## Team: p31

Our code used several design patterns to ensure clarity and scalability. For example, we used the factory design pattern to instantiate specific ‘Application’ subclasses and pass in the arguments, stdin and stdout that it expected. We also utilized the wrapper design pattern to implement unsafe versions of UNIX applications. The wrapper class would receive an ‘Application’ subclass and run its main function in a try-except block, capturing any exceptions and printing them to stdout. All these design choices made it extremely easy to add applications or modify their functioning, since they could just be polymorphically called by the factory and making them unsafe was implicitly handled by the wrapper.

We also managed to implement bash’s precedence when it comes to stdin and stdout. For example, if a command receives stdin from a pipe as well as a redirection, it will give precedence to the redirection. Another simple but effective feature we implemented was the Bash convention of ‘username@hostname’ followed by current directory as the shell prompt.

For the parser, we used the popular ANTLR4 parser generator. We wrote the BNF grammar, specifying useful tokens such as quotes, whitespace, text in the lexer section and making sure to avoid common pitfalls such as left recursion in the parser section. We then used the visitor design pattern to visit each node in the parse tree and perform certain actions such as creating a specific command (call, pipe, sequence), evaluating backquotes, expanding globs, evaluating redirections etc. ANTLR made it extremely easy to focus on the correctness of parser e.g., ensuring quote escaping worked, ensuring the right command was called, ensuring the correct stdin and stdout was passed, instead of having to focus on the mechanical task of manually creating the parser itself.

For the individual contributions, Chuan Yan contributed 0%, Yidan Zhu contributed 30%, implementing half of the UNIX applications, and Aaryaman Sharma contributed 70%, doing the other half of the UNIX applications, and everything else including creating the main program loop, writing the ANTLR grammar and parser, implementing a visitor for the parse-tree, creating classes for the different types of command, implementing the unsafe application wrapper, writing tests and modifying analysis, coverage and testing scripts as required.

## Reviewed submission: f174430fbf97a7563aecf7b327b1cf5d5cd0db28acc74ddb1c7ca032d291fb26

### Design Score: 3

The program is well-designed and uses several design patterns and good design principles. For example, an abstract class 'Application' is used to abstract away the details of the specific UNIX application (cat, cd, ls etc.) and allows for consistent interaction with all the subclasses via polymorphism. This abstract class also acts as a template for some common and unchanging methods, such as outputting to stdout, reducing the amount of code repetition. Similar design patterns are used for the 'Command' class. The code also makes good use of the factory design pattern to convert the parsed command-line text to a concrete instance of the 'Application' class. However, when dealing with unsafe applications, the factory class passes a flag to the specific instance of the 'Application' class. This means that each UNIX application implementation must deal with receiving this flag and printing exceptions to stdout, cluttering the code. Instead, they could have used a wrapper class to capture all exceptions.  In terms of the parser, the design is based on a monadic parser. The implementation is well considered, with good implementation of functional programming concepts such as binding, monads, functors, higher-order functions etc. However, this might have overcomplicated the parser, and a simpler recursive descent or ANTLR generated parser would have sufficed, while making the code much simpler to read and debug.

### Code Quality Score: 2

The style of the code is well written and follows python conventions regarding line-length, indentation, new-lines etc. While there are a few exceptions, especially with line-length, it doesn't detract too much from the clarity of the code. However, documentation is a huge problem in the code. Test cases and huge unused code fragments are commented out, for example in the parsing.py file. This greatly obfuscates the code and makes it difficult to distinguish between miscellaneous debugging notes and useful documentation. Furthermore, the actual documentation is lacking, especially for the parser, making it hard to follow along with the code. Another issue is that the packages are not very well structured. For example, the command package contains not just files for the commands, but also for the parser, app factory etc. It would have been better to split up these files to ensure that each package achieves a specific purpose. Another issue with the code quality is with the consistency of type annotations. Some parts of the code such as the parser are fully committed to type annotations, even introducing generics and typevars for increased clarity. However, other parts of the code are completely lacking in type annotations, making the transition from one file to another quite jarring. The tests are generally well written and cover many edge cases. However, the project has permanent directories and files used for testing, which clutters up the project directory and makes it hard to scale. They could have used features such as setUp and tearDown in testing frameworks instead.

### Error Handling Score: 3

There are custom error classes for several types of errors, making it very clear to the user and other programmers what errors are being encountered. However, the custom error handling in the main.py file is unnecessary, since the added message adds no extra information. Also, it doesn't adhere to the idea of raising errors as exceptions, since all of them are printed to stdout. This also undermines the function of the unsafe version of the app, since all exceptions are printed to stdout regardless.

## Reviewed submission: c3eff4e653984390fec36c3642ca007fe5be9ac51f4a1eb9afd4e1c625ece87c

### Design Score: 4

The application is very well-designed and utilizes several good design patterns and design principles. It uses the abstract class ‘App’ to abstract specific UNIX applications, allowing interactions with them via polymorphism. It also provides a sensible __init__ method that all the subclasses use to initialize the arguments and stdin. The code also makes good use of the factory design pattern to instantiate the specific UNIX application based on the parser output, and to instantiate the specific type of command (call, pipe, sequence). The way the factory deals with unsafe applications is also well-designed. It uses the decorator design pattern to pass the exec function of any ‘App’ subclass into a modified exec function with a try-except block that prints the exceptions to stdout. This standardizes the handling of unsafe apps, preventing repetition, and allows for the UNIX applications to focus on solely on their main operation. The design of the parser is also well-done, utilizing the popular ANTLR parser generator. The BNF grammar is specified correctly in the ShellGrammar.g4 file and the lexer tokens are well-defined. To deal with the ANTLR generated parse tree, the visitor design pattern is used to visit the nodes of the parse tree and create ‘Command’ instances, handle quotes, execute backquoted commands, expand globs etc. The program generally has a very sensible control flow from parsing to creating commands to the using the app factory to the eventual execution of the UNIX application itself.

### Code Quality Score: 5

The code is extremely well-written and adheres strongly to python conventions regarding line-length, indentation, new-lines etc. Class, variable and function names are also very clear and make the functioning of the program easy to understand. The documentation is also very well done, with detailed docstrings for the functions detailing the uses, parameters, expected return values, potential exceptions and a lot more. Even within functions, there are a lot of comments that illuminate their purpose and add a lot of clarity to the code. Type annotations are also used consistently throughout the code, which is extremely helpful when trying to understand the parse tree visitor and the way the program reads from and writes to stdin and stdout. The package structure is also excellent. Each package achieves a specific purpose, and the files and packages are sensibly named, allowing other programmers to quickly identify and understand how a specific part of the program functions. The tests are also extremely well written. There are no permanent files and directories used for testing, instead using setUp and tearDown functions. The program also uses frameworks such as hypothesis to generate well-defined randomized inputs, making the tests very comprehensive. The tests itself are split up into distinct packages, each testing a distinct component of the program (parser, command classes, application classes), making it very easy to debug major components of the program separately.

### Error Handling Score: 4

There are many custom exceptions that deal with arguments, flags, applications etc. This greatly improves clarity and makes error handling more specific, preventing insidious errors from slipping through. Several error prone operations such as reading/writing to files are written into functions that contain try-except blocks to catch errors. This prevents littering try-except blocks everywhere since you can just call these functions and implicitly deal with exceptions.