## Allowed Functions:

### Readline Library Functions

1. `readline`: Reads a line of input from the terminal with support for editing, history, and completion.
2. `rl_clear_history`: Clears the history of input lines.
3. `rl_on_new_line`: Tells the readline library that the cursor is on a new line.
4. `rl_replace_line`: Replaces the current line in the readline buffer with a new one.
5. `rl_redisplay`: Redisplays the current contents of the readline line buffer.
6. `add_history`: Adds the given string to the history list.

### Standard I/O Functions

7. `printf`: Prints formatted output to `stdout`.
8. `malloc`: Allocates a block of memory dynamically.
9. `free`: Frees previously allocated memory.
10. `write`: Writes data to a file descriptor.
11. `access`: Checks the accessibility of a file (e.g., whether it exists, whether it is readable, writable, etc.).
12. `open`: Opens a file and returns a file descriptor.
13. `read`: Reads data from a file descriptor.
14. `close`: Closes an open file descriptor.

### Process Management Functions

15. `fork`: Creates a new process by duplicating the calling process.
16. `wait`: Waits for a child process to terminate.
17. `waitpid`: Waits for a specific child process to terminate.
18. `wait3`: Waits for a child process to terminate and returns resource usage information.
19. `wait4`: Similar to `wait3`, but allows you to wait for a specific process.
20. `exit`: Terminates the current process.
21. `kill`: Sends a signal to a process.

### Signal Handling Functions

22. `signal`: Sets a handler for a signal.
23. `sigaction`: Used to change the action taken by a process on receipt of a specific signal.
24. `sigemptyset`: Initializes a signal set to exclude all signals.
25. `sigaddset`: Adds a signal to a signal set.

### File System Functions

26. **`getcwd`**: Gets the current working directory.
27. **`chdir`**: Changes the current working directory.
28. **`stat`**: Retrieves information about a file.
29. **`lstat`**: Similar to `stat`, but does not follow symbolic links.
30. **`fstat`**: Retrieves information about an open file.
31. **`unlink`**: Deletes a name from the filesystem, effectively deleting the file if it was the last reference.

### Process Execution Functions

32. **`execve`**: Replaces the current process image with a new one, specified by the path.
33. **`dup`**: Duplicates a file descriptor.
34. **`dup2`**: Duplicates a file descriptor to a specific value.
35. **`pipe`**: Creates a pipe, a unidirectional data channel that can be used for interprocess communication.

### Directory Handling Functions

36. **`opendir`**: Opens a directory stream corresponding to the directory name.
37. **`readdir`**: Reads a directory entry from the directory stream.
38. **`closedir`**: Closes a directory stream.

### Error Handling Functions

39. **`strerror`**: Returns a string describing the error code passed to it.
40. **`perror`**: Prints a description for the last error encountered.

### Terminal Handling Functions

41. **`isatty`**: Tests whether a file descriptor refers to a terminal.
42. **`ttyname`**: Returns the name of the terminal associated with a file descriptor.
43. **`ttyslot`**: Returns the slot number of the current user's terminal.
44. **`ioctl`**: Performs a variety of control operations on devices.

### Environment Functions

45. **`getenv`**: Retrieves the value of an environment variable.

### Terminal Attribute Functions

46. **`tcsetattr`**: Sets the parameters associated with the terminal.
47. **`tcgetattr`**: Gets the parameters associated with the terminal.

### Termcap Library Functions

48. `tgetent`: Loads the entry for a terminal from the termcap database.
49. `tgetflag`: Gets the value of a Boolean capability from the termcap entry.
50. `tgetnum`: Gets the value of a numeric capability from the termcap entry.
51. `tgetstr`: Gets the value of a string capability from the termcap entry.
52. `tgoto`: Computes a cursor movement string based on a capability string.
53. `tputs`: Outputs a string to the terminal, expanding padding information.

—-----------------------------------------------------------------------------------------------------------------

## Dictionary:

Fork - separated process

Child -

Terminal emulator - the window

Sh(Bourne shell) -  language of commands

Bash(Bourne again shell)/zsh(Zhong Shao shell) - more complex language of commands

execve()-runs only one command and quits, so we need to create new process via fork

there is no need for execve() for builtin functions

pipe - file where we put info, and from which we read info

we can't change where for example, ls writes info. So, we create a dup where it will write.

env has variables
it is defined in your configuration file
for example, PS1-prompt text. You can change the variable name.
check vars:
printenv | less
or
set | less

Environmental variables are dynamic values stored within a system or a user's environment that can influence the behavior of processes or applications running on that system. They provide a way to pass information into processes from outside the program. Environmental variables are used by the operating system and software to configure settings, determine file locations, and set preferences.

Bash:
includes many powerful features such as:

- Command history
- Job control
- Shell scripting capabilities
- Brace expansion
- Command-line editing

**Zsh:** includes many powerful features such as:

- Advanced command completion
- Spelling correction
- Highly customizable prompt
- Plugin system (e.g., Oh My Zsh)
- Improved scripting features, etc.

### Absolute Path:

- An **absolute path** starts from the **root directory**, denoted by `/`.
- It provides the complete path to a file or directory, no matter where you are in the file system.
  **Example**:
    - `/home/user/documents/file.txt`
    - `/usr/local/bin/script.sh`

### Relative Path:

- A **relative path** is based on the current working directory. It doesn't start from the root but instead from where you currently are in the file system.
- There are special symbols used to denote directories:
    - `.` refers to the **current directory**.
    - `..` refers to the **parent directory**.
- **Example**:
    - If you're in `/home/user` and want to reference `documents/file.txt`, you can use the relative path `documents/file.txt`.
    - To go one level up from the current directory and reference a file: `../file.txt`.

—------------------------------------------------------------------------------------------------------------------

## Subject Check list:

- [ ] Prompt(ms$)
- [ ] History
- [ ] Path
- [ ] Single quote
- [ ] Double quote
- [ ] < 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 the history!
- [ ] >> should redirect output in append mode.
- [ ] pipes
- [ ] environment variables ($ followed by a sequence of characters)
- [ ] $?
- [ ] ctrl-C, ctrl-D and ctrl-\ which should behave like in bash
- [ ] interactive mode:
    - [ ] ◦ ctrl-C displays a new prompt on a new line.
    - [ ] ◦ ctrl-D exits the shell.
    - [ ] ◦ ctrl-\ does nothing.
- [ ] builtins
    - [ ] ◦ echo with option -n (echo hello, echo -n hello, echo $PwD)
    - [ ] ◦ cd with only a relative or absolute path - change directory
    - [ ] ◦ pwd with no options - getcwd - current directory
    - [ ] ◦ export with no options - export [a][=b] - модификация и экспорт аргументов в другой shell
    - [ ] ◦ unset with no options - delete variable
    - [ ] ◦ env with no options or arguments - all variables
    - [ ] ◦ exit with no options - exit [n]   -> n%256

—-------------------------------------------------------------------------------------------------------------------

## Cases:

ms$ pwd (args)
ms$ pwd (args) |

ms$ wsd
command not found

ls dghjf
no such file or directory

./ls
no such file or directory

touch ls
./ls
permission denied

echo >
syntax error

|qrep

syntex error

```
echo $PATH
user/…

echo $PATH$qwe
user/…ls

echo $PATH $qwe
user/… ls

echo $PATHqwe
——

echo $PATH\qwe
user/…qwe

echo “   $PATH   “

echo ‘     $PATH    ‘

echo ‘ “   “ “     ‘

echo ‘   “$PATH”  ‘

echo “ ‘$PATH’  “

echo \t “\t” ‘\t’

echo \\t “\\t” ‘\\t’

echo “\”

echo “  ‘   ”
echo “     \’      ”

echo “  \$PATH   “

echo “$PATH” ; ls

echo ;

echo “;”

export qwe=1234 ; echo $qwe

./ls ; /bin/ls
```

---

## Pseudo code:

**main.c:**

```
Prompt message
Read
execute
```

**parsing.c**

```
errors:
command not found
no such file or directory
permission denied
syntax error
variables:
echo $PATH
echo $PATHqwe
echo $PATH\qwe
echo $PATH$qwe
echo $PARH $qwe
quotes:
echo “$PATH”
echo “ghj  $PATH  ghj ”
echo “ ‘ $PATH ‘ “
echo ‘$PATH’
echo ‘ghj  $PATH  ghj ‘
echo ‘ “ “ “ ‘
echo ‘ “$PATH” ‘
escaping(?):
echo \t “\t” ‘\t’
echo “    \”    “
echo “    \’    “
echo “     “    “
echo “  \$PATH  “
e”c”h”o hello
two and more commands:
ls ; ls
export qwe=123 ; echo $qwe
commands with path and without path:
without path:
ls - searches in $PATH
with path:
./ls - searches in current directory
pipe:
if there is | symbol, turn the flag on
```

**builtin.c**
KAMILLA:
echo: echo str(str\n), echo -n str(str), echo hello world(hello world\n), echo (\n)
cd: relative and absolut path
pwd: getcwd - current directory

TAHA:
env: list env vars
export:  list env vars with declare x; export pWd=0->adds a var; or modifies pWd=5;
unset: deletes a var
exit: exits;

**signals.c**
ctrl+c
ctrl+d
ctrl+\

forks.c(ls, etc.)
execve

pipes/redirect.c

history.c