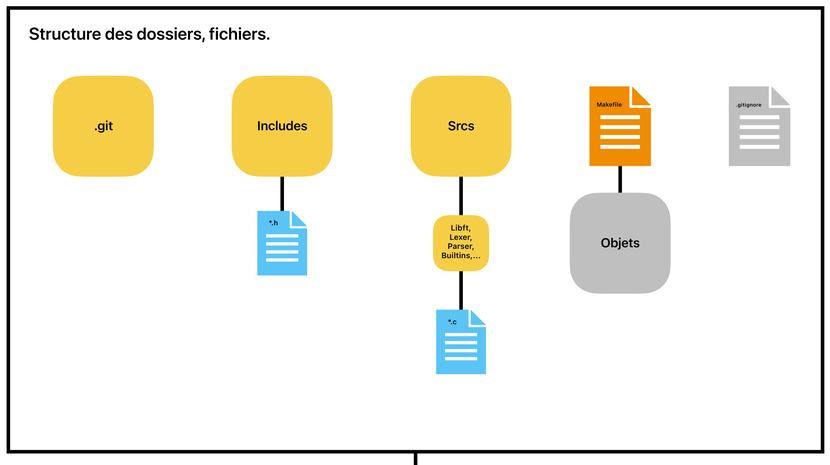
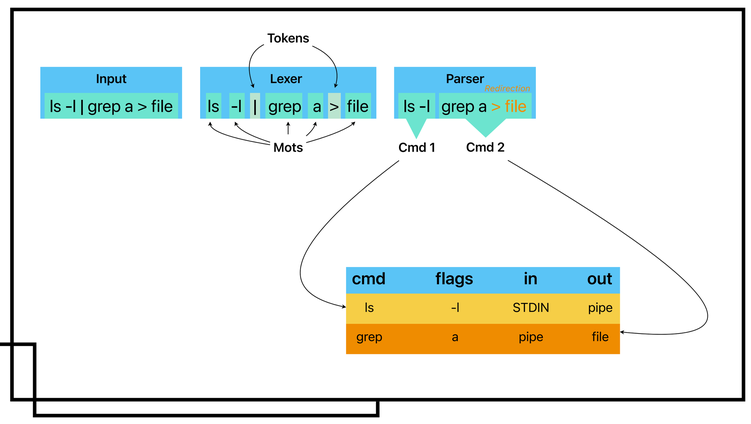


Minishell

1. Afficher le prompt tant qu'il n'y a aucune commande.
2. Avoir un historique de commande.
3. Trouver et lancer les executables en utilisant la variable d'environnement PATH ou en utilisant le chemin relatif ou le chemin absolu.
4. Ne pas interpréter de quotes (guillemets) non fermés ou de caractères spéciaux non demandés dans le sujet, tels que \ (le backslash) ou ; (le point-virgule).
5. Gérer ' (single quote) qui doit empêcher le shell d'interpréter les méta-caractères présents dans la séquence entre guillemets.
6. Gérer " (double quote) qui doit empêcher le shell d'interpréter les méta-caractères présents dans la séquence entre guillemets sauf le $ (signe dollar).
7. Implémenter les redirections comme :
   1. "<" redirect input.
   2. ">" redirect output.
   3. "<<" heredoc (n'affecte pas l'historique).
   4. ">>" redirige l'output en mode ajout.
8. Implémenter les pipes "|".
9. Gérer les variables d'environnement (un $ suivi d'une séquence de caractères)qui doivent être substituées par leur contenu.
10. Gérer les valeurs de retour avec "**$?**".
11. CTRL-C, CTRL-D et CTRL-/ doivent fonctionner comme dans bash.
12. Re-creer ses propres builtins :
    1. "**echo**" avec l'option **-n**.
    2. "**cd**" avec seulement le chemin absolu ou relatif.
    3. "**pwd**" (sans les flags).
    4. "**export**" (sans les flags).
    5. "**unset**" (sans les flags ou arguments).
    6. "**exit**" (sans les flags).



Implementation

Implementation :
Le programme est exécuté sans arguments (et génère une erreur s'il en utilise).
Le programme se compose essentiellement de deux fonctions qui s'appellent l'une l'autre indéfiniment. La première, minishell_loop, exécute les fonctions de minishell, l'autre nettoie et prépare la ligne suivante. Dans minishell_loop, une invite de commande apparaît, qui est implémentée par **readline**. Cela nous permet également d'utiliser la fonction d'historique intégrée. Une fois qu'une ligne a été saisie, il vérifie s'il y a des guillemets non fermés. S'il n'en trouve pas, il envoie la ligne au **Lexer**.

Lexer

Lexer / tokenizer :
Il prend en entrée la ligne saisie. Il lit ensuite la ligne mot par mot, en utilisant les white space comme délimiteurs. Il vérifie d'abord si le mot est un token, c'est-à-dire : |, <, <<, >, ou >>, et sinon il suppose qu'il s'agit d'un mot.

Enum token :

```
typedef enum e_token
{
        NOT_A_TOKEN = 0,
        PIPE,
        GREATER,
        D_GREATER,
        LOWER,
        D_LOWER
}       t_token;
```

Structure Liste doublement chainée :

```
typedef struct s_lexer
{
        int                     i;
        char                    *str;
        t_tokens                token;
        struct s_lexer  *next;
        struct s_lexer  *prev;
}       t_lexer;
```

Chaque node contient soit un char * contenant le mot soit un **t_token**. Chaque node est assigné à un index afin de les supprimer plus facilement par la suite.

Parser

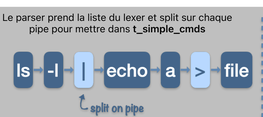
Parser
Le **Lexer** est envoyé au **Parser** qui regroupe les différentes nodes en fonction des tokens. Chaque groupe devient une commande

Structure Liste doublement chainée :

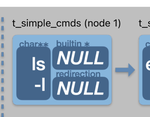
```
typedef struct s_simple_cmds
{
        char                            **str;
        int                             (*builtin)(t_tools *, struct s_simple_cmds *);
        int                             num_redirections;
        char                            *hd_file_name;
        t_lexer                         *redirections;
        struct s_simple_cmds    *next;
        struct s_simple_cmds    *prev;
}       t_simple_cmds;
```

La première chose que fait le parser est de parcourir en boucle la liste des lexers jusqu'à ce qu'il rencontre un pipe. Il prend alors toutes les nodes avant le pipe comme une seule commande, et crée une node dans la structure **t_simple_cmds**. S'il ne trouve pas de pipe, il prend toutes les nodes (restantes) comme une seule commande.

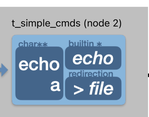
Le parser prend la liste du lexer et split sur chaque pipe pour mettre dans **t_simple_cmds**



t_simple_cmds (node 1)



t_simple_cmds (node 2)



Pour chaque commande, il vérifie d'abord les redirections, qu'il stocke dans la liste chainée ***redirections**, qui contient à la fois le **token** et **le nom de fichier** ou le **délimiteur** dans le cas d'un **heredoc**. Lorsque les nodes sont ajoutés à la liste ***redirections**, ils sont supprimés de la liste du **lexer**. Ensuite, il vérifie si le premier mot est un **builtin**, auquel cas il stocke un pointeur de fonction vers la fonction correspondante. Comme les redirections ont été supprimées de la liste, le **parser** peut facilement combiner tous les **mots restants** dans un **tableau 2D**, ce qui est un argument nécessaire d'**execve**. Cela permet également de gérer plus facilement les situations où les mots peuvent être séparés par des redirections

Exemple :

cat > file -e

Comme ">" et "**file**" sont déjà supprimés de la liste du lexer lorsqu'ils sont ajoutés à la liste des redirections, il ne reste que "**cat**" et "**-e**", qui peuvent alors être facilement ajoutés à un tableau.

Ce processus est répété jusqu'à la fin de la liste du lexer.

Builtins

Nous gérons les fonctions intégrées, comme nous l'avons vu plus haut, en stockant un pointeur de fonction dans **t_simple_cmds**. Pour ce faire, nous envoyons le premier mot d'une commande à une fonction **builtin_arr** qui parcourt un **tableau statique** des différentes fonctions intégrées. Si elle trouve une fonction correspondante, elle la renvoie au **parser**, sinon elle renvoie **NULL**. En outre, le fait de déterminer la construction au stade du parser simplifie grandement l'exécuteur, car l'exécution de la construction ne nécessite que deux lignes de code :

```
if (cmd->builtin != NULL)
  cmd->builtin(tools, cmd);
```

cd
Change le répertoire de travail de l'environnement d'exécution actuel du shell et met à jour les variables d'environnement **PWD** et **OLDPWD**. Sans arguments, il change le répertoire de travail en répertoire personnel.
- change le répertoire en **OLDPWD**.

echo
Affiche une ligne de texte

env
Affiche les variables d'environement

exit
Termine l'execution du shell. Accepte l'argument optionnel **n**, qui fixe l'état de sortie à **n**.

export
Accepte les arguments nom[=valeur].
Ajoute nom à l'environnement. Fixe la valeur de nom à valeur.
Si aucun argument n'est donné, affiche la liste des variables exportées.

pwd
Affiche le répertoire actuel sous la

unset
Prend un argument "name".
Supprime la variable d'environnement "name".

Executor

Lorsque le **parser** renvoie la liste **t_simple_cmds** à **minishell_loop**, une simple vérification est effectuée pour déterminer le nombre de commandes, car elles sont gérées par des fonctions différentes. Toutefois, à l'exception de quelques builtins intégrés, les commandes sont finalement exécutées par la même fonction **handle_cmd**, qui trouve et, si elle réussit, exécute la commande.

Expander

Avant qu'un node de **t_simple_cmds** ne soit traité, il est développé. L'expandeur prend les variables, identifiées par **$**, et les remplace par leur valeur dans les variables d'environnement. Ainsi, **$USER** devient **aabda**, et **$?** est remplacé par le code de sortie.

Heredoc

Avant de créer un processus enfant, le processus parent exécute le **heredocs**. Nous avons géré le **heredocs** en créant un fichier temporaire pour y écrire les données d'entrée. Le nom du fichier est stocké dans un node **t_simple_cmds** correspondant afin qu'il puisse être utilisé pour remplacer **STDIN**. Si un seul node **t_simple_cmds** contient plusieurs **heredocs**, le nom du fichier stocké sera celui du dernier **heredoc**. L'utilisation d'un fichier comporte des limitations et des problèmes de sécurité, mais nous avons pensé que c'était la façon la plus simple de traiter le problème, et qu'elle était proche de la façon dont bash le fait.

Single command

Comme dans bash, les commandes builtins, en particulier **cd**, **exit**, **export** et **unset**, ne peuvent pas être exécutées dans un processus distinct, car la variable environnementale ne peut alors pas être modifiée correctement. S'il n'y a qu'une seule commande, et qu'il s'agit d'une des commandes builtins, elle est exécutée dans le processus parent et la fonction retourne à la boucle **minishell_loop**. Si la commande n'est pas une commande builtin, la fonction de commande unique crée un nouveau processus et envoie la commande à **handle_cmd**.

Multiple commands

Si il y a plusieurs commandes, l'executor boucle sur chaque node de **t_simple_cmds** et créer un processus enfant en utilisant la fonction **fork()**, et en utilisant la fonction **pipe()** pour créer un pipe et envoyer l'output d'une commande dans l'input de la prochaine. Pour chaque commande il se passe:
1. La commande est développé.
2. un pipe est crée avec **pipe[0]** et **pipe[1]**, excepté la dernière commande.
3. Utilise **fork()** pour créer un processus enfant. Dans le processus enfant :
   1. A l'exception de la premiere commande, **dup2** remplace **STDIN** avec output de la commande précédente.
   2. A l'exception de la dernière commande, **dup2** remplace **STDOUT** avec **pipe[1]**.
   3. Dans le cas des redirections, le **STDIN** ou le **STDOUT** est remplacé avec leurs file descripteur respectif.
   4. **handle_cmd** trouve et execute la commande.
4. **pipe[0]** est stocké pour la commande suivante.

Le processus parent attend ensuite que tous les processus enfants se terminent, puis retourne à la boucle **minishell_loop**.

Reset

Le programme effectue ensuite une réinitialisation complète, libérant toute les nodes qui n'ont pas encore été libérés ou supprimés, et réinitialise diverses variables afin que le programme puisse redémarrer en affichant une nouvelle invitée de commande.