MiniShell

MiniShell will introduce you to the world of shells, which provides a convenient text interface to interact with your system. Shells might seem very easy to understand but have very specific and defined behavior in almost every single case, most of which will need to be handled properly.

Hootgam quote : `lexer` -> `parser` -> `expander` -> `executor`

▼ Prompt

```
char *str1 = "┌──(Minishell㉿Aliens)-[PWD]";
  char *str2 = "└─$";
```

▼ Resources :

- GitHub : 5

https://github.com/Swoorup/mysh

- PDF :

https://www.cs.purdue.edu/homes/grr/SystemsProgrammingBook/Book/Chapter5-WritingYourOwnShell.pdf

- Shell Command Language :

Shell Command Language
https://pubs.opengroup.org/onlinepubs/009695399/utilities/xcu_chap02.html

▼ Function :

- `readline` : Reads a line of input from the user.
- `rl_clear_history` : Clears the history list maintained by the readline library.
- `rl_on_new_line` : Causes the current line to be considered complete and adds it to the history.
- `rl_replace_line` : Replaces the current line with a new one.
- `rl_redisplay` : Updates the display to show the current line.
- `add_history` : Adds a line to the history list.
- `printf` : Prints formatted output to the console.

- `malloc`: Allocates memory dynamically.
- `free`: Frees dynamically allocated memory.
- `write`: Writes data to a file descriptor.
- `access`: Checks the accessibility of a file.
- `open`: Opens a file.
- `read`: Reads data from a file descriptor.
- `close`: Closes a file descriptor.
- `fork`: Creates a new process by duplicating the calling process.
- `wait`: Waits for a child process to terminate.
- `waitpid`: Waits for a specific child process to terminate.
- `wait3`: A variant of `waitpid` that provides more information about the terminated process.
- `wait4`: A variant of `waitpid` that provides even more information about the terminated process.
- `signal`: Sets a signal handler for a specific signal.
- `sigaction`: A more flexible version of `signal` that allows for more control over signal handling.
- `sigemptyset`: Initializes an empty set of signals.
- `sigaddset`: Adds a signal to a set of signals.
- `kill`: Sends a signal to a process or group of processes.
- `exit`: Terminates the calling process.
- `getcwd`: Gets the current working directory.
- `chdir`: Changes the current working directory.
- `stat`: Gets information about a file.
- `lstat`: Gets information about a file, but doesn't follow symbolic links.
- `fstat`: Gets information about an open file descriptor.
- `unlink`: Deletes a file.
- `execve`: Replaces the current process with a new one.
- `dup`: Duplicates a file descriptor.
- `dup2`: Duplicates a file descriptor, but allows for the target descriptor to be specified.
- `pipe`: Creates a pipe for inter-process communication.
- `opendir`: Opens a directory stream.
- `readdir`: Reads a directory stream.
- `closedir`: Closes a directory stream.
- `strerror`: Converts an error number to a string describing the error.
- `perror`: Prints an error message to the console, including the string representation of the current error.
- `isatty`: Checks whether a file descriptor refers to a terminal.
- `ttyname`: Gets the name of the terminal associated with a file descriptor.
- `ttyslot`: Gets the slot number of the current terminal in the terminal database.
- `ioctl`: Performs I/O control operations on a file descriptor.
- `getenv`: Gets the value of an environment variable.

- `tcsetattr`: Sets the attributes of a terminal.
- `tcgetattr`: Gets the attributes of a terminal.
- `tgetent`: Gets the entry for the current terminal type in the terminal database.
- `tgetflag`: Gets the value of a terminal capability flag.
- `tgetnum`: Gets the value of a numeric terminal capability.
- `tgetstr`: Gets the value of a string terminal capability.
- `tgoto`: Constructs a cursor movement sequence for the current terminal type.

▼ Shell Should :

- Create a command-line prompt that can execute different programs based on their location.
- Have a history of previously executed commands.
- Do not use more than one global variable.
- Ignore certain special characters like backslashes and semicolons unless they are in quotes.
- Handle single and double quotes properly so that meta-characters are not interpreted inside the quotes.
- Implement redirections, such as input and output redirection, and append mode.
- Implement pipes so that the output of one command is connected to the input of the next command via a pipe.
- Handle environment variables by expanding them to their values.

▼ Exit Status :

- `0` - Success status. This indicates that the process has been completed successfully without any errors.
- `1` - Generic error status. This can be used to indicate any kind of error that does not have a specific exit code.
- `2` - Misuse or syntax error status. This indicates that the command was used incorrectly or that there was a syntax error in the command.
- `3` - Fatal error status. This indicates that the process has encountered a fatal error and cannot continue.
- `126` - Command not executable status. This indicates that the command could not be executed because it is not executable.
- `127` - Command not found status. This indicates that the command could not be found or is not in the search path.
- `128+n` - Fatal error signal `n` status. This indicates that the process has terminated due to a fatal signal (where `n` is the signal number).
- `130` - Interrupted by the signal status. This indicates that the process has been interrupted by a signal (usually `SIGINT`).
- `137` - Killed by signal status. This indicates that the process has been killed by a signal (usually `SIGKILL`).
- `255` - Exit status out of range status. This indicates that the exit status is outside the valid range of values (0-255).

▼ Signals :

◦ ctrl-C displays a new prompt on a new line.
◦ ctrl-D exits the shell.
◦ ctrl-\ does nothing.

▼ Built-in :

- echo with option -n
- cd with only a relative or absolute path
- pwd with no options
- export with no options
- unset with no options

- env with no options or arguments
- exit with no options

Lexer

▼ Lexer

Lexer analysis (also known as lexical analysis or tokenization) is the process of breaking down a stream of input text into individual tokens, which are meaningful units of the language being processed.

lexer analysis is often used as the first step in the process of compiling or interpreting a programming language, The lexer then outputs a sequence of tokens that the parser can use to build an abstract syntax tree, which represents the structure of the program.

For example, in the following code snippet:

```
pythonCopy code
int x = 5 + 2;
```

the lexer would break it down into the following tokens:

```
vbnetCopy code
int     keyword
x       identifier
=       operator
5       integer
+       operator
2       integer
;       semicolon
```

The lexer analysis makes it easier for the compiler or interpreter to process the input language because it has a structured representation of the program that it can work with.

▼ How to implement a Lexer analysis

1. Define the language grammar: Before you can perform lexer analysis, you need to define the grammar of the language you are working with. This includes specifying the set of valid tokens and their syntax.
2. Define the token definitions: Once you have defined the grammar, you can define the individual token definitions. Each token definition specifies a regular expression pattern that matches the syntax of the token.
3. Write the lexer code: Using the token definitions and the language grammar, you can now write the code for the lexer. The lexer reads in the input text as a stream of characters and matches each character sequence with the regular expression pattern for the corresponding token.
4. Generate the token stream: As the lexer reads in the input text, it generates a stream of tokens. Each token is represented by a tuple or object that contains the token type and its value.
5. Pass the token stream to the parser

Parser

▼ Parser

A parser is a component of a compiler or interpreter that takes the stream of tokens generated by the lexer and uses it to build an abstract syntax tree (AST). The AST is a data structure that represents the structure of the program being compiled or interpreted and is used by the compiler or interpreter to generate executable code or to execute the program directly.

The parser operates according to the rules of the language grammar, which specify the syntax and semantics of the language being processed. The parser reads in the stream of tokens and uses the grammar rules to determine how to build the AST. Each rule in the grammar specifies a production that describes how a certain construct in the language can be built from other constructs in the language. The parser applies these rules recursively to construct the AST, starting with the top-level rule of the grammar.

As the parser constructs the AST, it checks that the structure of the program is well-formed according to the rules of the language grammar. If the program contains syntax errors or violates the grammar rules, the parser reports an error and halts the compilation or interpretation process.

Once the AST has been constructed, the compiler or interpreter can use it to generate executable code or to execute the program directly. The AST provides a structured representation of the program that can be efficiently traversed and analyzed and used to generate optimized code for performance.

In summary, the parser is responsible for taking the stream of tokens generated by the lexer and constructing an abstract syntax tree that represents the structure of the program being compiled or interpreted. The AST provides a structured representation of the program that can be efficiently traversed and analyzed and is used by the compiler or interpreter to generate executable code or to execute the program directly.

▼ How to implement a Parser analysis

▼ Abstract Tree

- **Define the priorities**

You need to know the priorities of the Operation

> 💡 (“ * “ —> “ / ” —> ” - ” —> ” + ”)

> 💡 && → || → | → Word

- You can use a recursive descent parser to parse the command and construct the AST.
- You need to Create a struct :

```
typedef struct s_tree
{
      int            type;
      char           *value;
      struct s_tree* left;
      struct s_tree* right;
}                    t_tree;
```

- You can manage node like this :
  - Command: example (“ echo hello ”, “ ls -la ”)
  - Pipe: example (“ | “)
  - OR and AND: (” && ” , “ || ”)
  - Redirect_input: example (“< in file ”)
  - Redirect_outfile: example (“ > out file ”)
  - Redirect_outfile (APPEND MODE): example (“ >> out file ”)
  - Heredoc: example (“ << Delimiter ”)

Resource :

https://www.youtube.com/watch?v=iddRD8tJi44&t=382s

https://www.youtube.com/watch?v=SToUyjAsaFk&t=603s

Expander

Executor

Write code to traverse the AST and execute the command:

- Once you have constructed the AST, you can traverse it and execute the command represented by the AST.
- To do this, you can write a recursive function that takes a node as input and executes the command represented by that node.
- For example, if the node represents a command, you would execute that command with any arguments and input/output files specified in the node. If the node represents a pipeline, you would execute the left child node and pipe its output to the input of the right child node.

```
Display a prompt when waiting for a new command.
Have a working history.
Search and launch the right executable (based on the PATH variable or using a relative or an absolute path).
Not use more than one global variable. Think about it. You will have to explain its purpose.
Not interpret unclosed quotes or special characters which are not required by the subject such as \ (backslash) or ; (semicolon).
Handle ’ (single quote) which should prevent the shell from interpreting the metacharacters in the quoted sequence.
Handle " (double quote) which should prevent the shell from interpreting the metacharacters in the quoted sequence except for $ (dollar
Implement redirections:
< should redirect input.
> should redirect output.

<< should be given a delimiter, then read the input until a line containing the delimiter is seen. However, it doesn’t have to update th
should redirect output in append mode.

Implement pipes (| character). The output of each command in the pipeline is connected to the input of the next command via a pipe.
Handle environment variables ($ followed by a sequence of characters) which should expand to their values.
Handle $? which should expand to the exit status of the most recently executed foreground pipeline.
Handle ctrl-C, ctrl-D and ctrl-\ which should behave like in bash.
In interactive mode:
ctrl-C displays a new prompt on a new line.
ctrl-D exits the shell.
ctrl-\ does nothing.
&& and || with parenthesis for priorities.
Wildcards * should work for the current working directory.
```

```
                     ||
                 /        \
        redirect_input     |
              /              \
          infile       redirect_output
                      /                \
                    cat              outfile
                   /   \
              pipe      command
              /  \        /    \
        command  cat     ls    command
                 /                 \
              outfile             cat
```

To-Do List for Make Minishell Project

Getting Path with Expand_env()

- [x] ~~Implement the `expand_env()` function to expand environment variables in the `PATH` variable.~~
- [x] ~~Update the code to use the expanded `PATH` variable when searching for executables.~~
- [x] ~~Test the function thoroughly to ensure it works as expected.~~

Checking for Leaks

- [ ] Use a memory leak detection tool, such as Valgrind, to check for memory leaks in the code.
- [ ] Fix any memory leaks that are found.
- [ ] Test the code again to ensure that there are no more memory leaks.

Fixing Segmentation Fault in Heredoc

- [ ] Identify the cause of the segmentation fault when using heredoc.
- [ ] Fix the issue by updating the code to handle heredoc properly.
- [ ] Test the code with heredoc to ensure that the segmentation fault is fixed.

Updating make execute_x() Function

- [x] ~~Convert linked list to a 2D char array or **char to make it easier to pass as an argument to the execute_x() function.~~
- [x] ~~Modify the execute_x() function to take a **char parameter instead of a linked list parameter.~~
- [x] ~~Update the code that calls the execute_x() function to pass the **char parameter instead of the linked list parameter.~~
- [x] ~~Test the updated execute_x() function thoroughly to ensure that it works correctly with the new **char parameter.~~
- [x] ~~Update any other functions that rely on the linked list parameter to use the 2D char array or **char parameter instead.~~
- [x] ~~Check for any memory leaks or errors that could be caused by the new implementation and fix them if necessary.~~
- [x] ~~Document the changes made to the code and ensure that the code is properly commented for future maintenance.~~

Close All Old Pipe To-Do List

Identify all open pipe file descriptors that are no longer needed in the program.

- [ ] Use the `close()` system call to close all the identified pipe file desc~~riptors.~~
- [ ] ~~Ensure that the pipe file des~~criptors have been duplicated or closed properly in all parts of the program to avoid potential errors or memory leaks.
- [ ] Test the program to ensure that it behaves as expected and all pipe file descriptors are being properly managed and closed.
- [ ] Document the code to make it clear how the pipe file descriptors are being managed and closed.

Optimized To-Do List

- Identify the areas of the system that need optimization.
- Establish performance benchmarks and metrics to measure the effectiveness of optimization efforts.
- Use profiling tools to identify bottlenecks and areas of inefficiency in the system.
- Prioritize optimization efforts based on their potential impact on overall system performance.
- Implement data caching and other techniques to reduce the amount of time spent accessing data from external sources.
- Use optimized algorithms and data structures to improve the efficiency of operations.
- Optimize database queries and indexing to reduce query times and improve database performance.
- Optimize network communication protocols and algorithms to reduce latency and improve throughput.

- Use parallel processing and distributed architectures to improve performance and scalability.
- Optimize memory usage by minimizing redundant data and freeing memory resources when they are no longer needed.
- Use hardware acceleration and specialized processors to improve performance in specific areas of the system.
- Continuously monitor and measure system performance to identify areas for further optimization.
- Document optimization efforts and their impact on system performance to inform future development efforts.
- Test and validate optimization efforts to ensure that they do not introduce new bugs or performance issues.
- Evaluate the trade-offs between performance and other system requirements, such as security and maintainability, when making optimization decisions.

Organize Project Files To-do List

- [ ] Create a main folder for the project.
- [ ] Decide on a clear naming convention for files and folders.
- [ ] Create subfolders for different types of files (e.g. code, images, documents).
- [ ] Create an "assets" folder for shared resources.
- [ ] Organize code files into logical folders based on their purpose (e.g. models, views, controllers).
- [ ] Create a "README" file with project information and instructions.
- [ ] Add a "LICENSE" file to protect your work.
- [ ] Use a version control system (e.g. Git) to track changes and collaborate with others.
- [ ] Create a "docs" folder to store project documentation.
- [ ] Implement a consistent file naming convention for easier file search and retrieval.
- [ ] Regularly review and clean up unused files and folders to avoid clutter.
- [ ] Backup important files regularly in a secure location.
- [ ] Consider using a task management tool to keep track of project tasks and deadlines.
- [ ] Document any changes made to the project in a change log file.
- [ ] Consider using a build tool to automate project tasks and streamline the development process.

Test for minishell

https://github.com/nenieiri-42Yerevan/Minishell

minishell-tester/test.sh at master · solaldunckel/minishell-tester

A basic tester for 42's new minishell. Contribute to solaldunckel/minishell-tester development by creating an account on GitHub.

https://github.com/solaldunckel/minishell-tester/blob/master/test.sh

solaldunckel/**minishell-tester**

A basic tester for 42's new minishell

1 Contributor 0 Issues 30 Stars 11 Forks