Sabanci University

Operating Systems

CS 307

Programming Assignment - 1: SUShell Implementation with the LoopPipe Command

Release Date: 23 October 2025
Deadline: 3 November 2025 23.55

1 Introduction

In the lectures, you have learned that the Command Line Interface (CLI, in short, or shell) is a simple user program that can be implemented using `fork`, `wait`, and `exec` system calls. In the recitation session, you will see that the most fundamental command composition primitive of the UNIX-based operating systems, called *pipes*, can also be implemented inside a simple shell using some file manipulation utilizing `pipe` and `dup` standard C library methods.

In this Programming Assignment (PA), you will be implementing your own shell simulator called SUSHELL . In addition to executing simple CLI commands like `ls`, `wc`, `man` etc., your shell will allow piping commands to run more complex tasks and include a brand new command composition mechanism called LOOPPIPE. To make everything a little bit more difficult, file redirectioning using `'<'` and `'>'` at the end of commands will be possible. Details of the SUSHELL language in terms of both its syntax and semantics are provided in Section 2.

1.1 Motivating Example

Consider the following example that demonstrates the use of multiple SUSHELL features:

```
echo abcd | (rev | tr a-z A-Z)_2 > out.txt
```

This command consists of three main parts. The first part is a simple command `"echo abcd"`, which is supposed to print the string `"abcd"` to the console.

The second part, `"(...)_2"`, is a LOOPPIPE command. The number 2 at the end indicates that the piped command inside will iterate twice by giving the output of the first iteration as input to the second iteration. Moreover, the first iteration's input is given as the output of the first part, i.e., the `"echo"` command since the first part is piped with the second part.

The body of the second part is also obtained by piping two commands: `rev` reverses its input, and `"tr a-z A-Z"` replaces every occurrence of a small letter with a capital letter. So, when the second part starts its execution, it will take the string `"abcd"` as input. After the first iteration, it will output its reversed and capitalized version, `"DCBA"`. This string will be the input for the

second iteration and, after reversing and capitalizing again, the LoopPipe command will output the string `"ABCD"`.

The last part, `"> out.txt"`, redirects the second part's output `"DBCA"` to the file `out.txt` instead of the console.

As can be seen from this example, commands `SUShell` will understand and execute are complex. We strongly recommend you read the whole document at least once before writing any code.

Moreover, we suggest implementing SUShell incrementally by adding features one-by-one according to steps described in Section 6. Even if you cannot implement all functionality, you can get partial points from basic constructs.

2 SUShell Syntax & Semantics

This section formally defines the structure and interpretation of commands accepted by the SUShell . Each subsection introduces a specific feature of SUShell (basic commands, pipes, loops, and file redirections), accompanied by the corresponding grammar fragments and illustrative examples.

2.1 Basic Commands

A **basic command** in SUShell is any executable program or binary that can also be run in a regular Unix shell. Commands such as `ls`, `wc`, `man`, and `grep` are all valid in SUShell , as well as binaries specified by their absolute or relative file paths (e.g., `/bin/cat` or `./program`). Any number of options, flags, and arguments can be supplied to these commands.

Examples.

- `ls -l`
- `wc -l input.txt`
- `grep "error" logfile.txt`
- `/bin/cat data.txt`

Every command in SUShell is executed using the system call `execvp()`, which takes the command name and its arguments as an array of C strings.

Obtaining the input array by parsing (processing) an arbitrarily long and complex command can be difficult. Hence, we provide an API method that can do this job for you. The details of this method and how to use it in will be explained in Section 3.1.

2.2 Piping Commands

Pipes are one of the most fundamental composition mechanisms in Unix-based systems. They allow chaining multiple commands so that the **output of one command becomes the input of the next**. Internally, this behavior is implemented through the `pipe()` and `dup2()` system calls, which establish a unidirectional data stream between processes.

The partial grammar that enables piping in SUShell is:

$$Pipe \rightarrow Cmd\ PipeTail$$

$$PipeTail \rightarrow \texttt{"|"}\ Cmd\ PipeTail\ \mid\ \epsilon$$

This means that one or more commands can be combined using the pipe operator `|`. For example, the command `cmd1 | cmd2 | cmd3` executes three processes concurrently, connecting their input and output streams sequentially.

Examples.

- `ls | wc -l`
  This command lists the files in the current directory using `ls`, then sends the list to the next command through a pipe. The command `wc -l` counts the number of lines it receives as input. Therefore, the final output is the total number of files and directories in the current working directory.

- `cat file.txt | grep "error" | wc -l`
  In this example, the contents of `file.txt` are printed by `cat` and passed through two pipes. The first pipe sends the text to `grep "error"`, which filters and outputs only the lines containing the word `"error"`. The second pipe sends those filtered lines to `wc -l`, which counts them. As a result, the final output represents the number of lines in `file.txt` that contain the word `"error"`.

2.3 LoopPipe Structure

The **LoopPipe** is the extension introduced by SUShell , designed to extend the standard Unix piping mechanism. In simple terms, a LoopPipe allows repeating an entire pipeline multiple times in a row. The output produced by one iteration is automatically fed back as the input to the next, effectively creating a self-feeding loop over the same pipeline. This mechanism provides a new form of command composition that is not available in standard shells.

Formally, the syntax of a LoopPipe is defined as:

$$loop \rightarrow (\texttt{"("}\ pipe\ \texttt{")"}\ \texttt{"\_"}\ n)$$

where:

- `(` and `)` enclose the body of the pipeline that will be looped,
- `_` separates the loop body from the repetition count,
- and n is a single digit specifying how many times the loop should run.

At runtime, SUShell executes the pipeline in parentheses as a separate process sequence. For each iteration:

1. The output of the previous iteration is captured through an internal pipe.
2. That output is then used as the input for the next iteration.
3. This continues until the pipeline has executed exactly `n` times.

The LoopPipe structure can also appear as part of a larger pipeline, meaning it may have preceding or succeeding commands connected through standard pipes.

This flexibility enables nested compositions like:

- `cat file.txt | (grep error | sort)_2`
- `(rev | tr a-z A-Z)_3 | head -n 10`

where the LoopPipe operates as a single unit within a larger command sequence.

Examples.

- `(sed 's/a/aa/g')_2` : Replaces every `a` with `aa` twice, so each original `a` becomes four `a`'s in the final output.
- `echo abc | (tr a-z A-Z | rev)_2` : The text is converted to uppercase and reversed twice. The first pass gives `CBA`, the second reverses it back to `ABC`, showing the iterative effect of the loop.
- `cat words.txt | (sort | uniq)_3 | wc -l` : Reads words from a file, repeatedly sorts and removes duplicates three times, and then counts the remaining unique lines.

2.4 File Redirectioning

SUShell supports input and output redirection at the end of any command (simple or compound). This feature allows the user to replace the standard input or output streams of a command with files. The input redirection operator `<` causes a command to read from a file, while the output redirection operator `>` causes it to write to a file.

The grammar for redirection is:

$$
\begin{aligned}
\textit{OptRedir} &\rightarrow \textit{Redir} \mid \epsilon \\
\textit{Redir} &\rightarrow \texttt{">"}\ \textit{FNAME}\ \textit{OptIn} \mid \texttt{"<"}\ \textit{FNAME}\ \textit{OptOut} \\
\textit{OptIn} &\rightarrow \texttt{"<"}\ \textit{FNAME} \mid \epsilon \\
\textit{OptOut} &\rightarrow \texttt{">"}\ \textit{FNAME} \mid \epsilon
\end{aligned}
$$

This means that, redirection can include only one input redirection; or one output redirection; or both of them (order of input and output redirection doesn't matter); or as stated in the grammar, none of them.

Examples.

- `cat < input.txt`
- `wc -l > result.txt`
- `grep error < logfile.txt > errors.txt`

- `grep error > logfile.txt < errors.txt`

Each redirection modifies the file descriptors of the command before execution:

- Input files (`<`) are opened in read-only mode and attached to `STDIN`.
- Output files (`>`) are opened with the flags `O_WRONLY | O_TRUNC | O_CREAT` to enable writing, truncate existing contents, or create the file if it does not exist, and are then attached to `STDOUT`.

2.5 CLI Command Grammar and Token Definitions

In this assignment, we define a simple **Command-Line Interface (CLI) language grammar** that describes how a command can be written, combined, and repeated using pipes and loops. A **grammar** is a set of rules that formally specifies how valid commands are constructed. It provides a precise and unambiguous description of what forms of input our CLI simulator should accept.

Each rule in a grammar defines how smaller pieces (such as words or symbols) can be combined to form larger, valid structures—just as the grammar of a natural language defines how words form sentences. In this context, the smallest building blocks are called **terminals** (for example, command names, filenames, and symbols like `|` or `>`), and the abstract combinations of these terminals are called **nonterminals** (for example, *cmd* or *loop*). Together, they define a **Context-Free Grammar (CFG)** that can be used by a parser to check whether a given command line follows the correct structure.

The grammar described in Figure 1 provides the syntax of our CLI command language.

The grammar starts with a **Line**, which represents a single line of input that the shell can understand. A line can either be:

- the special quit command `:q`, which tells the shell to exit, or
- a normal command structure called **RedirCmd**, which may contain commands, pipelines, loops, and optional input/output redirections.

A **command (Cmd)** represents one executable program such as `cat`, `grep`, or `rev`. Several commands can be joined together with the pipe symbol

Start: $Line \rightarrow Quit \mid RedirCmd$

$Quit \rightarrow$ `":q"`

$RedirCmd \rightarrow CmpCmd\ OptRedir$

$OptRedir \rightarrow Redir \mid \epsilon$

$CmpCmd \rightarrow Loop \mid Pipe$

$Loop \rightarrow OptBefore$ `"("` $Pipe$ `")"` `"_"` $n\ OptAfter$

$OptBefore \rightarrow Pipe$ `"|"` $\mid \epsilon$

$OptAfter \rightarrow$ `"|"` $Pipe \mid \epsilon$

$Pipe \rightarrow Cmd\ PipeTail$

$PipeTail \rightarrow$ `"|"` $Cmd\ PipeTail \mid \epsilon$

$Cmd \rightarrow CMD$

$n \rightarrow DIGIT$

$Redir \rightarrow$ `">"` $FNAME\ OptIn \mid$ `"<"` $FNAME\ OptOut$

$OptIn \rightarrow$ `"<"` $FNAME \mid \epsilon$

$OptOut \rightarrow$ `">"` $FNAME \mid \epsilon$

Terminals:

CMD : A single command token (later split into argv), matches $[0-9A-Za-z.,\ ''\ ]+$

$FNAME$: $[A-Za-z\_][A-Za-z0-9\_.]*$

$DIGIT$: $[0-9]$

`"|"` : pipe `"(""")"` : loop body `"_"` : separator before count

`"<"` : stdin redirection `">"` : stdout redirection `":q"` : quit

ϵ : empty

Figure 1: Complete grammar for the SUShell commands.

(|) to form a **pipeline (Pipe)**, meaning the output of one command becomes the input of the next, just like in a normal Linux shell.

A **loop (Loop)** is a special structure of the form:

$$(\textit{before}\ |)?\ (\textit{loop\_body})\_N\ (|\ \textit{after})?$$

It allows repeating the pipeline inside the parentheses exactly `N` times. The parts before and after the loop (*OptBefore* and *OptAfter*) are *optional pipelines*—they may or may not exist. This means the grammar supports examples such as:

- `(rev)_3` : run `rev` three times,
- `cat | (tr a-z A-Z)_2` : run `cat`, then loop the uppercase conversion twice,
- `(tr a-z A-Z)_2 | head -n 1` : run the loop, then pass its result to `head`.

At the end of any command line, there can be **optional redirections (Redir)**, these specify which files to read from or write to. For instance:

$$<\ \textit{infile}\ >\ \textit{outfile}$$

means take input from *infile* and write the final output to *outfile*. The grammar also allows the reverse order (`> outfile < infile`), since both are logically equivalent.

Finally, the smallest elements—called **terminals**—are the actual symbols typed by the user: command names, filenames, digits, and special characters such as `|`, `<`, `>`, `( )`, and `_`.

Altogether, this grammar defines how a single shell line can combine simple commands, connect them with pipes, repeat them in loops, and redirect input or output files.

3 C Implementation

The first and most critical task for any shell is to understand the command a user has typed. This process of taking a raw string of text and converting it into a structured, meaningful representation is called **parsing**. This section details the parser library provided for your assignment, explaining its

structure, the data it produces, and the specific API functions you must use to interact with it.

Parsing is the process of analyzing a string of symbols to determine its grammatical structure. The parser uses a set of rules (a **Context-Free Grammar**) to break the input down. The result is an **Abstract Syntax Tree (AST)**, a tree-like data structure that represents the command's logical structure. For example, a command like `ls -l | (grep a)_2` is transformed into an AST like the one shown in Figure 1.

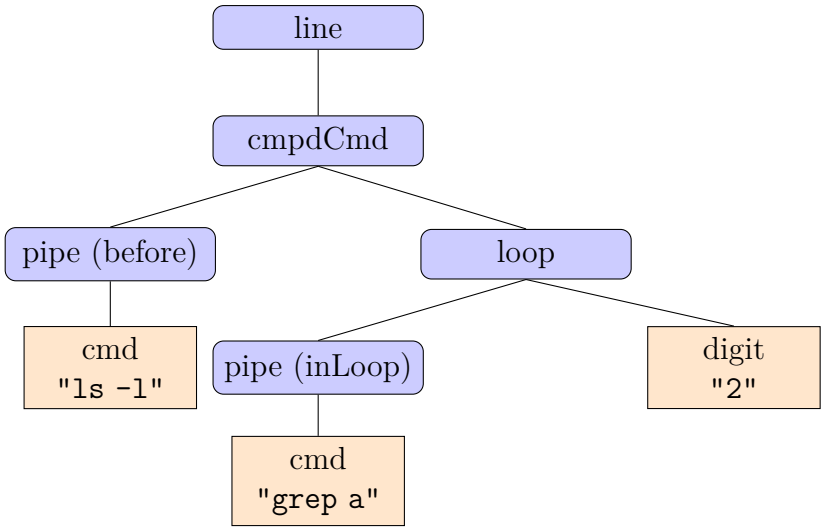

Figure 2: An Abstract Syntax Tree (AST) for the command `ls -l | (grep a)_2`.

The parser library we provide is designed with a clear, two-part structure:

1. A **Complex Command Parser**: The main engine that builds the AST from a given line and then partitions simple commands into *before loop*, *inside loop*, *after loop* arrays and identifies files for input and output redirectioning if they exist.

2. A **String-to-Argument-Vector Converter**: A component that takes basic command strings produced by the main parser and converts them into string array format required by `execvp`.

You will **not** write any parsing logic yourself. Instead, you will use the specific API functions described below.

3.1 Parsing Commands

For commands involving pipes or loops, the parser must analyze the entire command structure.

3.1.1 Internal Representation: The `sparser_t` Struct

The `sparser_t` struct is the core of the parser engine. It holds the definitions of the grammar used to understand your shell's command language. When you initialize the parser with `initParser`, this struct is filled with compiled grammar rules. You will **never** directly read from or write to this struct; you will only pass a pointer to it as an argument to the API functions. It is an internal, "black box" component.

3.1.2 The Main API and `compiledCmd` Struct

Your entire interaction with the complex parser will be through the `compileCommand` function and the `compiledCmd` struct it populates.

The `compiledCmd` struct is the clean, final output of the parser and the primary data structure you will work with and its definition is described below:

Listing 1: The CmdVec and compiledCmd definitions from parser.h

```
typedef struct {
    char ***argvs;
    size_t n;
} CmdVec;

typedef struct {
    CmdVec before;
    CmdVec inLoop;
    CmdVec after;
    size_t loopLen;
    char *inFile;
    char *outFile;
    int isQuit;
} compiledCmd;
```

In this struct, `before`, `inLoop` and `after` are two dimensional string arrays keeping a sequence of basic commands obtained from piped commands

before, inside and after the loopPipe command for the given line, respectively. If the line is a single piped command without a loopPipe construct, this command is stored in the `before` field, and other `CmdVec`'s are empty. At each index, a `CmdVec` keeps a string array that can be directly given to `execvp` as an argument.

`loopLen` field keeps the iteration count for the loopPipe construct if it exists. Similarly, `inFile` and `outFile` fields keep the names of input and output redirectionings respectively, if they exist. Lastly, `isQuit` shows whether this line is a quit command (`":q"`) or not.

`compileCommand` is the main method that transforms the line represented by `sparser_t` instance `sp` into the `compileCmd C` as described below.

```
int compileCommand(sparser_t *sp, char *input, compiledCmd* C)
```

- **Functionality**: This is the main entry point to the parser. It takes the initialized parser engine (`sp`), the user's raw input string (`input`), and a pointer to a `compiledCmd` struct (`C`) that you want it to fill. The function uses the grammar in `sparsert` to build an AST, traverses that tree, and extracts all relevant information into the simple, easy-to-use `compiledCmd` struct.
- **Return Value**: An `int`. A return value of **1** indicates a successful parse. A return value of **0** indicates a syntax error.
- **Example**: If the input string is `"echo hello | (tr a-z A-Z | rev)_2 > out.txt"`, after calling this function, the `compiledCmd` struct you passed in will be populated as follows:
  - `before`: a `CmdVec` struct containing argvs: `{{"echo", "hello", NULL}}` and `n: 1`
  - `inLoop`: a `CmdVec` struct containing argvs: `{{"tr", "a-z", "A-Z", NULL}`, {"rev", NULL }} and `n: 2`
  - `after`: a `CmdVec` struct containing argvs: `NULL` and `n: 0`
  - `loopLen`: 2
  - `inFile`: `NULL`
  - `outFile`: a string, `"out.txt"`
  - `isQuit`: an int, `0`

3.1.3 Remaining API Methods

`int initParser(sparser_t *p)`

- **Functionality**: Initializes the parser engine. It compiles the grammar rules and prepares the `sparsert` struct for use. This function **must** be called once at the very beginning of your program.

`void freeParser(sparser_t *p)`

- **Functionality**: Cleans up all resources used by the parser engine. This function **must** be called once at the very end of your program.

`void freeCompiledCmd(compiledCmd *c)`

- **Functionality**: Frees all memory associated with a `compiledCmd` struct, including the struct itself, its internal string vectors, and any filenames. This **must** be called after you have finished executing a command.

`void printCompiledCmd(compiledCmd *c)`

- **Functionality**: Prints the `compiledCmd` struct. You can use this function to understand the structure of compiledCmd and debug easily. You will not use this function in the final implementation of the homework.

`void initCompiledCmd(compiledCmd *c)`

- **Functionality**: This function is called by `compileCommand` directly. If you want to manually create a compiledCmd struct, you can use this function. The usage of this function is not necessary for your implementation.

3.2 SUShell Program Flow

Below is a general description of the program flow.

Parser setup. Before entering the main loop, the parser must be initialized using: `int initParser(sparser_t *p)`. If initialization fails, the program should terminate.

Interactive loop. Since SUShell is designed as an interactive shell, it continuously waits for user input. After the execution of the input command, shell waits for the next input. At each iteration, the shell prints its prompt such as `SUShell$` and flushes it. Then it reads the command using `getline`. If `getline` returns -1 (for instance, when the user presses `Ctrl+D`), the loop breaks and the program performs cleanup before exiting.

Parsing. The input line is passed to the parser using `compileCommand`. If the parsing fails, the shell prints `parse error`, frees the memory allocated for `C` using `freeCompiledCmd`, and continues to the next prompt without terminating.

File redirection. Before executing, the program checks for any input or output redirection operators (`<`, `>`). If such redirections exist, it opens the specified files. If a file cannot be opened, the shell cleans up and skips to the next iteration. If no files are provided, `STDIN` and `STDOUT` are used by default.

Execution. After the setup phase, you are expected to implement the execution algorithm that simulates how a shell runs commands. Each basic command must be run in a new process, and if two basic commands are connected through a pipe, you must arrange the file descriptors of processes corresponding to these commands such that one's output flows into the next one as its input.

Cleanup and resource management. After every command, the shell must free all allocated memory and close any open file descriptors. For this purpose, the helper function `freeCompiledCmd` should be used and file descriptors should be closed to ensure that no memory or file descriptors leak between commands.

Shutdown. When the user enters the quit command (`:q`) or the end-of-file signal is received, the shell terminates the loop, frees all remaining parser objects using `freeParser`, and exits by returning `0`. If the user enters the command (`:q`), terminate by printing `"Exiting shell..."` followed by a newline character.

4 Report Contents

In your report, you must answer the following questions explicitly, in separate sections.

- Explain your process hierarchy. During a pipe command execution, in which order does the parent (shell) process create command processes? Can they run concurrently? To what extent? Please, explain your answers over a figure.
- Please, describe the pipe structure you construct between two piped processes. How many pipes do you create per piped command? How do you make sure that the latter command gets the correct input from the former one? Can your pipes be accessed concurrently? If so, how do you ensure synchronization for concurrent accesses.
- How do you decide on the input of a loopPipe structure? What are the all possible sources of input redirectioning in a loopPipe structure? How do you differentiate these cases in your program?
- If there is another piped command after the looppipe command, how do you determine the input of the latter in your program?

Please note that if your answers in the report does not match with your code, you will not get any credits from your report.

5 Submission Guidelines

For this homework, you are expected to submit two files. The first one is a pdf that will contain your report. The name of this file must be *"report.pdf"*. The second file that you will submit is the code part of your assignment. Name of this file must be *"shell.c"* Below you can see the content of your files.

- shell.c: This file will contain your SUShell implementation.
- report.pdf: Your report that explains your code briefly. Please note that all the reports are read carefully to understand your program in a better sense. In your report, you are expected to make the instructor

understand such details as : What your program does, how your functions work (if there is any), what is the role of pipe on that function etc.

During the submission of this homework, you will see two different sections on SUCourse. For this assignment you are expected to submit your files separately. You should NOT zip any of your files. While you are submitting your homework, please submit your report to "PA1 – REPORT Submission" and your code to "PA1 – CODE Submission". SUCourse will not except if your files are at another format, so please be careful. If your submission does not fit to the format specified above, your grade may be penalized up to 10 points.

6 Grading

Dear students, your first programming assignment will be graded using automated test cases. Please keep in mind that even slight discrepancies on output will make you lose points.

- Compilation (10 pts): Your program compiles, runs and terminates without an error.
- Basic command execution (10 pts)
- Basic command with input or output redirection (10 pts)
- Piped command (10 pt)
- Basic looppipe command (20 pts): We will check both with and without redirectioning.
- Loppipe combined with pipes (20 pts): Again, redirectioning can be used for these commands.
- Report (20 pts): Your report has to meet the requirements described in Section 4 and be consistent with your implementation.

Each item is a precondition for the next one except the last item. It means that, if you do not get at least half of the points for an item, we will not test your programs for the next items.

Note that grading will be done automatically. Your shell's output will be redirected to a file and compared to the expected answer. If your shell name or whitespace characters differ, they will be considered incorrect.

Also note that commands containing characters not defined by the grammar, or commands that do not conform to the grammar, will result in a "parse error". Any cases where the parser produces a parse error will not be considered during grading.

7 Penalties

Dear Students, we would like to mark down some common mistakes for this Programming Assignment that will lead to point reduction.

- If your program is not a multi-process one, then you get 0.
- You have to use fork-exec couple for creating new process. Using other system calls like `"system"` is prohibited. If you do not obey this rule, you will get a 50 points penalty.
- Incorrect name usage (-5 pts): Please note that the c file you will submit must be named as `"shell.c"`.
- If your report is not consistent with your implementation, you will not get any points for your report and lose 20 overall points.

8 Sample Runs

8.1 Makefile, Compilation & Testing

Here is a one-time, detailed explanation of how to compile C files in a Linux terminal, in case some of you haven't done this before. We will cover what compilation is, how to do it manually, and how to use a `makefile` (the recommended way) to automate the process.

Briefly, compilation is the process of using a program (called a compiler, in our case gcc) to translate your human-readable C code (.c files) into a single, machine-readable executable file that the operating system can run.

First, you must add the following line to the top of your `shell.c` file, along with your other includes (like `<stdio.h>, <unistd.h>,` etc.). This line gives your `shell.c` file access to the functions and structures defined in the parser.

```
#include "parser.h"
```

To create your shell, you need to compile all three source code files (`mpc.c, shell.c, parser.c`) together into one executable. You can do this by running the following command on your terminal. To be able to run this command, all those three files must be under the same directory.

```
> gcc -o shell shell.c parser.c mpc.c -Wall -Wextra
```

By running this command, you can see the executable named `shell`, under the same directory with your source codes. After you generate the executable you can run it and start your testing with this command.

```
> ./shell
```

Instead of performing all these steps manually, you can use the provided Makefile to easily compile your source code. To do this, place all the files in the same directory. Remember, all your files must be named correctly for the Makefile to work properly. Then, simply run the following command:

```
> make shell
```

This will basically do the same operation with manual command and generate an executable file.

8.2 Sample Runs

Sample Run 1

Your simulated shell should be able to run basic commands just like the real one, here we will test some simple ones. Once again you are not responsible of the commands that parser is unable to parse.

Command: `./shell`

Terminal Output:

```
SUShell$ echo "hello cs307"
hello cs307
SUShell$ seq 1 3
1
2
3
SUShell$ uname -s
Linux
SUShell$ printf "dont try a new line"
dont try a new lineSUShell$
SUShell$ :q
Exiting shell...
```

Sample Run 2

Like the original one, your SUShell should be able to utilize basic pipe commands. Here we will try their execution.

Command: `./shell`

Terminal Output:

```
SUShell$ echo 'hello' | tr a-z A-Z
HELLO
SUShell$ seq 1 3 | wc -l | tr -d ' '
3
SUShell$ uname -s | tr a-z A-Z
LINUX
SUShell$ yes OK | head -n 3 | tail -n 1
OK
SUShell$ echo 'a,b,c' | cut -d, -f2
b
SUShell$ echo 'Hello World' | tr A-Za-z N-ZA-Mn-za-m
Uryyb Jbeyq
SUShell$ echo 'abcdef' | rev
fedcba
SUShell$ seq 1 3 | sort -r | paste -sd- -
3-2-1
SUShell$ :q
Exiting shell...
```

Sample Run 3

Like the original one, your SUShell should be able to perform input/output re-directions. Below are the input.txt, terminal outputs and output.txt that you will be seeing in your executions as well. input.txt should be prepeared as below before running your commands!

Command: `./shell`

input1.txt:

```
red
green
blue
```

Terminal Output:

```
SUShell$ tr a-z A-Z < input1.txt
RED
GREEN
BLUESUShell$ cat | wc -l | tr -d ' ' < input1.txt
```

```
2
SUShell$ head -n 2 | tail -n 1 | tr a-z A-Z < input1.txt
   > output1.txt
SUShell$ :q
Exiting shell...
```

output1.txt:

```
GREEN
```

Sample Run 4

Apart from the regular shell, your core functionality of the SUShell(which is a looped pipe) should also be working.

Command: `./shell`

Terminal Output:

```
SUShell$ (echo 'Hello World' | tr A-Za-z N-ZA-Mn-za-m)_2
Uryyb Jbeyq
SUShell$ echo 'abcd' | (rev)_2
abcd
SUShell$ echo 'abcd' | (rev)_1
dcba
SUShell$ (yes OK | head -n 3)_2 | tail -n 1
OK
SUShell$ echo 'a,b,c' | (tr , " ")_1 | wc -w | tr -d ' '
3
SUShell$ (seq 1 5 | paste -sd- - | rev)_2
5-4-3-2-1
SUShell$ :q
Exiting shell...
```

Sample Run 5

Like the original one, your SUShell should be able to perform input/output re-directions, but sometimes you may be required to re-direct your looped pipe to an output file, or take its inputs from an input file. We have given

basic sample executions for those cases below. There are the input.txt, terminal outputs and output.txt that you will be seeing in your executions as well. input.txt should be prepeared as below before running your commands!

Command: `./shell`

input2.txt:

```
alpha
beta
gamma
```

Terminal Output:

```
SUShell$ (tr a-z A-Z)_1 < input2.txt
ALPHA
BETA
GAMMASUShell$ cat | (tail -n 1)_3 < input2.txt
gammaSUShell$ (head -n 2 | tr a-z A-Z)_1 < input2.txt >
   output2.txt
SUShell$ :q
Exiting shell...
```

output2.txt:

```
ALPHA
BETA
```