiques
minishell> ls && echo "Success"
minishell> false || echo "Failed"

# Variables d'environnement
minishell> export MY_VAR="Hello"
minishell> echo $MY_VAR
minishell> unset MY_VAR

# Wildcards
minishell> ls *.c
minishell> echo test_*

# Sous-commandes et groupement
minishell> (cd /tmp && ls)
```

1.9 Gestion des Erreurs

1.9.1 Types d'Erreurs

```
# Pipes mal formés
minishell> ls |
bash: syntax error near unexpected token 'newline'

# Parenthèses non fermées
minishell> (echo test
bash: syntax error: unexpected end of file
```

1.9.1.1 1. Erreurs de Syntaxe

```
# Commande introuvable
minishell> nonexistent_command
bash: nonexistent_command: command not found

# Permission refusée
minishell> ./file_without_execute_permission
bash: ./file_without_execute_permission: Permission denied
```

1.9.1.2 2. Erreurs de Commandes

```
# Fichier introuvable pour redirection
minishell> cat < nonexistent_file
bash: nonexistent_file: No such file or directory

# Répertoire au lieu de fichier
```

```
minishell> ./directory
bash: ./directory: Is a directory
```

1.9.1.3 3. Erreurs de Fichiers

1.9.2 Codes de Sortie

Le shell utilise les codes de sortie standards Unix : - `0` : Succès - `1` : Erreur générale - `2` : Mauvaise utilisation de builtin - `126` : Commande trouvée mais non exécutable - `127` : Commande introuvable - `128+n` : Terminé par signal n

1.9.3 Gestion dans le Code

```c
// Gestion erreur commande introuvable
void handle_cmd_path_errors_s(const char *cmd_path, const char *cmd_name, t_user_info *user)
{
    if (!cmd_path)
    {
        ft_putstr_fd("minishell: ", 2);
        ft_putstr_fd(cmd_name, 2);
        ft_putstr_fd(": command not found\n", 2);
        exit_sim(127, NULL, NULL, user);
    }
    // ... autres vérifications
}

// Gestion erreur permission
void print_and_exit_perm(const char *cmd_name, t_user_info *user)
{
    ft_putstr_fd("minishell: ", 2);
    ft_putstr_fd(cmd_name, 2);
    ft_putstr_fd(": Permission denied\n", 2);
    exit_sim(126, NULL, NULL, user);
}
```

1.10 Tests et Debugging

1.10.1 Tests Manuels

```
# Test commandes simples
echo "Test simple commands"
ls
pwd
cd /tmp
pwd
cd -

# Test pipes
```

```
echo "Test pipes"
ls | wc -l
cat /etc/passwd | head -5 | tail -1

# Test redirections
echo "Test redirections"
echo "hello" > test.txt
cat test.txt
echo "world" >> test.txt
cat test.txt
rm test.txt
```

1.10.1.1 Tests Basiques

```
# Test opérateurs logiques
echo "Test logical operators"
true && echo "Success"
false || echo "Failed"
false && echo "Not printed"
true || echo "Not printed"

# Test variables
echo "Test variables"
export TEST_VAR="Hello World"
echo $TEST_VAR
echo "$TEST_VAR from shell"
unset TEST_VAR
echo $TEST_VAR

# Test quotes
echo "Test quotes"
echo 'Single quotes: $HOME'
echo "Double quotes: $HOME"
echo "Mixed: '$HOME'"
```

1.10.1.2 Tests Avancés

1.10.2 Debugging avec Valgrind

```
# Vérification fuites mémoire
valgrind --leak-check=full \
         --show-leak-kinds=all \
         --track-origins=yes \
         --suppressions=readline.supp \
         ./minishell

# Debugging avec GDB
gdb ./minishell
(gdb) run
```

```
(gdb) bt
(gdb) print variable_name
```

1.10.3 Tests de Stress

```
# Test pipes multiples
ls | grep .c | sort | uniq | wc -l

# Test redirections multiples
echo "test" > file1 && cat file1 > file2 && cat file2

# Test commandes longues
find /usr -name "*.h" 2>/dev/null | head -10 | while read file; do echo "Found: $file"; done
```

1.10.4 Vérification Normes 42

```
# Vérification avec norminette
norminette source/ include/

# Vérification interdictions
# - Pas de variables globales (sauf g_received_signal)
# - Pas de fonctions interdites
# - Gestion correcte de la mémoire
```

1.11 Conclusion

Ce projet Minishell représente une implémentation complète et fonctionnelle d'un shell Unix. L'architecture modulaire facilite la maintenance et l'extension, tandis que la gestion rigoureuse de la mémoire et des erreurs assure la robustesse du programme.

1.11.1 Points Forts du Projet

1. **Architecture Claire** : Séparation nette entre parsing, exécution et utilities
2. **Gestion Mémoire** : Nettoyage systématique des ressources
3. **Compatibilité** : Comportement proche de bash
4. **Extensibilité** : Ajout facile de nouvelles fonctionnalités
5. **Robustesse** : Gestion complète des cas d'erreur

1.11.2 Améliorations Possibles

1. **Performance** : Optimisation du parsing pour les commandes complexes
2. **Fonctionnalités** : Ajout de job control, completion automatique
3. **Tests** : Suite de tests automatisés plus complète
4. **Documentation** : Ajout d'exemples d'utilisation avancée

Ce document constitue une référence complète pour comprendre, utiliser et maintenir le projet Minishell.