Commander

Design Document

Cayden Lund (u1182408)
Eduardo Valdivia (u1105168)
Jaden Gill (u1259060)
Preston Hales (u1171135)

CS 4000—Senior Capstone Design
23 October 2023

1. Executive Summary

Commander was born from the frustrations we experienced with traditional scripting environments, especially those relating to command line automation. Common command line scripting environments include Bash and Powershell. While great for their direct connection to the command line and using/parsing command outputs, they also have a limited set of actions the user may perform and unintuitive syntax. As for Bash, the language is not strongly typed which makes things confusing for programmers coming from a more structured language, and makes tasks such as doing math more involved due to having to parse strings. In comparison, programming languages targeted toward a general audience like Python and Perl are powerful with better syntax and typing, but lack the simple directness of a command line scripting environment and easy parsing of output from commands.

The first of many differences with Commander compared to its contemporaries is the language's syntax. The goal of Commander is to be a simple language with an interpreter, like Python, but include additional syntax for user clarity. Much of the syntax takes inspiration from C and its derivative languages like C#, Java, and Javascript since these languages are pretty familiar to most programmers, thus allowing for a less steep learning curve for people new to Commander. Commander allows users to quickly develop scripts for their projects while maintaining readable code. Moreover, language interpreters such as Python's are notorious in certain groups for a variety of reasons, often low performance. The Commander interpreter is written in C++ for faster interpretation and execution time.

Similarly, scripting environments are primarily exclusive to a specific platform. Powershell, for instance, is included on Windows by default as the go-to scripting environment. While it is possible to install and run Powershell on Linux, one must go through the process of installation, and even once installed will still have to use Bash to utilize certain dependencies tied exclusively to it. One benefit of Commander compared to other environments is its ability to be run on multiple platforms, including Windows, macOS, and Linux, via the multi-platform interpreter and run commands independent of the primary scripting environment. Additionally, no additional dependencies will be required to run the interpreter using the pre-built executables that will be released for each version of Commander on the GitLab page. Upon completion, the project will also be released as open-source under the MIT license, allowing other users to compile the application on their machines or make contributions as they feel inclined. If time permits, we will also create plugins for various text editors and IDEs in order to provide nice features including syntax highlighting for the Commander language. The result is ultimately an environment that can be built and run anywhere the user would like with the tools needed to easily program their Commander scripts.

Our goal for Commander is to provide a scripting environment which resolves the aforementioned problems that present implementations have. Users will want to use Commander due to its accessibility and ease along with the benefits of a traditional programming language. It is our hope that Commander will enable veterans and new command-line users to develop the command line scripts that they desire.

2. Background and Technical Requirements

a. Similar Languages

Projects similar to ours include *AngelScript*, *Bash*, *Batsh*, *Lua*, *Perl*, *PowerShell*, and *Python*. A brief overview of each of these languages is provided below.

Most of the languages we have observed attempt to provide general purpose features rather than emphasize command execution. Meanwhile, the other scripting languages for command-line programs tend to have unintuitive syntax with a steep learning curve. In the case of Batsh, the creators only provide a transpiler, which translates Batsh into Bash and Windows Batch scripts, so the code cannot be directly executed across multiple operating systems as it can with an interpreter. The goal of Commander is to be a scripting-first language that users will find simple to understand, while offering a speedy interpreter that will be able to execute commands across the three main operating systems: Windows, MacOS, and Linux. A summary of the languages we have observed are as follows:

AngelScript: A flexible scripting library designed to be functional through external scripts. It was designed to be easier for both application and script writers. It implements types like C++ and has similar syntax. A caveat we have identified is AngelScript's usage in game development. In other words, the language is designed with a specific use case in mind.

Bash: Most commonly used in Linux systems, Bash is the GNU Project's shell (*B*ourne *A*gain *SH*ell). It encompasses both a shell environment and a scripting language that allows for users to automate their daily tasks through the shell. Its steep learning curve and confusing syntax may result in unreadable script files. Additionally, it is weakly typed, which makes certain things like math operations more difficult to program.

Batsh: A command-line scripting language with C-like syntax which transpiles (i.e. translates), to either Bash or Windows Batch. There exists an online tool provided by the creators as well which allows you to write a Batsh script and it will output the resulting

Bash/Batch translation. However, it lacks an interpreter or other methods for directly executing the code.

Lua: A scripting language that supports object-oriented programming, functional programming, data-driven programming, and data description. It is used in a wide range of applications like games, web, and image processing. Similarly to Python, Lua is often used as a general-purpose programming language. The language's interaction with the command line for any given operating system is unrefined.

Perl: A general-purpose interpreted scripting language. Its own philosophy is to enable developers to quickly create and test their code. Like most general-purpose scripting languages, interaction with the command line may be indirect or discouraged.

PowerShell: A modern shell-scripting language developed by Microsoft. It is similar to Bash in many respects, and shares some of the same drawbacks. In short, it can directly execute commands well, but its syntax is verbose and not very readable, and it is primarily limited to Windows. (It can run on other platforms, but has some compatibility issues and many dependencies.) One of PowerShell's benefits over Bash is that it is strongly typed, which makes it easier to avoid bugs and do certain things like math.

Python: A general purpose programming language with high emphasis on code readability. The language is very general-purpose, though, and lacks the directness for automating the command line (that is, it takes a lot of code just to run a single command). Other criticisms include its generally slower performance compared to other languages and its used of indentation for defining scopes.

b. Required Technology

While we develop Commander, we intend to use the following technologies:

C++: We plan on programming the interpreter in a low-level language, both for direct access to system files and streams and for runtime efficiency. We selected C++ as everyone in our team is familiar with the language.

Unit Testing: We plan on writing extensive unit tests in order to ensure the accuracy of the Commander interpreter. These tests will use the GoogleTest unit testing library for ease-of-use, reproducibility, and reliability.

File I/O and Commands: The interpreter will require access to and utilize the file system on the operating system of choice when it is run. Not only will this be required to

read in and run the script files, but to also find and execute the referenced commands that are stored as executable binaries in the system.

Text Editors/IDEs: We plan on utilizing powerful text editors and IDEs such as Vim, Emacs, VSCode, and CLion in order to program the interpreter. These editors often provide many tools and plugins such as Intellisense, syntax highlighting, and debuggers to help make programming easier and less error-prone. If time permits, we will write plugins for these editors in order to help people programming in Commander to have access to some of these same features.

GitLab/GitHub: We will use GitLab for version control. GitLab has many tools for managing features and goals the project will have. Upon completing CS4500, we will move the repository to GitHub as an open source project for future version control. This choice is due to the sheer number of users working with GitHub, which will allow more developers to have easier access to the project.

c. Software/Hardware Requirements

In order to compile and build the interpreter, a C++ compiler such as gcc or clang is required. However, pre-built executables will be provided, so there are no software requirements or dependencies that clients will need to install. Commander will be multi-platform; that is, it will run on Windows, macOS, and Linux computers. The development focus will be on making it work on x86-64 architectures, but it will likely be portable to ARM or other architectures as well. There are no further anticipated hardware requirements.

# 3. Requirements Analysis

System Architecture

This project will be structured as a series of stages. The output of one stage becomes the input for the next stage, and through a step-by-step process, the script is parsed and then interpreted.

First, the input script file is fed into the lexer as a series of bytes. The bytes are then lexed into *tokens,* which are essentially the individual units of keywords, punctuation, variable names, and numbers. These tokens are fed to the parser. The parser examines these tokens and parses them into *Abstract Syntax Tree Nodes*, which have meaning. These AST nodes form a series of commands, which can be interpreted and run. Before that, the AST nodes are sent to the type-checker, which ensures that expressions are well-formed

and variables are defined before use. The following diagram represents the parsing workflow of the interpreter:

After this workflow is complete, the commands are ready to be interpreted, and each node in the series is executed one by one. The interpreter is composed of a few different pieces. First, the flow controller executes the AST nodes as they are encountered. Things like conditional statements, function calls, and loops will be interpreted by the flow controller, and the location in the series of AST nodes will be updated appropriately. The symbol table tracks the values of variables, and stores the code of functions. The job runner will run commands on the command line, and will manage advanced operations like pipes, input/output redirection, and running sub-processes in the background.
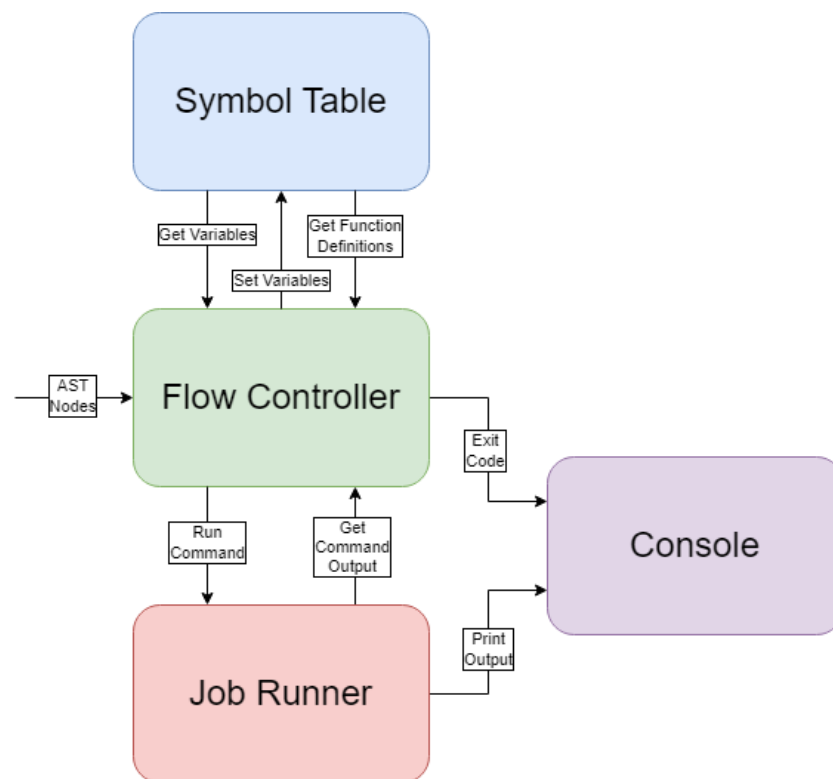
Personnel

The following list details the required modules, as well as the team members' responsibilities for seeing these modules to completion:
- **Formal language specification:**
- **Lexer:**

- **Parser:**
- **Type-checker:**
- **Flow controller:**
- **Job runner:**
- **Symbol table:**
- **Unit testing system:**
    - **Preston Hales**
- **Syntax Highlighter Plugins (stretch goal):**
- **Transpiler (super stretch goal):**
- **Shell (super mega stretch goal):**

System Features

The core infrastructure and functionality of our project includes the Commander language specification and the interpreter. First, the language must be well-defined; without a well-defined language, the project will lack a unified direction, and we won't get a functional end product. In addition to the language design, the interpreter must be working on at least a basic level in order for the language to be of any use. In this manner, the language specification and the interpreter are co-dependent, and both of these will be our highest priority with the project.

Beyond a formal language specification and a basic, functioning interpreter, we plan to implement some of the more interesting language features, including functions, lambdas, loops, and string interpolation. We also plan on architecting the interpreter to be flexible enough such that in the future we will be able to easily add or update language features, especially since it will one day be open source.

Once we have implemented the entire language specification in our interpreter, we can move on to our more advanced feature: IDE integration. We plan to write plugins for popular IDEs, including CLion and Visual Studio Code, among others, which will provide syntax highlighting. Additionally, if we still have more time, we will work on adding in a transpiler for translating Commander into Bash and/or Powershell, and creating an interactive shell or interactive interpreter that will execute Commander code line by line as entered by the user.

4. Appendix

Use Cases

The tables on the following pages are dedicated to use cases the Commander team has developed. Use cases describe a variety of situations we expect users to encounter in

the finished software. These cases do not reflect the total number of situations a user may encounter, rather aim to describe how a situation will be handled.

To organize each use case, a table will be provided with a letter and number. Numbers will determine the category a use case is organized into, while letters serve as individual cases. For instance, section 1 is dedicated to expected inputs. The first section assumes no errors are present in the user's scripts. Alternatively, section 2 provides some insight into documentation and interaction with other software. Section 3 describes stretch goals; features we intend to implement if we have time in CS 4500. Finally, section 4 describes error handling.

Number	Use Case 1.A.
Title	**Running Script**
Preparer	Preston Hales
Actor/User	Commander Scripter
User Story	As a Commander Scripter, I want to be able to write a Commander script in a separate file and pass that file into the interpreter to run the script. In doing so, I can rerun the same script multiple times or in other places.
Course of Events	1. Create a file. 2. Write Commander script. 3. Save the file. 4. Run interpreter with file path as an argument.
Exceptions/Alternates	None
Related UI	**See Sketch 1.A.**

Number	Use Case 1.B.
Title	**Multiple Scripts and Code Reuse**
Preparer	Preston Hales

Actor/User	Commander Scripter
User Story	As a Commander scripter, I want to be able to write multiple Commander scripts in different files and import the code (i.e., functions and variables) from one script into another to practice code reuse.
Course of Events	1. Create and write multiple scripts in separate files. 2. Write a script file that imports one or more other scripts, which contain functions and variables. 3. Proceed to use those functions and variables in the current file. 4. Run the main script, using functions and variables from other scripts.
Exceptions/Alternates	• An included script file has an error—in this case, the error will be printed along with the filename.
Related UI	**See Sketch 1.B.**

Number	**Use Case 1.C.**
Title	**User Input in Commander**
Preparer	Eduardo Valdivia
Actor/User	Computer User
User Story	As a computer user, I want to be able to automate simple tasks. I want a program that can count the number of files in my Download folder and ask me whether I want to empty the contents in that folder.

Course of Events	1. Write a script in the Commander language using the **read** and **if** keywords to read user input and act on it. 2. Save the file. 3. Run the script with the Commander interpreter.
Exceptions/Alternates	None
Related UI	**See Sketch 1.C.**

Number	**Use Case 1.D.**
Title	**OS-specific Commands**
Preparer	Jaden Gill
Actor/User	Command Line User
User Story	As a command line user, I would like to be able to run a script on multiple operating systems, running the related commands on each operating system based on defined aliases so I don't have to rewrite the script to use different commands.
Course of Events	1. Create a script using aliases. 2. Run command in Linux, and use the corresponding Linux command. 3. Run command in Windows, and use the corresponding Windows command.
Exceptions/Alternates	● No command alias exists on the target platform - see 4.D.
Related UI	**See Sketch 1.D.**

Number	**Use Case 1.E.**
Title	**Loops in Commander**

Preparer	Eduardo Valdivia
Actor/User	Commander Scripter
User Story	As a Commander scripter, I would like to be able to run a command multiple times so I don't have to write it over and over.
Course of Events	1. Create a new file using a text editor. 2. Write a script using the **for** or **while** keywords.
Exceptions/ Alternates	• Do-While loops, implied loops, etc.
Related UI	**See Sketch 1.E.**

Number	**Use Case 1.F.**
Title	**Functional Programming**
Preparer	Preston Hales
Actor/User	Functional Programmer
User Story	As a functional programmer, I would like the ability to be able to store functions in variables and pass those functions into other functions as parameters in order to allow for better code reuse. Additionally, lambda expressions would be nice for defining functions on-the-fly.
Course of Events	1. Define function that use parameters as functions 2. Call those functions in that function 3. Call that function with either variables that store other functions, or by using a lambda expression to define a function for use in that one place.

Exceptions/Alternates	• None
Related UI	**See Sketch 1.F.**

Number	**Use Case 1.G.**
Title	**Advanced Command Execution Features**
Preparer	Preston Hales
Actor/User	Advanced Command-line Programmer
User Story	As an advanced command-line programmer, I'd like the ability to call commands with certain advanced features available in other command-line scripting languages (e.g. running commands in the background asynchronously), or even new features entirely, in order to give me more power to do what I need/want to do.
Course of Events	1. Run a command that can run in the background (i.e., asynchronously). 2. Run a command whose output can be piped into the input for another command. 3. Run a command and store output into a file. 4. Run a command, taking input from an existing file. 5. Run a command, and utilize the output of that command throughout the script (by storing it into a variable, or using it in some operation/expression). 6. Run a command that can output either standard output, error output, or both, to a file. 7. Run a command, and timeout the command if it takes too long. 8. Run a command, and detect if it succeeded or failed. 9. Run a command, and be able to specify what type the output is (e.g. a string, a boolean, an int, etc.).
Exceptions/Alternates	Possibly many other advanced features not currently listed here.
Related UI	**See Sketch 1.G.**

Number	Use Case 1.H.
Title	**Scriptless Execution**
Preparer	Cayden Lund
Actor/User	Terminal User
User Story	As a terminal user, I want to be able to write a short script of commands to execute without creating a temporary script file.
Course of Events	1. Write a short script of commands to execute in the format of a string. 2. Invoke the Commander executable with that string script as an argument. 3. The Commander executable interprets that argument as if it were read from a file.
Exceptions/Alternates	• If the script is malformed, errors will be reported in the same way as if it were read from a file. • Compare with Python's scriptless evaluation: **python -c "print(2 ** 32)"**. • Compare with directly executing a series of commands with logic in Bash or Zsh: **for dir in {tests,api,components}; do cp ./$dir/readme.md ./docs/$dir.md; done**
Related UI	**See Sketch 1.H.**

Number	Use Case 1.I
Title	**User Defined Scopes**
Preparer	Eduardo Valdivia
Actor/User	Scripter

User Story	As a scripter I sometimes want control of the scope of some variables. I do not want my "Index" variable to be reused by another process, for instance.
Course of Events	1. Create a Commander script. 2. Use symbol **{** to start the scope 3. Create variables 4. End scope with **}** (Variables defined in scope will be lost)
Exceptions/Alternates	Function scopes, main scope.
Related UI	**See Sketch 1.I**

Number	**Use Case 1.J**
Title	**Types and Operations**
Preparer	Eduardo Valdivia
Actor/User	Commander Scripter
User Story	As a Commander Scripter, I want my variables to have a type so I can perform certain operations with them.
Course of Events	1. Create a Commander script with some type specific operations, such as substring() for strings, or length() for arrays/tuples. 2. Run the script.
Exceptions/Alternates	None
Related UI	**See Sketch 1.J.**

Number	Use Case 1.K.
Title	**Script Syntax**
Preparer	Jaden Gill
Actor/User	Novice Commander Programmer
User Story	As a novice to Commander, I would like Commander to have familiar and easy to learn syntax so I don't have to spend a lot of time learning it.
Course of Events	1. Look at documentation on language to learn basic syntax, realizing it looks similar to languages like Javascript and C. 2. Use Commander to create a script based on a previous script written in Bash, realizing it is simpler to write the same code in Commander. 3. Run script in Commander interpreter.
Exceptions/Alternates	• None
Related UI	**See Sketch 1.K.**

Number	Use Case 1.L.
Title	**Multi-Platform Interaction**
Preparer	Jaden Gill
Actor/User	Programmer / user with multiple operating systems.
User Story	As a programmer that utilizes multiple operating systems, I would like to be able to run the scripts I write in multiple operating systems so that I don't have to rewrite my scripts to run in other operating systems.

Course of Events	1. Create a script file and distribute it to multiple machines. 2. For each machine, run the script and observe results. 3. The results will be similar or identical in most cases.
Exceptions/Alternates	• The script calls a method specific to an operating system - see use case TODO • The script yields similar results, but not identical - see use case TODO
Related UI	**See Sketch 1.L.**

Number	**Use Case 2.A.**
Title	**Documentation and Examples**
Preparer	Cayden Lund
Actor/User	Language Learner
User Story	As someone unfamiliar with the Commander language, I want to learn the language and toolkit by reading the documentation and experimenting with provided examples.
Course of Events	1. Go through the documentation on the project's website (i.e., the GitLab repository). 2. Run some provided examples. 3. Adjust the examples in various ways, and observe how the adjustments affect the behavior of the examples. 4. Write new scripts based on the examples that do something useful and relevant to the user.
Exceptions/Alternates	None
Related UI	**See Sketch 2.A.**

Number	**Use Case 2.B.**

Title	**Multi-system Administration**
Preparer	Cayden Lund
Actor/User	Administrator of Multiple Systems
User Story	As the administrator of many computers (e.g., a computer lab or a workplace), I want to be able to write a script that will work on all of the managed computers. I may need to install software packages on systems that don't have that package, check that all firewalls are in working order, or run other logic as needed.
Course of Events	1. Install the Commander executable on all managed computers. 2. Write a script that will run checks, conditionally execute certain commands, and report to a central server. 3. Copy the script to each computer, and run it on each system. 4. Examine the reports and see the results.
Exceptions/Alternates	None.
Related UI	**See Sketch 2.B.**

Number	**Use Case 2.C.**
Title	**Project Bootstrapping**
Preparer	Cayden Lund
Actor/User	Toolchain Developer
User Story	As a developer of a specific toolchain, I want to write a script that will bootstrap a new project ready to integrate with this toolchain. This script will need to prompt the user with various questions and menu options.

Course of Events	1. Write a script that prompts the user with various questions. (project name, version, website, etc.). 2. The script then downloads base files from the internet. 3. Finally, the script auto-generates relevant files with the prompted information.
Exceptions/ Alternates	• Compare to **create-react-app** and **npm init**.
Related UI	**See Sketch 2.C.**

Number	**Use Case 3.A. (stretch goal)**
Title	**Syntax Highlighting**
Preparer	Preston Hales
Actor/User	Commander Scripter
User Story	As a Commander Scripter, I want to be able to write Commander scripts in the text editor or IDE that I like. I would also like the option to utilize syntax highlighting to make it easier to read and write.
Course of Events	1. Boot up text editor or IDE of choice with Commander syntax highlighting plugin. 2. Install Commander syntax highlighting plugin. 3. Write code that is more readable.
Exceptions/Alternates	• Won't work with all text editors, especially if they don't support plugins or don't have a Commander plugin available to use.
Related UI	**See Sketches 3.A.i, 3.A.ii, and 3.A.iii.**

Number	Use Case 3.B. (stretch goal)
Title	**Code Transpilation**
Preparer	Preston Hales
Actor/User	Commander Scripter
User Story	As a Commander Scripter, I want the option to be able to transpile my Commander code into Bash or Powershell so they can be directly executed in those respective shells on those respective operating systems.
Course of Events	1. Write Commander script 2. Run interpreter with **--bash** or **--powershell** flags, and the file path afterwards to signify where to save the transpiled code 3. Transfer new transpiled Bash or Powershell code to system of choice, and run them in their respective shells.
Exceptions/Alternates	• Not all shell languages will be supported.
Related UI	**See Sketch 3.B.**

Number	Use Case 4.A.
Title	**Syntax Errors**
Preparer	Eduardo Valdivia
Actor/User	Commander Scripter
User Story	As a Commander Scripter I would like feedback when I accidentally write syntax errors.

Course of Events	1. Create a Commander script with some syntax errors 2. Run the script 3. The Commander interpreter will throw error information.
Exceptions/ Alternates	• Getting errors/warnings while editing code (part of syntax highlighting plugins; stretch goal).
Related UI	See Case 4.A.

Number	**Use Case 4.B.**
Title	**Bad Interpreter Flag**
Preparer	Jaden Gill
Actor/User	Commander Script Runner
User Story	As a Commander script runner, I want it so that when I run the Commander interpreter and pass in an unknown flag such as --hello that it will display a helpful error message explaining that the flag does not exist.
Course of Events	1. Create Commander script. 2. Run Commander script through interpreter with **--hello**, an unknown flag. 3. The Commander interpreter displays an error: **Unknown flag '--hello'**
Exceptions/Alternates	• The flag is valid, in which case the interpreter just runs as normal.
Related UI	See Sketch 4.B.

Number	**Use Case 4.C.**
Title	**Variable Not Initialized**

Preparer	Jaden Gill
Actor/User	Programmer / Scripter
User Story	As a programmer, I would like for a helpful error to be displayed when I make a typo, such as when referencing a variable, so that I can know how to easily fix it.
Course of Events	1. Commander begins checking the script before execution. 2. Commander attempts to find initialization of each variable; "car" does not appear to have been initialized. 3. Checking stops, Commander throws an error such as "Type error: Variable 'car' has not been initialized".
Exceptions/Alternates	● The script coincidentally has a variable named "car" which has already been initialized - See use case 1.I
Related UI	**See Sketch 4.C.**

Number	**Use Case 4.D.**
Title	**No Command Aliases Exist**
Preparer	Jaden Gill
Actor/User	Scripting user
User Story	As a scripting user, I would like an error displayed when I try to use a command, such as **ar** (Linux), that does not exist on my current operating system (Windows).

Course of Events	1. Run the relevant script in Commander. 2. Commander encounters the unfamiliar command. 3. The command is passed to a lookup table of alternatives - the command exists on Linux but no such alternative exists for Windows. 4. The command will be highlighted in the editor paired with the error message "**Command `ar` not found. No alternatives exist.**"
Exceptions/Alternates	● An alternative command exists in the lookup table - See case 1.D.
Related UI	**See Sketch 4.D.**

UI Sketches

1. Basic Functionality UI

1.A. Running Script


```
phales@LAPTOP-5NQSR1UF:~/demo$ echo 'echo "Hello World!";' > hello.cmd
phales@LAPTOP-5NQSR1UF:~/demo$ ls
commander  hello.cmd
phales@LAPTOP-5NQSR1UF:~/demo$ ./commander hello.cmd
Hello World!
```

1.B. Multiple Scripts and Code Reuse


```
phales@LAPTOP-5NQSR1UF:~/demo$ cat <<EOF > add.cmd
add(a: int, b: int): int {
return a + b;
}
EOF
phales@LAPTOP-5NQSR1UF:~/demo$ cat <<EOF > main.cmd
import add.cmd
echo f"{add(4, 2)}";
EOF
phales@LAPTOP-5NQSR1UF:~/demo$ ./commander main.cmd
6
```

1.C. User Input in Commander

```
phales@LAPTOP-5NQSR1UF:~/demo/downloads$ ls
directory  file.txt
phales@LAPTOP-5NQSR1UF:~/demo/downloads$ cd ..
phales@LAPTOP-5NQSR1UF:~/demo$ cat <<EOF > countAndDeleteDir.cmd
dir = read("Enter directory: ");
numItemsInDir = `ls | wc -l`;
deleteItems = read(f"There are {numItemsInDir} items in the {dir} directory. Would you like to delete them (y/n)?: ") == "y";
if (deleteItems) {
rm -rf dir;
}
EOF
phales@LAPTOP-5NQSR1UF:~/demo$ ls
commander  countAndDeleteDir.cmd  downloads
phales@LAPTOP-5NQSR1UF:~/demo$ ./commander countAndDeleteDir.cmd
Enter directory: downloads
There are 2 items in the downloads directory. Would you like to delete them (y/n)?: y
phales@LAPTOP-5NQSR1UF:~/demo$ ls
commander  countAndDeleteDir.cmd
```

1.D. OS-specific Commands


```
phales@LAPTOP-5NQSR1UF:~/demo$ cat <<EOF > listDir.cmd
alias lsDir = dir -Force helloDir || ls -a helloDir;
lsDir;
EOF
phales@LAPTOP-5NQSR1UF:~/demo$ ls
commander  helloDir  listDir.cmd
phales@LAPTOP-5NQSR1UF:~/demo$ ls -a helloDir
.   ..   hello.txt  hello2.txt
phales@LAPTOP-5NQSR1UF:~/demo$ ./commander listDir.cmd
.   ..   hello.txt  hello2.txt
phales@LAPTOP-5NQSR1UF:~/demo$
```

1.E. Loops in Commander

```
phales@LAPTOP-5NQSR1UF:~/demo$ cat <<EOF > hiThree.cmd
sayHi = read("Say hi 3 times (y/n)?: ") == "y";
while (sayHi) {
for (i = 0; i < 3; i++) {
echo "Hello!\n";
}
sayHi = read("3 more times (y/n)?: ") == "y";
}
EOF
phales@LAPTOP-5NQSR1UF:~/demo$ ls
commander  hiThree.cmd
phales@LAPTOP-5NQSR1UF:~/demo$ ./commander hiThree.cmd
Say hi 3 times (y/n)?: y
Hello!
Hello!
Hello!
3 more times (y/n)?: y
Hello!
Hello!
Hello!
3 more times (y/n)?: n
phales@LAPTOP-5NQSR1UF:~/demo$
```

1.F. Functional Programming

```
phales@LAPTOP-5NQSR1UF:~/demo$ cat <<EOF > ops.cmd
a = parseInt(read("Enter first number: "));
b = parseInt(read("Enter second number: "));
performOp(a: int, b: int, op: (int, int) => int): int {
return op(a, b);
}
//Add the two numbers
add = (x, y) => x + y;
echo f"Sum: {performOp(a, b, add)}\n";
//Subtract the two numbers
echo f"Difference: {performOp(a, b, (x, y) => x - y)}\n";
EOF
phales@LAPTOP-5NQSR1UF:~/demo$ ls
commander  ops.cmd
phales@LAPTOP-5NQSR1UF:~/demo$ ./commander ops.cmd
Enter first number: 5
Enter second number: 7
Sum: 12
Difference: -2
phales@LAPTOP-5NQSR1UF:~/demo$
```

1.G. Advanced Command Execution Features

```
phales@LAPTOP-5NQSR1UF:~/demo$ cat <<EOF > sample.txt
First Line
Second Line
Third Line
Fourth Line
Fifth Line
EOF
phales@LAPTOP-5NQSR1UF:~/demo$ cat <<EOF > countLines.cmd
try {
numLinesInFile = parseInt(`cat sample.txt | wc -l`);
printStd(f"There are {numLinesInFile} lines in sample.txt");
}
catch (1) {
printErr("Command failed, received error code 1; unable to count number of lines in sample.txt");
}
EOF
phales@LAPTOP-5NQSR1UF:~/demo$ ls
commander  countLines.cmd  sample.txt
phales@LAPTOP-5NQSR1UF:~/demo$ ./commander countLines.cmd
There are 5 lines in sample.txt
phales@LAPTOP-5NQSR1UF:~/demo$
```

1.H. Scriptless Execution

```
$ commander 'for (int num = 1; num <= 1024; num *= 4)
     ./simulate f"{num}"'
Simulating 1 object...
Simulating 4 objects...
Simulating 16 objects...
Simulating 64 objects...
Simulating 256 objects...
Simulating 1024 objects...
```

1.I User Defined Scopes

```
1
2 int x = 10;
3
4 { // Start of scope here
5
6     int y = 10;
7
8     echo f"x times y is: {x * y}"
9
10 } // End of scope here
11
12 echo f"{y}" // ERROR!!
```

```
edd ~ $ ./commander scopes.cmd
Error in line 12: Unknown symbol y
edd ~ $ 
```

1.J. Types and Operations

```
phales@LAPTOP-5NQSR1UF:~/demo$ cat <<EOF > types.cmd
myString = "Hello World!";
echo f"Substring: {myString.substring(0, 5)}";
myArray = [1, 2, 3];
echo f"Length: {myArray.length()}";
EOF
phales@LAPTOP-5NQSR1UF:~/demo$ ./commander types.cmd
Substring: Hello
Length: 3
phales@LAPTOP-5NQSR1UF:~/demo$
```

1.K. Commander Syntax

```
File Edit Options Buffers Tools Bat Help
sayHi = read("Say hi (y/n)?: ")
if (sayHi == "y") {
    echo "Hi!"
}
```

```
File Edit Options Buffers Tools
#!/bin/bash
read -p "Say hi (y/n)?: " sayHi
if [[ "$sayHi" == "y" ]]; then
    echo "Hi!"
fi
```

1.L. Multi-Platform Interaction

2. Documentation and External Functionality UI

2.A. Documentation and Examples

Functions

Grammar

```
stmt  : <variable> ( <binding> , ... ) : <type> {
            <stmt>
            ...
        }
      | <variable> ( <binding> , ... ) {
            <stmt>
            ...
        }
```

Note that the grammar implies you may define functions inside of functions. Additionally, functions may be recursive (see examples below).

Examples

```
hello() {
    echo "Hello World!";
}
```

```
goodbye(): void {
    echo "Goodbye!";
}
```

```
add(a, b) {
    echo f"Performing {a} + {b} ...";
    return a + b;
}
```

```
factorial(n: int): int {
    return n == 1 ? 1 : factorial(n - 1) * n;
}
```

2.B. Multi-system Administration

```
$ for computer in server-{0..6}.lab.domain;
do
    rsync run-updates.cmd admin@$computer:;
    ssh admin@$computer "commander run-updates.cmd";
done

Finished updates on server-0.lab.domain
Finished updates on server-1.lab.domain
Finished updates on server-2.lab.domain
Finished updates on server-3.lab.domain
Finished updates on server-4.lab.domain
Finished updates on server-5.lab.domain
Finished updates on server-6.lab.domain
```

2.C. Project Bootstrapping

3. Stretch Goals UI

3.A.i Syntax Highlighting I

3.A.ii Syntax Highlighting II

3.A.iii Syntax Highlighting III

3.B. Code Transpilation

```
phales@LAPTOP-5NQSR1UF:~/demo$ cat <<EOF > hiThree.cmd
sayHi = read("Say hi 3 times (y/n)?: ") == "y";
while (sayHi) {
for (i = 0; i < 3; i++) {
echo "Hello!\n";
}
sayHi = read("3 more times (y/n)?: ") == "y";
}
EOF
phales@LAPTOP-5NQSR1UF:~/demo$ ./commander --bash out.sh hiThree.cmd
Bash script has been saved to 'out.sh'
phales@LAPTOP-5NQSR1UF:~/demo$ ls
commander  hiThree.cmd  out.sh
phales@LAPTOP-5NQSR1UF:~/demo$ cat out.sh
#!/bin/bash

read -p "Say hi 3 times (y/n)?: " userInput
while [ "$userInput" = "y" ]; do
    for ((i = 0; i < 3; i++)); do
        echo "Hello!"
    done
    read -p "3 more times (y/n)?: " userInput
done
phales@LAPTOP-5NQSR1UF:~/demo$ chmod +x out.sh
phales@LAPTOP-5NQSR1UF:~/demo$ ./out.sh
Say hi 3 times (y/n)?: y
Hello!
Hello!
Hello!
3 more times (y/n)?: y
Hello!
Hello!
Hello!
3 more times (y/n)?: n
phales@LAPTOP-5NQSR1UF:~/demo$
```

4. Error Handling and Edge Cases UI

4.A. Syntax Errors

```
phales@LAPTOP-5NQSR1UF:~/demo$ cat <<EOF > add.cmd
add(a, b {
return a + b;
}
echo f"Sum: {add(4, 6)}\n";
EOF
phales@LAPTOP-5NQSR1UF:~/demo$ ls
add.cmd   commander
phales@LAPTOP-5NQSR1UF:~/demo$ ./commander add.cmd
Syntax error: expected ')' at line 1 column 9
phales@LAPTOP-5NQSR1UF:~/demo$
```

4.B. Bad Interpreter Flag

```
phales@LAPTOP-5NQSR1UF:~/demo$ cat <<EOF > script.cmd
echo "Hello World!";
EOF
phales@LAPTOP-5NQSR1UF:~/demo$ ls
commander   script.cmd
phales@LAPTOP-5NQSR1UF:~/demo$ ./commander --hello script.cmd
Error: Unknown flag '--hello'
phales@LAPTOP-5NQSR1UF:~/demo$
```

4.C. Variable Not Initialized

```
phales@LAPTOP-5NQSR1UF:~/demo$ cat <<EOF > script.cmd
cat = "Cat";
dog = "Dog";
echo f"I have a {dog} and a {car}";
EOF
phales@LAPTOP-5NQSR1UF:~/demo$ ./commander script.cmd
Type Error: Variable 'car' has not been initialized.
phales@LAPTOP-5NQSR1UF:~/demo$
```

4.D. No Command Aliases Exist

```
$ commander script.cmd
Error at 4:1: Command `ar` not found.
              No alternatives exist.

 4:   ar cr libfantastic.a api.o backend.o middleware.o
      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```