Team: p23

Before starting each cycle of development, our team always discussed how to divide the tasks, and created GitHub issues for TODOs. Therefore, all team members evenly contributed to this project.

Contribution to parsing: We used Lark as a parser in this shell program. We tried both Lark and ANTLR, but we found that Lark is simpler and has more detailed documentation. Because Lark has various tutorials which explain the entire process of using it and it supports EBNF (Extended Backus-Naur form) and regex style in the grammar, we were able to implement more concise grammar faster than ANTLR. In terms of implementing the logic to process the parse tree, we used the Visitor pattern which enabled the separation of algorithms so that we can focus on algorithms themselves. We also considered using the Transformer pattern to process it because it looked like we can simplify our algorithms using that by replacing the nodes of literal value with the actual values; however, it is slightly less efficient than Visitor, so we ended up using Visitor and completed the logic using a deque inside the tree processing.

Contribution to shell features: There are three new shell features in our program. At the end of the project, we decided to add one more feature per person to make our shell close to the one we use such as zsh. Harryn, Youngwoo and I (Dongyeon) took wc application, history application and syntax highlighting, respectively. Our wc application is almost the same as the existing UNIX command, but it is different in that we only show the total number of lines, words, and characters even if multiple files are specified. It supports three flags (-l, -w and -m) to check the number of lines, words, and characters, respectively. The history application was added as we support loading previous commands with up or down arrow keys. It stores up to 100 previous commands. Lastly, the syntax highlighting colours registered commands (e.g., cd) green, red otherwise. Using regex, it works well even if there are multiple '|'s or ';'s. The most challenging part was to create a new input system, instead of using input() function. Our shell gets a single key input in real-time so that the change in colouring is displayed immediately as the user types. Thanks to the new system, we were also able to support up or down arrow keys to load the previous commands on the command line.

Contribution to design: We used a singleton pattern for history and history manager classes. The history class is an application that saves previous commands, and the history manager class is for controlling arrow-up or arrow-down keys to retrieve previous commands from the history class. To minimise memory use and keep the data of command history, they should have one object each, so we used this design. Essentially, this design has a drawback in that an instance lives until the program ends, but it rather fits our need as they must manage the command history until the program terminates.

Contribution to workflow/CI: The Git workflow we set was influenced by Gitflow. We set a rule that we should not directly push to the main branch, instead, we should create a new branch and then merge it into the main branch through Pull Request (PR). It enabled all team members to read the others' code at least once and find any errors, misunderstandings, or improvements. Plus, we added a branch protection rule in the GitHub repository setting so that there must be at least one member to review a new PR. For CI, we added GitHub Action to run all tests we have such as unit tests, system tests, and code analysis whenever a new PR is merged into the main branch. As a result, our team was able to keep our code quality consistent and bug-free when merging different branches.