

CS3500 - Software Engineering

Design Documentation

Version 2

Calculator System Comprising of

1. Tokenizer
 2. infixtopostfix Converter
 3. Code Generator
 4. Interpreter/ Virtual Machine
-

Colin Kelleher - 117303363

Liam de la Cour - 117317853

Karol Przestrzelski - 117360873

Jonathan Hanley - 1174743096

Table of Contents

An Overview of the Whole System.....	Page 3
Set of Detailed Requirements.....	Page 7
Interface's Description.....	Page 9
Test Suite	Page 10
Coding Standards	Page 12
Design Diagrams.....	Page 15
Instructions on Use.....	Page 18
Additional Notes.....	Page 21

An Overview of the Whole System

1.Tokenizer

The purpose of the tokenizer is to convert the program's input into separate “tokens”. For example, 42 is a token, + is a token. The tokenizer has to read the input character by character, checking whether it is a digit, and continues reading the input until a non-digit character is identified. Every operator within the calculator can be expressed by a single character. The tokenizer then exports the token into a format <type> <value>. See **Table 1** for example This will be outputted to a file.

Types:

- ❑ I = Integer
- ❑ F = Float
- ❑ R = Right bracket
- ❑ L = Left bracket
- ❑ O = Operator

Sample Input	Sample Output
14 + 52	I 14 O + I 52
12.5* (12 + 5)	F 12.5 O * L (I 12 O + I 5 R)

Table 1. Tokenizer sample input & output

2. infixtopostfix Converter

The infixtopostfix converter will take the output from the tokenizer in the format specified above, and will change the order, see the example below. This will then be outputted to a file. See example in **Table 2**.

The infixtopostfix Converter will use the “shunting-yard” algorithm

- ❑ The output will be separated by spaces

Sample Input	Sample Output
I 1 O + I 2 O * I 3	1 2 3 * +

Table 2. infixtopostfix Converter sample input & output

3. Code Generator

The Code Generator reads the output of the infixtopostfix converter and generates instructions for the Virtual Machine (VM). We have defined an instruction set below-containing instructions to complete the actual operations such as addition, multiplication, division etc (**Table 3**). These instructions will be outputted to a file.

Instruction Set:

- ❑ LOADINT
- ❑ LOADFLOAT
- ❑ ADD
- ❑ SUB
- ❑ DIV
- ❑ MUL

Sample Input	Sample Output
1 2 3 * +	LOADINT 1 LOADINT 2 LOADINT 3 MUL ADD

Table 3. Code Generator sample input & output

4. Virtual Machine

The virtual machine will read in the output file from the code generator. It will execute the provided instructions to complete the calculation. The output of this calculation will then be printed.

5. Overall View

The output of the VM will be the final result of the calculation and will be printed to “*stdout*”. The instruction set in the code generator is readable by the VM. The Tokenizer, infixtopostfix converter, code generator, and virtual machine all read the input from an input file (‘.txt’ files)

See diagrams ***Figure 2*** and ***Figure 3***

Set of Detailed Requirements

The Calculator should fulfil the following requirements

1. Tokenizer Requirements :

1. Tokenizer will read a line from the input text file called "input.txt"
2. Input file will contain a single line of integers, floats, brackets and operators
3. Tokenizer will create tokens from each of the above
 - a. I = Integer
 - b. F = Float
 - c. R = Right bracket
 - d. L = Left bracket
 - e. O = Operator
4. Each token will be placed on a separate line, as key-value pairs.
5. The above will be outputted to a text file called "tokenized.txt"

2. infixtopostfix Requirements

1. infixtopostfix converter reads its input from a text file called "tokenized.txt".
2. The tokens in the input file are type, value pairs, with each pair separated by a newline.

-
3. It will output the calculation in postfix notation, as a string on a single line, with each value separated by a single space
 4. The above will be outputted to a text file called "postfixed.txt"

3. Code Generator Requirements

1. The Code Generator reads its input from a text file called "postfixed.txt", which has been written out to by infixtopostfix.
2. It will convert this postfix data into the instruction set used by the virtual machine.
 - a. LOADINT
 - b. LOADFLOAT
 - c. ADD
 - d. SUB
 - e. DIV
 - f. MUL
3. It will write out the output to a text file called "codegenerated.txt".

4. Virtual Machine Requirements

1. This reads its input from a text file called "codegenerated.txt"
2. The Virtual Machine should execute the code generated and output the result to *stdout*.

Interface's Description

Tokenizer -> Infixtopostfix

One token per line, in key, value pairs. The key tells the infixtopostfix converter what data type the value is.

For example,

- I 42 (Integer, with value 42)
- F 4.2 (Float, with value 4.2)

Infixtopostfix -> Code Converter

Input is a single line, each input separated by a single whitespace, in postfix notation.

E.g. 1 2 3 * /

Code Converter -> Virtual Machine

Input is a text file, where each line consists of a single instruction (as defined above).

E.g. LOADINT 42

Test Suite

Testing (with CTap)

A test suite is a collection of test cases that are intended to be used to test our software programs to show that we have satisfied specific behaviours.

The Test Suite will be available on GitHub along with our code, which will be used throughout the coding and design process to ensure that our code operates as it is supposed to.

The test files are listed below in **Table 4., Table 5. And Table 6.**

INDIVIDUAL FILE TESTING	Testing each individual file to ensure each function works correctly
unittest_codegenenerator.c	Testing the code generator as a whole, checking if output matches input. Testing the classification of characters (i.e. - = 5)
unittest_common.c	Testing writing an item to a file, resetting the structure and creating a new character type
unittest_infixtopostfix.c	Testing the infixToPostfix r as a whole, checking if output matches input. Also checking greater and equal precedence
unittest_stack.c	Testing if the stack is empty, along with testing the stack operations (Push, Pop, Size)

unittest_tokenizer.c	Testing the outfile files, structure, resetting structure, writing to files and converting characters to tokens
unittest_virtualmachine.c	Testing of Addition, Subtraction, Multiplication and Division. Testing the parsing of instructions

Table 4. Individual File Testing

INTEGRATION TESTING	Combines two components of the system and tests how they interact with each other
Integrationtest_tokenizer_infixtopostfix.c	Creating various inputs, running the two files, and ensuring the output is in the correct format. I.E.integrationtest_infixtopostifc_codegenerator "O +" = MUL
Integrationtest_infixtopostfix_codegenerator.c	

Table 5. Integration Testing

SYSTEM TESTING (Figure 4.)	System testing is end-to-end testing and is "black box" so only focussing on the input and output on various test cases
test_system.c	Testing the system as a while - Putting an equation in the input.txt file and ensuring the output is the correct result.

Table 6. System Testing

Coding Standards

Commenting

Block Commenting

There should be a commenting above all functions in the following format:

```
/*  
this is an example of a comment line 1  
This is an example of a comment line 2  
*/
```

Explaining what the function does.

Single Line Commenting

These types of comments should be used within functions/blocks of code if further explanation is needed / an additional line of code needs to be explained and should be in the following format:

```
// this is an example of a comment line 1  
// this is an example of a comment line 2
```

Makefiles & Include / Header Files

Include Files / Header files (.h)

A header file is used by C source code. It contains functions & variables commonly used within files in the project & allows them to be referenced by other source files when required.

Header files written & used within this project by the team are referenced in **Table 5**.

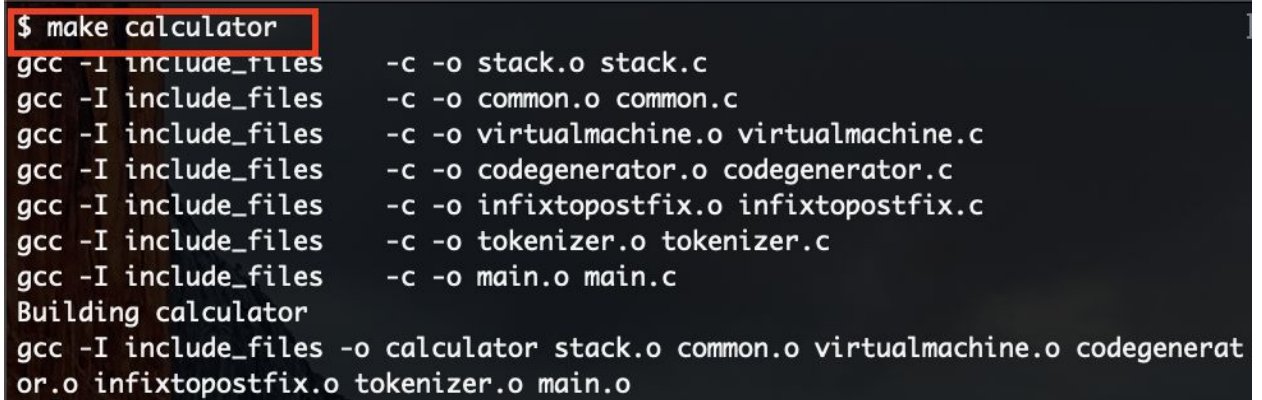
codegenerator.h
common.h
infixtopostfix.h
stack.h
tokenizer.h
virtualmachine.h

Table 5. Header files used within the project written by team

Makefile

A makefile defines a set of tasks to be executed and is a handy automation tool to run and compile programs efficiently. We have used a makefile to 'make' (compile) the calculator program so the user only has to execute one command. See **Sample 1**. We have also done the same for tests. A simple command as follows is executed within the terminal to compile the calculator.

```
$ make calculator
```

A terminal window with a dark background showing the output of the 'make calculator' command. The command '\$ make calculator' is highlighted with a red box. The output shows several gcc compilation commands for individual source files, followed by 'Building calculator' and a final gcc command linking all object files into the 'calculator' executable.

```
$ make calculator
gcc -I include_files -c -o stack.o stack.c
gcc -I include_files -c -o common.o common.c
gcc -I include_files -c -o virtualmachine.o virtualmachine.c
gcc -I include_files -c -o codegenerator.o codegenerator.c
gcc -I include_files -c -o infixtopostfix.o infixtopostfix.c
gcc -I include_files -c -o tokenizer.o tokenizer.c
gcc -I include_files -c -o main.o main.c
Building calculator
gcc -I include_files -o calculator stack.o common.o virtualmachine.o codegenerat
or.o infixtopostfix.o tokenizer.o main.o
```

Sample 1. An example of the make command in the terminal

Design Diagrams

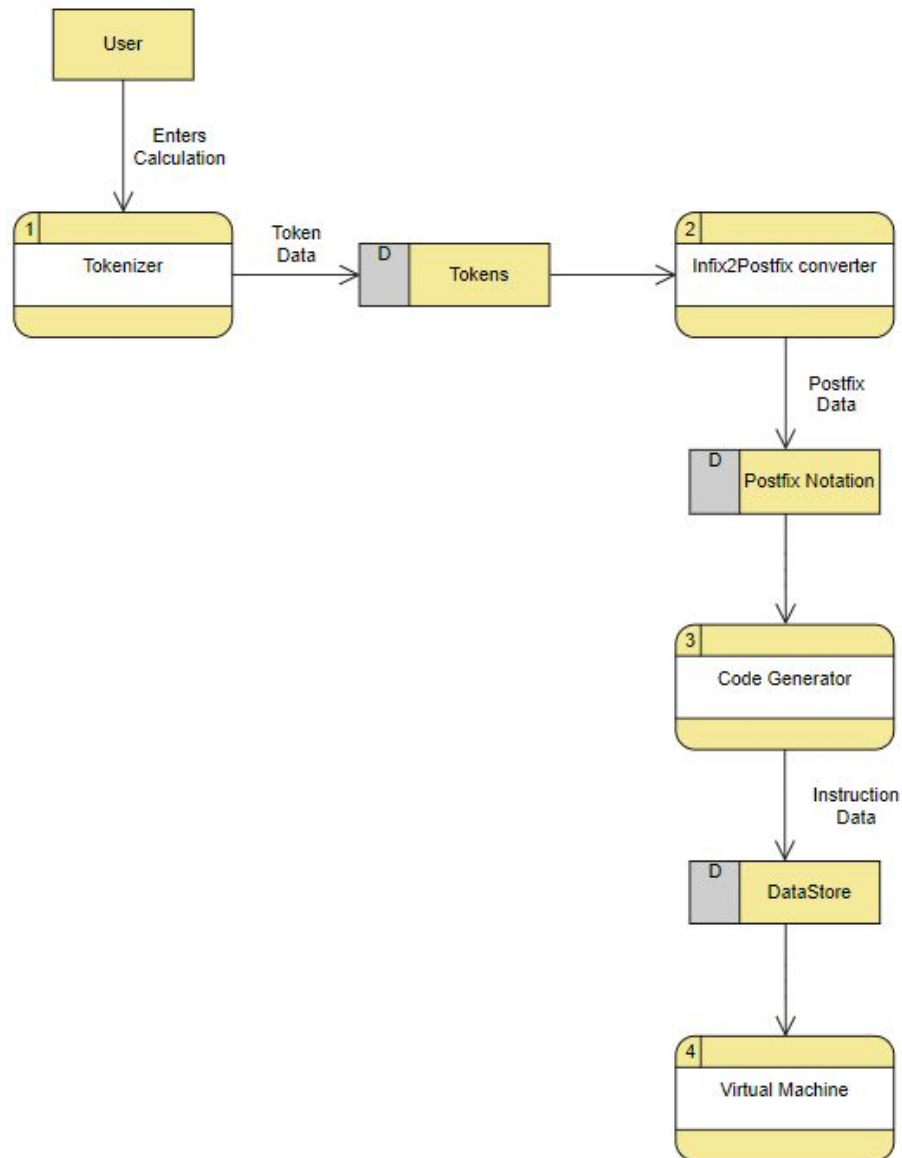


Figure 1. Data-Flow Diagram

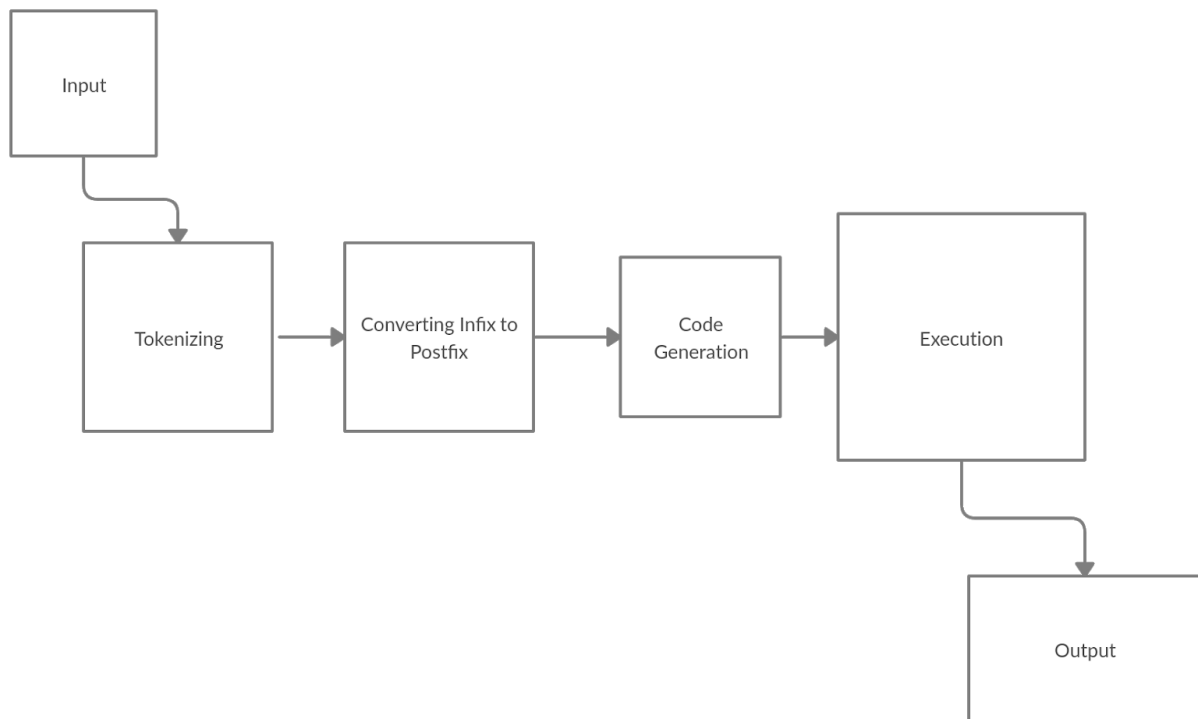


Figure 2. High-Level Architecture Diagram

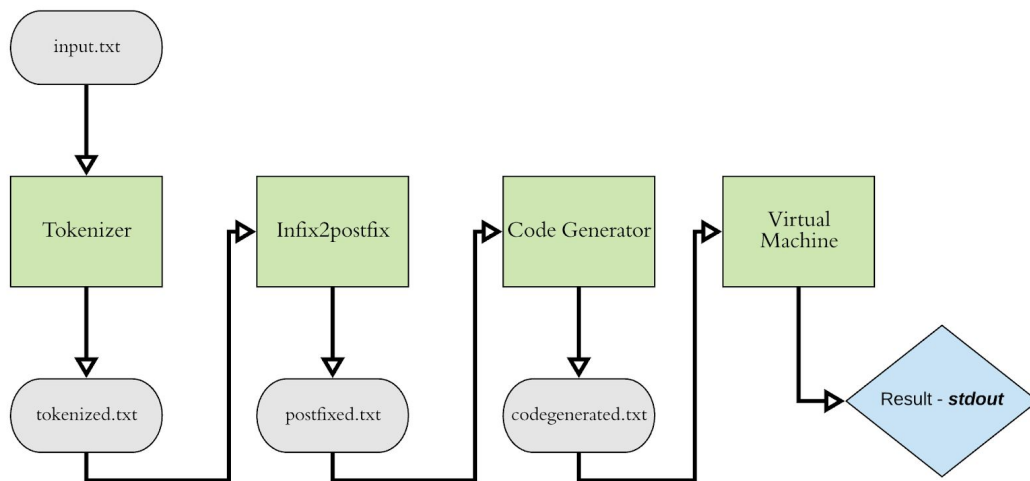


Figure 3. High-Level Architecture Diagram

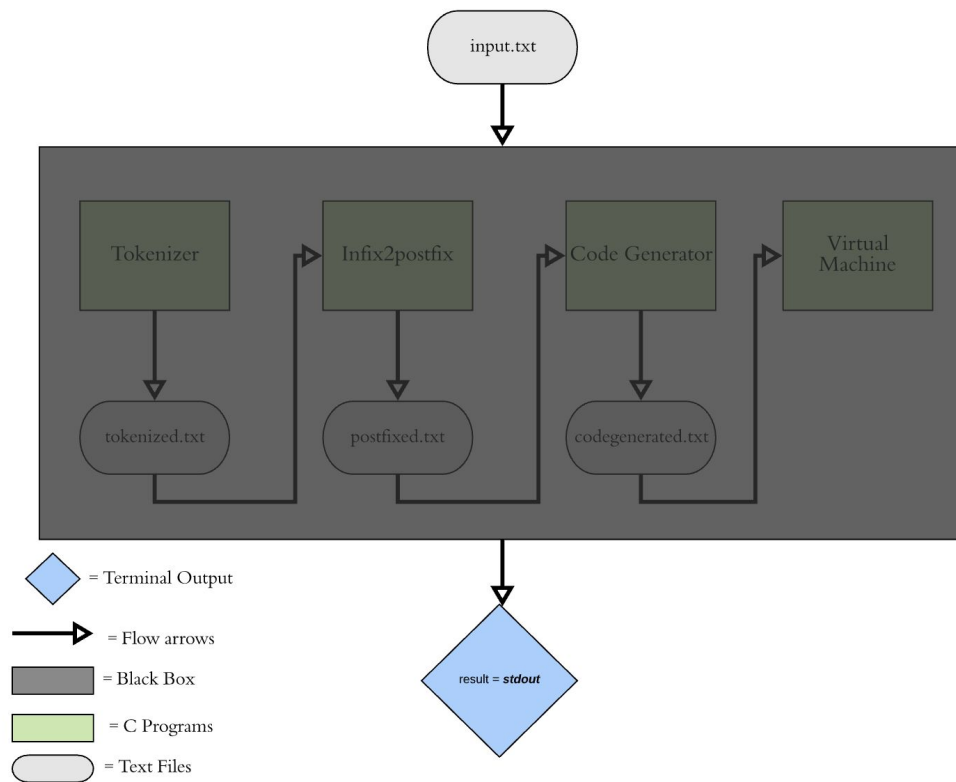


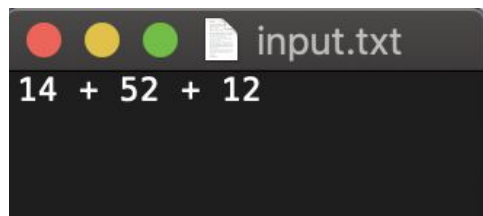
Figure 4. System Testing Diagram

Instructions on Use

Calculator

1. Text File

Insert the equation you wish to calculate in the input file **input.txt** as demonstrated below in **Sample 2** and then save.



Sample 2. input.txt

2. Invoke the MakeFile

Run the following make command in the terminal

```
$ make calculator
```

```
$ make calculator
gcc -I include_files -c -o stack.o stack.c
gcc -I include_files -c -o common.o common.c
gcc -I include_files -c -o virtualmachine.o virtualmachine.c
gcc -I include_files -c -o codegenerator.o codegenerator.c
gcc -I include_files -c -o infixtopostfix.o infixtopostfix.c
gcc -I include_files -c -o tokenizer.o tokenizer.c
gcc -I include_files -c -o main.o main.c
Building calculator
gcc -I include_files -o calculator stack.o common.o virtualmachine.o codegenerat
or.o infixtopostfix.o tokenizer.o main.o
```

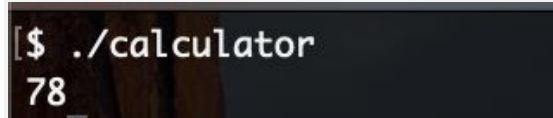
Sample 1. An example of the make command in the terminal

3. Running the Calculator

Run the following make command in the terminal:

```
$ ./calculator
```

And the result of the calculation is printed to the screen via the terminal. `14 + 52 + 12 = 78`

A terminal window with a dark background. The prompt is '\$./calculator' and the output is '78'.

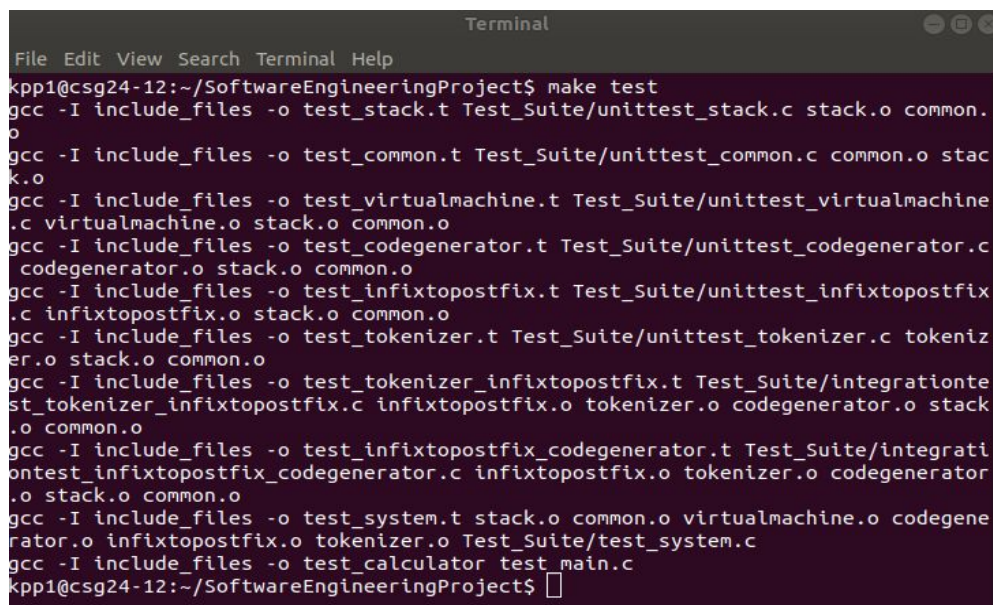
Sample 3. Running the Calculator

Tests

1. Invoke the MakeFile

To run a test, invoke the makefile in the following format:

```
$ make test / check
```

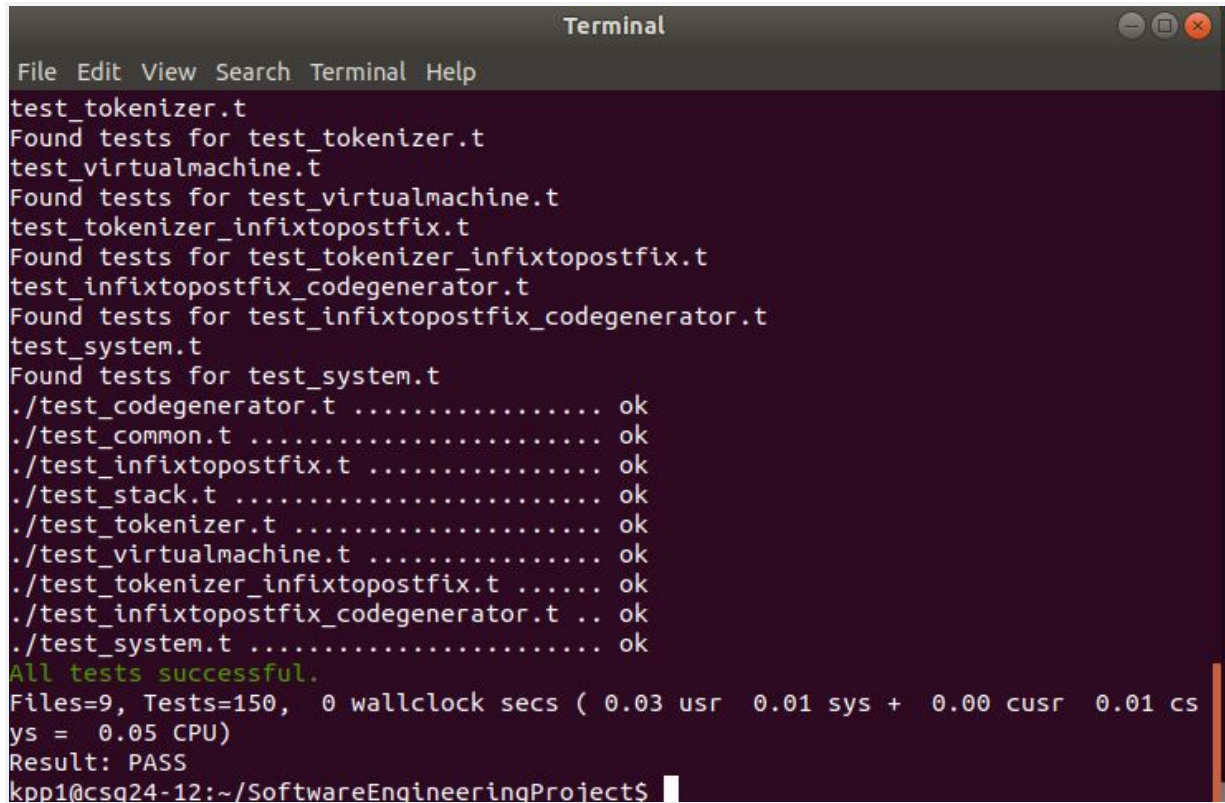
A terminal window titled 'Terminal' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'kpp1@csg24-12:~/SoftwareEngineeringProject\$'. The command 'make test' has been executed, resulting in a series of gcc compilation commands for various test files (test_stack.t, test_common.t, test_virtualmachine.t, test_codegenator.t, test_infixtopostfix.t, test_tokenizer.t, test_tokenizer_infixtopostfix.t, test_infixtopostfix_codegenator.t, test_system.t, test_calculator) and their dependencies. The prompt returns to 'kpp1@csg24-12:~/SoftwareEngineeringProject\$'.

Sample 4. An example of the make command in the terminal

2. Running the test

Run the following make command in the terminal:

```
$ ./test_calculator
```

A screenshot of a terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal output shows the execution of the test_calculator script, which runs a series of tests for different components. The tests are listed in a column, and their results are shown in a second column. All tests passed, indicated by "ok" or "successful". The output also includes summary statistics: Files=9, Tests=150, 0 wallclock secs (0.03 usr, 0.01 sys, 0.00 cusr, 0.01 cs), and Result: PASS. The prompt at the bottom shows the user is kpp1@csq24-12 in the directory ~/SoftwareEngineeringProject.

```
test_tokenizer.t
Found tests for test_tokenizer.t
test_virtualmachine.t
Found tests for test_virtualmachine.t
test_tokenizer_infixtopostfix.t
Found tests for test_tokenizer_infixtopostfix.t
test_infixtopostfix_codegenerator.t
Found tests for test_infixtopostfix_codegenerator.t
test_system.t
Found tests for test_system.t
./test_codegenator.t ..... ok
./test_common.t ..... ok
./test_infixtopostfix.t ..... ok
./test_stack.t ..... ok
./test_tokenizer.t ..... ok
./test_virtualmachine.t ..... ok
./test_tokenizer_infixtopostfix.t ..... ok
./test_infixtopostfix_codegenerator.t .. ok
./test_system.t ..... ok
All tests successful.
Files=9, Tests=150, 0 wallclock secs ( 0.03 usr  0.01 sys +  0.00 cusr  0.01 cs
ys =  0.05 CPU)
Result: PASS
kpp1@csq24-12:~/SoftwareEngineeringProject$
```

Sample 5. An example of running the tests

Additional Notes

Team Members

Colin Kelleher	117303363
Liam de la Cour	117317853
Karol Przestrzelski	117360873
Jonathan Hanley	1174743096

CS3500.2020.X 6 on Canvas

Programming Language

C Programming Language

GitHub

<https://github.com/jonathanhanley/SoftwareEngineeringProject>

Coding Requirements

All code will be commented to ensure everyone within the team can understand what is happening, and ensure maintainability