If you have not read the how to use documentation in this folder yet, you should do that first.

**This directory contains**

**commands.l** – defines the token types that are lexed and returned to the parser from the give rules. Some of these token types may have a token value associated with it. To add a token, define the regex, and in curly brackets, set yylval to the token if necessary and return the token type. Make sure the token type is declared in parser.y.

**parser.y** – declares the token types and defines the grammar rules. To add a command, add an entry under the necessary command category. You have to specify the token types in the order that comprise the command, and how the struct is populated based on the tokens. Don’t forget to add the new command code in parser.h. The struct is automatically returned from the block. See other examples for more details.

**parser.h** – defines the command codes that each command parses to. Also defines some error codes

**parser\_types.h** – defines codes for token values and the parsed command structure

**command\_caller.c** – has the one function that is called by menu.c to switch on each command and call the right backend function with the command’s arguments.

**command\_caller.h** – header file for command\_caller.c. Specifies some constants

**smoke\_test.c** – a c program that tests all the commands for their common usages and common failures. If a command has been changed or modified, make some tests to reflect that. This is made to be run after changes have been made to the commands of the parser.

**Makefile** – the make recipe for testing and building to an executable using cmake. To make the smoke test, uncomment the smoke\_test executable output lines if necessary and *make*.

**make.ps1** – a PowerShell script to translate the commands.l and parser.y to .c and .h files

**README.md** – Some basic background and file references for the project

**.gitignore** – a list of files for git to not track

**Running the smoke test**

You should run the smoke test with no arguments. It will then output the debug information of each command parsed. If there is a seg fault, that probably is not good and means there was an issue parsing. Look at the command being parsed in the debug statements to fix the issue.

Whether or not things were successful, the output file (currently called new\_commands.txt) should have each command tested with its command code and arguments.

At the end of a successful run, the last line of the output file should contain a count of the failed assertions. If an assertion failed, there will be the text “—assertion failed” in the command that failed. You can control-f that text and it will jump to that spot.