CS3500

Software Engineering Project
Lecturer: Dr. Klass-Jan Stol

Colum O'Donovan Peter O'Donoghue Sophie O'Dwyer Tadhg Kearney

Contents

1. Overview

This is the documentation for a calculator system created in a modular fashion.

The functionality of the system is made up of the following four programs:

1. Tokenizer.
2. Infix2postfix converter.
3. Code generator.
4. Interpreter/Virtual Machine.

All 4 programs must read the input from an input file. This means that each individual program needs to interpret the input. The programs will work as a system that takes in an original mathematical expression and outputs the float/integer answer or an appropriate error message. As shown in fig. 1, these 4 programs will be run centrally from a 'main' program for the purpose of this project, however each can be taken out and run separately if needed.

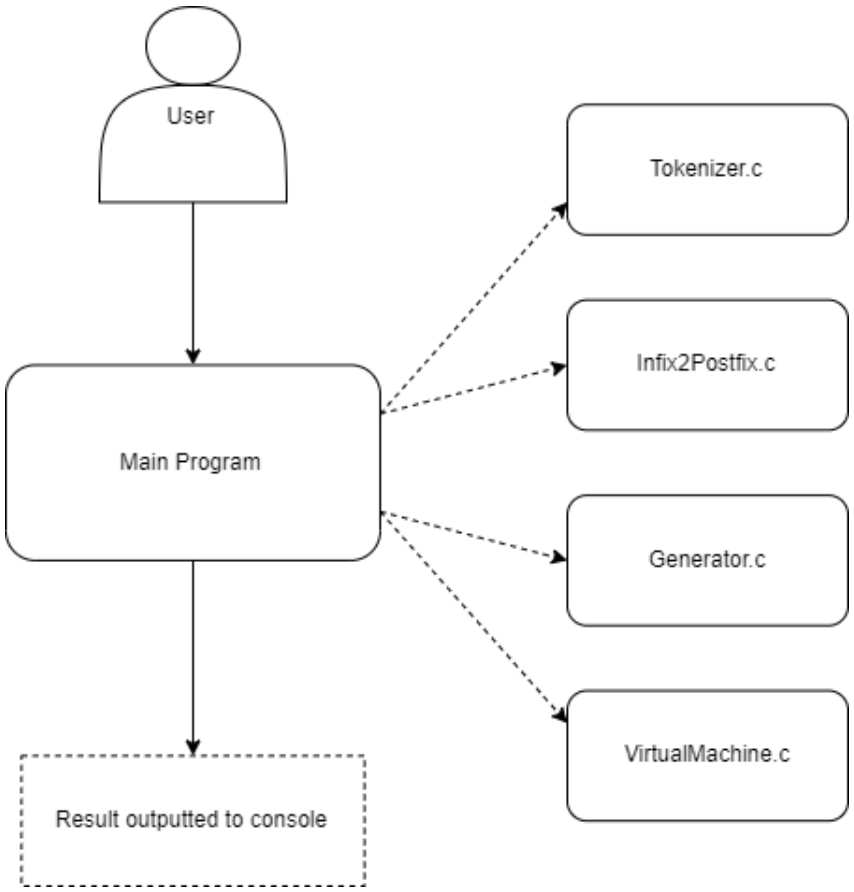

Fig. 1 – HL Architecture – Solid arrows show the system as the user sees it and the dotted arrows show the background calls they may not see.

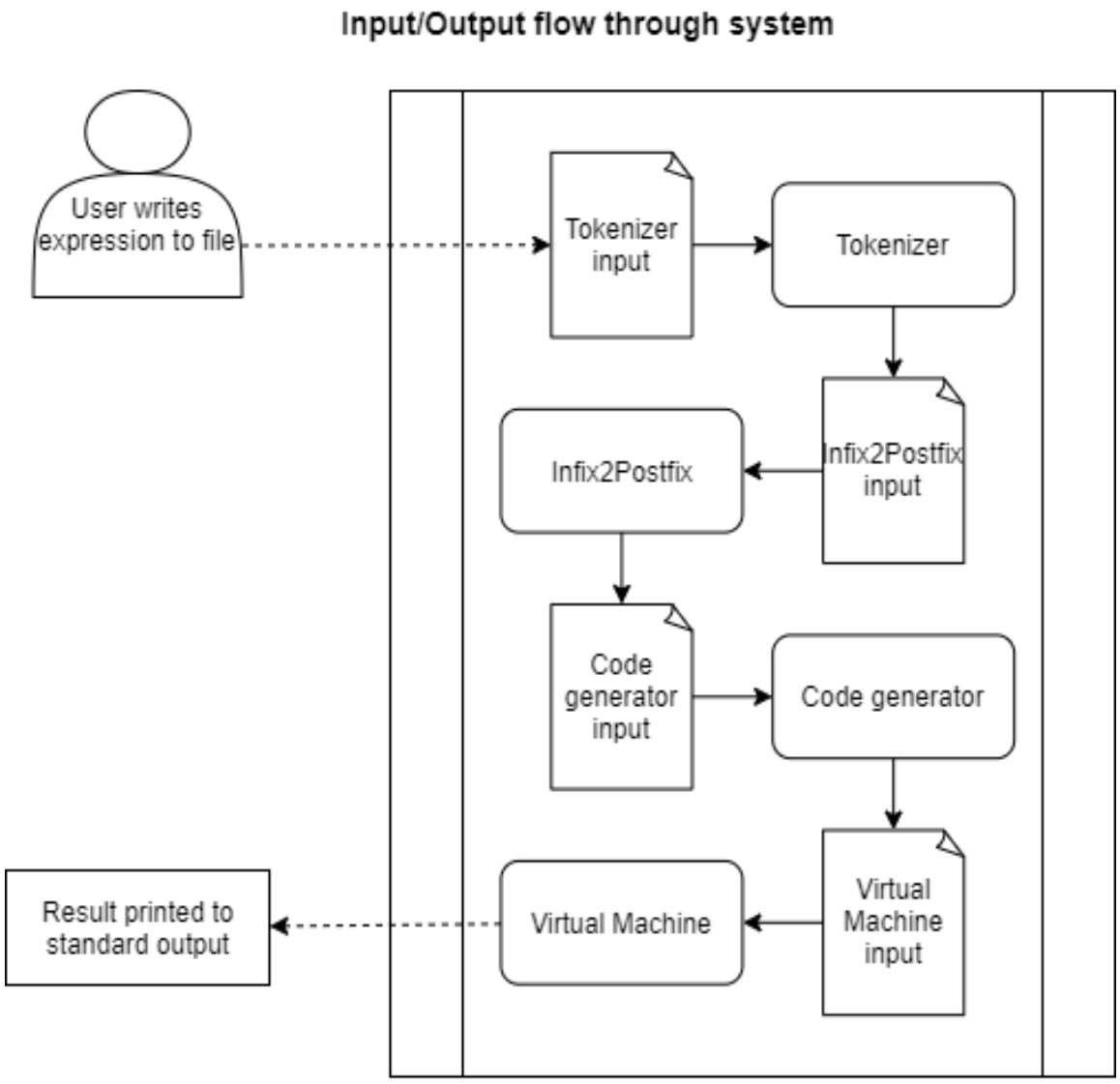

Fig. 2 – Diagram showing how user input turns into screen output. Data is processed in the programs in the rounded rectangles and flows with the arrows between files and these programs.

1.1 Tokenizer

The purpose of the tokenizer is to be able to distinguish between the various “tokens”. A token will either be a number or an operator from the inputted mathematical expression. Each of the tokens will then be outputted in order to an output file. Acceptable tokens for the tokenizer are integers, floats, +, -, /, *, ^, %, as well as (and). Fig. 3 shows how the input file is processed through the tokenizer.

Example:

Input:

5 + 6

Tokens:

“5”, “+” and “6”

1.2 Infix2postfix (I2P) converter

The purpose of the infix2postfix converter is to take the output from the tokenizer and change the order. The tokens taken as input will be in infix notation (operators between numbers) and the converter will output these tokens in postfix order notation (numbers first followed by the operators). The reordered tokens will be outputted to a file. The algorithm to convert from infix to postfix notation is illustrated in fig. 4.

Example:

Infix:

5 + 6 * 8

Postfix:

5 6 8 * +

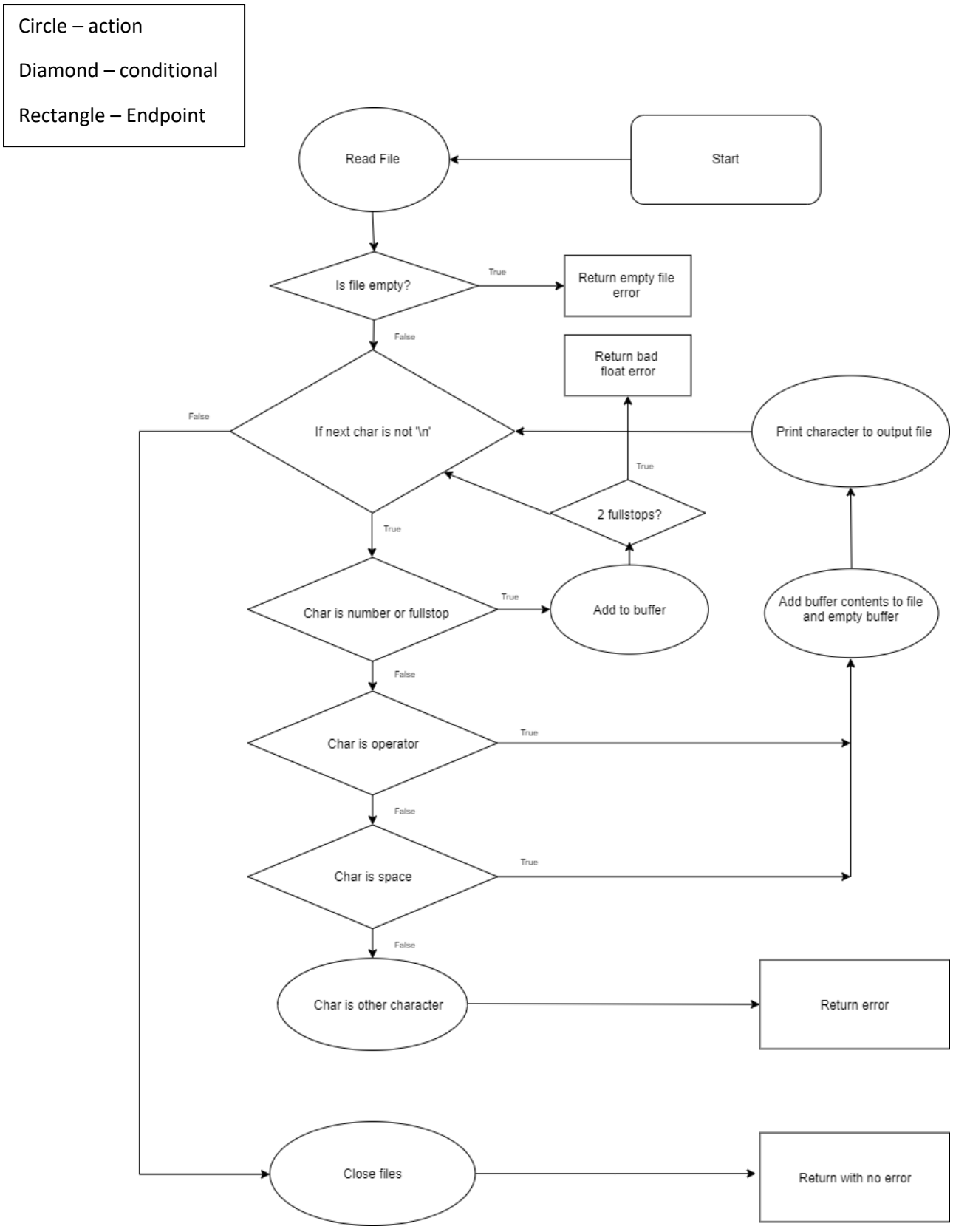

Fig. 3 – Diagram showing tokenizer's process.

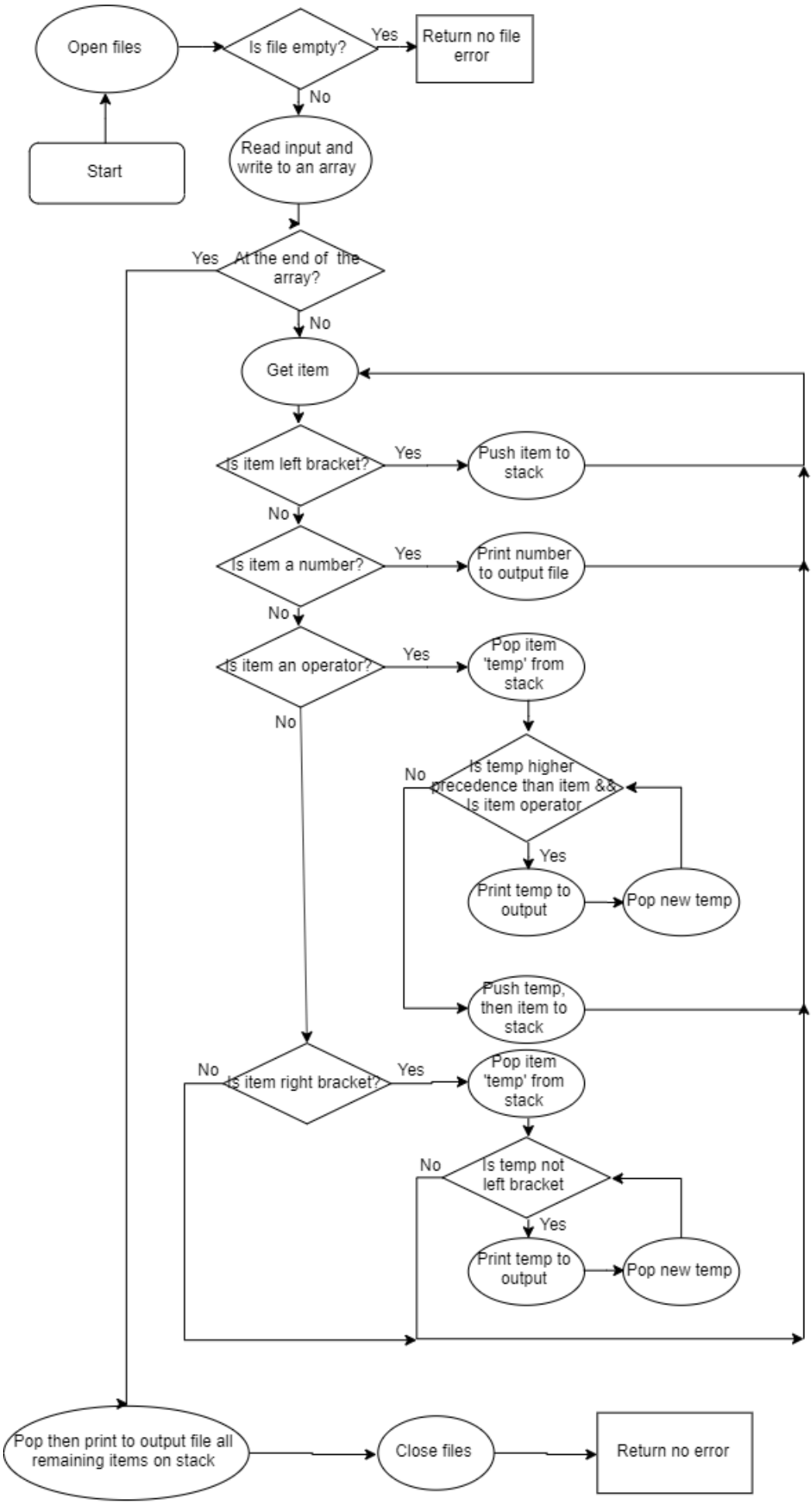

Fig. 4 – Diagram showing Infix2Postfix's process.

1.3 Code generator

The purpose of the code generator is to read the output of the I2P converter and generate instructions for a stack-based virtual machine/interpreter. Each token taken in the input will be converted to a corresponding instruction from a defined instruction set. The program will output each of these instructions. Shown in fig. 5.

Example

The input:

2 4 6 * +

Instructions:

LOADINT 2

LOADINT 4

LOADINT 6

MUL

ADD

1.4 Interpreter/Virtual Machine

The Virtual Machine takes in a set of pre-defined instructions as input. This is a stack-based virtual machine. As shown in fig.6, it pushes numbers to the stack and pops them off and operates on them if it hits an operator instruction. In this manner, it carries out the instructions it is given and calculates the result of the instructions. The result is then outputted to the terminal.

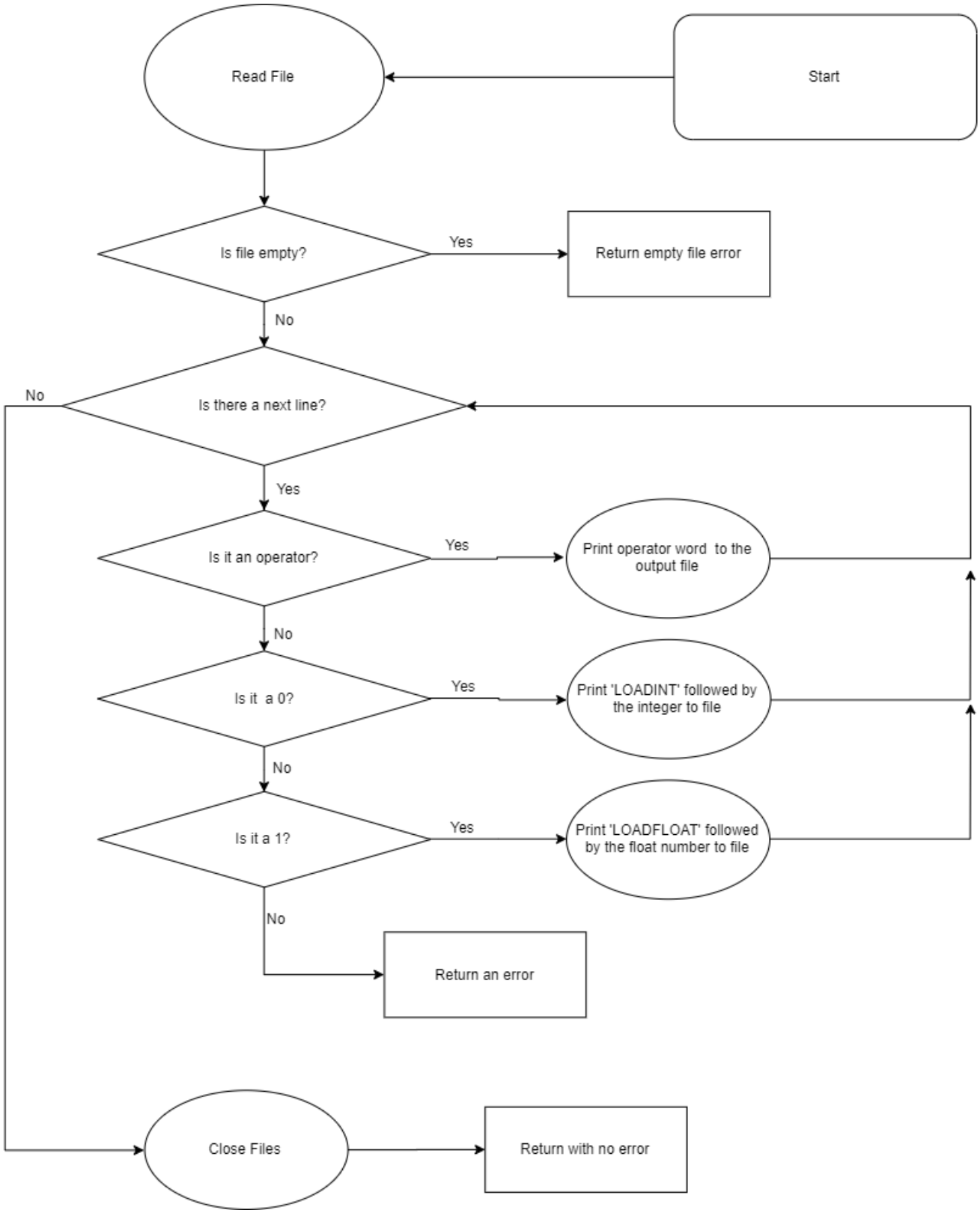

Fig. 5 – Diagram showing code generator's process.

Fig. 6 – Diagram showing virtual machine's process.

2. Software Requirements

2.1 System:

1. The system will calculate the answer to a mathematical expression.
2. The system will take in an arithmetic statement in infix notation.
3. It will then convert the arithmetic statement to postfix notation.
4. It shall then compute the calculation and output the result to the screen.
5. The system shall be made of four separate programs
6. Each program shall be piped into each other.
7. The system shall be coded in C.
8. The system shall be highly reliable.

2.2 Tokenizer:

1. The tokenizer shall receive input from the user.
2. It shall then proceed to separate the input into various tokens.
3. The accepted inputs will be: numeric digits, “*”, “+”, “-”, “/”, “^”, “%”, “(“, “)” and decimal points.
4. The tokenizer will make these into tokens that are either an integer, a float or an operator.
5. The tokenizer shall output a suitable error message if an incorrect token is inputted or when/if tokens are inputted in the wrong order.
6. The tokenizer shall export the tokens to a text file.

2.3 Infix2postfix Converter:

1. The infix2postfix converter shall receive input from a text file created by the tokenizer.
2. The converter shall convert the input from infix notation to postfix notation.
3. The converter shall export the result to a text file.

2.4 Code Generator:

1. The code generator shall receive input from a text file created by the infix2postfix converter.
2. Using the input, the code generator shall create instructions.
3. These instructions are to be based off a pre-defined instruction set.
4. The code generator shall output the instructions in the form of a text file.

2.5 Virtual Machine:

1. The virtual machine shall receive input from a text file created by the code generator.
2. Upon execution, the virtual machine shall read the instructions in the text file and compute the calculation.
3. The virtual machine shall output the result to the screen at the end of execution.
4. The virtual machine shall output an error message if it is required to divide by zero.

3. Interfaces

This piece of software has a pipe-and-filter architecture where there are four separate programs that the data must flow through. This was shown in fig. 2. The interfaces between the programs are as follows:

3.1 Tokenizer → I2P Converter

- Tokenizer outputs a file of tokens for the I2P Converter to use as input.
- Each token will be on a new line in the output file.
- These tokens will either be integers, floats or operators, including +, -, *, /, ^, %, (, and).
- If a token is a number, it will be preceded by its type – 0 for an integer, 1 for a float – so the I2P Converter easily knows how to store it.
- The I2P Converter will read in each token line by line from this file.

3.2 I2P Converter → Code Generator

- I2P Converter outputs the reordered tokens to a new file.
- The tokens will be printed to the new file in one line, space separated.
- The tokens will be either integers, floats or operators and each number will be matched with its type (int or float).
- The Code Generator will take this file as input and read in each token line by line.

3.3 Code Generator → Virtual Machine

- The Code Generator takes the tokens it has read in and converts them each to instructions.

- Instructions are outputted to a file one per line
- The valid instructions and their corresponding tokens are as follows:
    - LOADINT <integer> - 0 <integer>
    - LOADFLOAT <float> - 1 <float>
    - ADD - +
    - SUB - -
    - MUL - *
    - DIV - /
    - POW - ^
    - MOD - %

- The VM will read in instructions line by line and carry them out to complete the calculation.

4. Makefiles

There will be two makefiles included with this implementation. The first will compile all the necessary files for the calculator program as a whole. This will produce an executable. If this executable is run then the system will run, taking the expression in 'tokenizer_input.txt' as its input, and will output its result to the console.

The second makefile will be for testing purposes. When this is run it will test all 4 programs and output the result of the tests to the console.

1st command: make

2nd command: make test