CSC615M MP4: Combination of Turing Machines

Ryan Austin Fernandez
CS-ST Student
2401 Taft Avenue,
Manila, Philippines 1004
(+63)917-621-6469
ryan_fernandez@dlsu.edu.ph

ABSTRACT

Turing Machines are involved in the Church-Turing thesis, which states that any computation that can be done can be modelled using a Turing Machine. To show this, complex computations are performed such that each computation is built up from simpler Turing Machines. This project aims to show that this is possible, in the hopes that appreciation will be gained for this model of computation. Using predefined Turing Machines, the system was designed with each Turing machine as a command in the Command Design Pattern and the model was implemented in Python. The testing was successful, with the model being able to do simple computations such as GCF and square root.

I. INTRODUCTION

Turing Machines provide a generalized model for computation. In the Church-Turing thesis, it is stated that any computation that can be regarded as possible to carry out can be performed by a Turing machine with suitable instructions. Therefore, any possible computation can be modelled as an algorithm to be carried out by a Turing Machine (Denning, P.J., Dennis, J.B., & Qualitz, J.E., 1978).

This raises the question of whether a low level simulation of Turing Machines is possible, such that with elementary Turing Machines, simple computations such as increment or decrement can be built upon.

This project's goal is to simulate more complex Turing Machines with simple Turing Machines to be described in the next section. The project's scope is also further described in the next section.

This project's significance is to see the importance of the Church-Turing thesis, and how it can be used to model basic computation, such as greatest common divisor or square root.

II. DESIGN

The elementary Turing Machines described in the project operate on the general notion that Turing Machines operate on a tape with #-delimited natural numbers in unary notation. This means an input of x1,x2 would look like $\#1^{x1}\#1^{x2}\#$.

The elementary Turing machines implemented are as follows:

shL k / shR k – shift left/right moves the tape head k numbers to the left or right

copy k – copies the kth number to the left of the head and places it to the right of the initial head position. The head ends up on the right of the new copy.

const k – prints the constant k in unary notation to the right of the tape head. Tape head remains to the left of the constant.

move m,n – deletes m numbers to the left of the initial tape head and moves n numbers from the right of the initial tape head location to the left. Tape head ends at the left of the n numbers.

pushL – equivalent to move 1, 1

inc – Current number is copied and incremented. Tape head is to the left of the incremented copy.

dec - Current number is copied and decremented. Tape head is to the left of the decremented copy.

add – adds the two numbers to the right of the tape head and leaves the result there.

mult - multiplies the two numbers to the right of the tape head and leaves the result there.

monus - subtracts the two numbers to the right of the tape head and leaves the result there. If the first number is less than the second number, a 0 is set as the result.

swap – swaps the two numbers to the right of the initial tape head.

gotoEQ – checks if the first number to the right of the tape head is equal to the second number

gotoNE – checks if the first number to the right of the tape head is not equal to the second number

gotoGE – checks if the first number to the right of the tape head is greater than or equal to the second number

gotoGT – checks if the first number to the right of the tape head is greater than the second number

gotoLE – checks if the first number to the right of the tape head is less than or equal to the second number

gotoLT – checks if the first number to the right of the tape head is less than the second number

There is also a goto option which does an unconditional jump to an instruction.

Combined Turing Machines are a set of instructions of the form

Instruction no.,label,[param1,…],[next instruction]

For goto type instructions, it is implied that if the condition is evaluated to false, the program executes the next statement in sequence i.e. no jump.

All these Turing Machines were abstracted in the Command Design Pattern, where their execution is abstracted as a do() method in the Command interface.

The do() method takes in the list of integers on the tape, the current tape head location, and then outputs the new tape, the new tape head location, and the new instruction counter. HALT returns NIL instead of the usual 3-tuple.

A Turing Machine, then, is just composed of a list of commands to be executed as specified.

Figure 2.1 shows the UML diagram of the system.

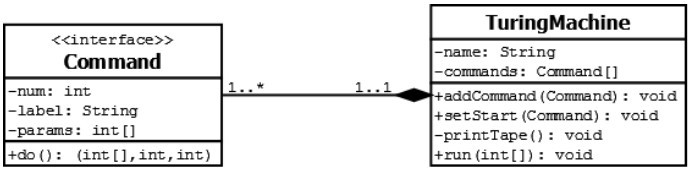

Figure 2.1. – UML Diagram of the System

The algorithm now for running the Turing Machine is in Table 2.1.

Table 2.1. – Algorithm for running the Turing Machine

```
def run(input)
    input – integers on the tape
    curr = start #start instruction
    index = 0

    list = input

    print("START")
    printTape()

    while curr is not NIL:
        ret = curr.do(list,index)
        print("After " + curr)
        if ret is None:
            curr = None
        else:
            list = ret[0]
            index = ret[1]
            curr = commands[ret[2]]
        printTape()
```

III. IMPLEMENTATION

The system was implemented in Python. The project had four modules: TuringMachine.py for modelling the actual machine, Command.py to model the states, FileReader.py to read the file, and Main4.py to act as the driver.

The TuringMachine class used a dictionary to map instruction ids to commands. The commands simply extended the Command class that acts as an interface.

IV. TESTING

The first set of programs tested the basic Turing machines first i.e. no goto yet.

The first is a manual increment program, which looks like

1,shR,1
2,copy,1
3,const,1
4,shL,1
5,add
6,HALT

Input is 0, 1, and 2. Expected output is 1,2 and 3. Figure 4.1. shows the output.

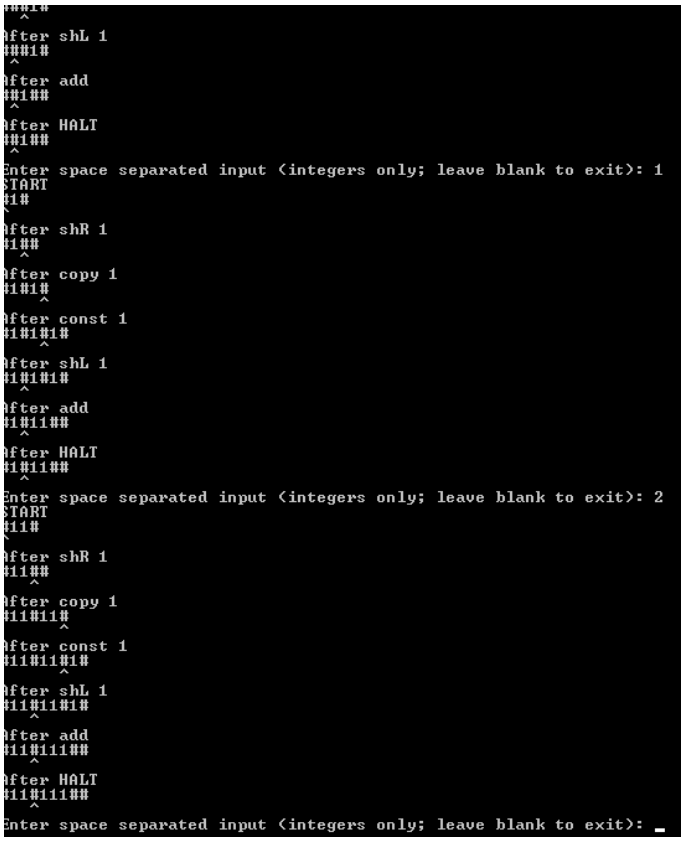

Figure 4.1. – Output for Increment.

Taking the output as the number to the right of the head, the output is, in fact, 1, 2, and 3.

Decrement was also tested, replacing the add with a monus. Inputting 2 and 3 should produce an output of 1 and 2. Figure 4.2. shows the actual output.

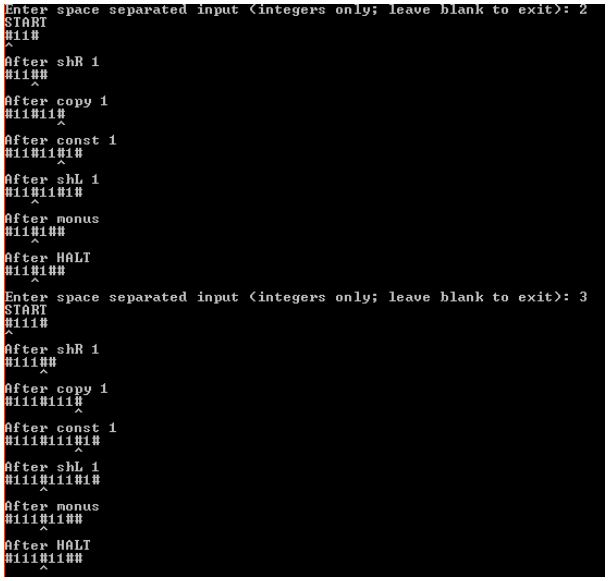

Figure 4.2. – Output for Decrement

To test the remaining instructions, a program called square was produced, which produces the square of a number. The code is:

1,shR,1

2,copy,1
3,shL,2
4,mult
5,HALT

Feeding it an input of 1, 2, and 3, expected output is 1, 4, and 9.

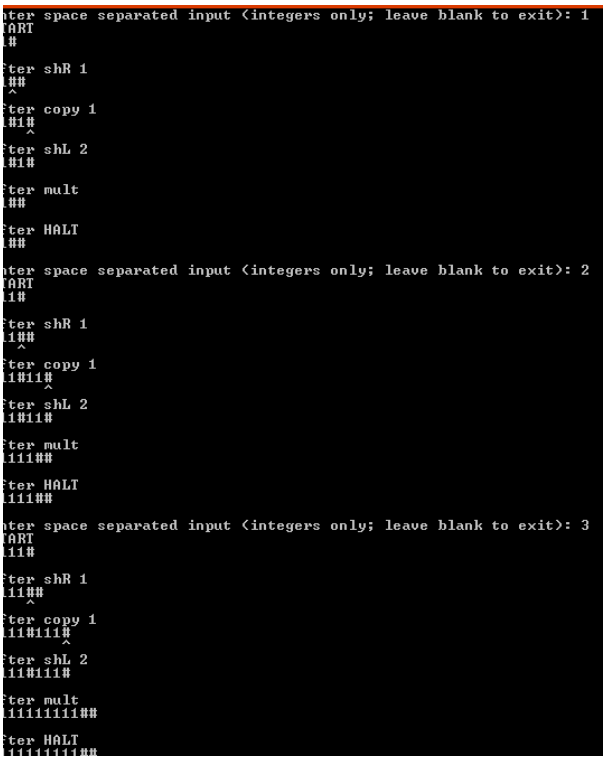

Figure 4.3. – Output for Square

Figure 4.3. shows the output, which is in fact 1, 4, and 9.

Another program called incswap, tests the pushL, move, and swap modules. It essentially produces a 2-tuple with the increment as the first number and the decrement as a second number. The code is:
1,inc
2,shL,1
3,swap
4,shR,1
5,dec
6,move,1,1
7,shL,1
8,HALT

Inputting 2 and 3 should produce (3,1) and (4,2). Figure 4.4. shows the output.

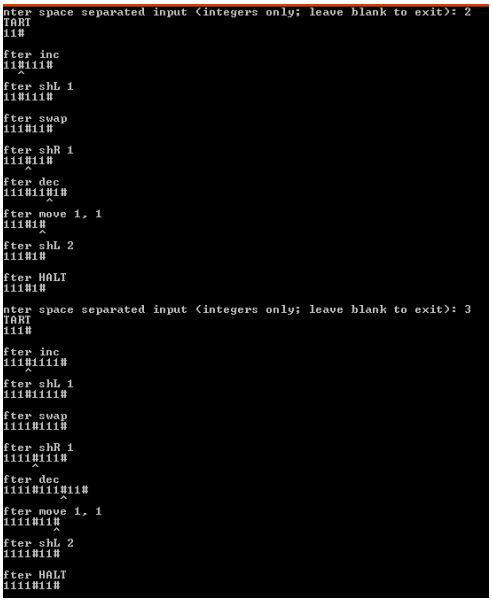

Figure 4.4. – Output for incswap

The actual output matches the expected output.

To test the goto functions, a greatest common factor program was coded. The code is:

1,shR,2
2,copy,2
3,copy,2
4,shL,2
5,gotoNE,10
6,shL,1
7,pushL
8,shL,1
9,HALT
10,copy,2
11,copy,2
12,shL,2
13,gotoGT,22
14,copy,2
15,copy,2
16,copy,2
17,shL,2
18,monus
19,shL,1
20,move,2,2
21,goto,1
22,copy,1
23,copy,3
24,copy,2
25,shL,2
26,monus
27,shL,1
28,move,2,2
29,goto,1

Inputting 3 and 2, 4 and 2, should produce 1 and 2 respectively.

Figure 4.5 shows the last few steps and the final output for 3 and 2 as input. Figure 4.6 shows the last few steps for 4 and 2.

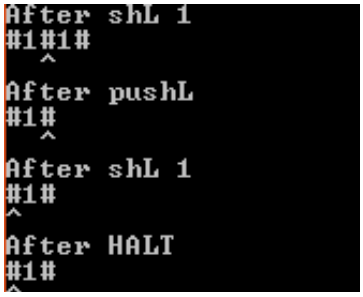

Figure 4.5.- Last few steps for GCF of 3 and 2

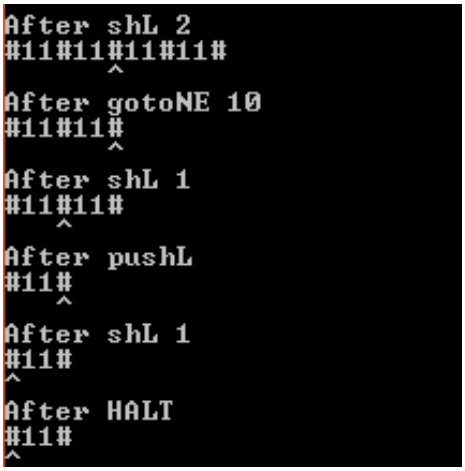

Figure 4.6 – Last few steps for GCF of 4 and 2

The actual output matches the expected output.

Another test was using the square root function. Input of 1 and 16 should produce 1 and 4 respectively.

Figure 4.7. shows the last lines of program output for 1 as input.

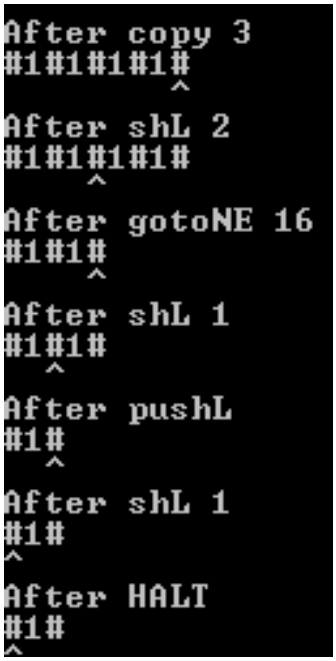

Figure 4.7. – Last lines for Square Root Program with 1 as input

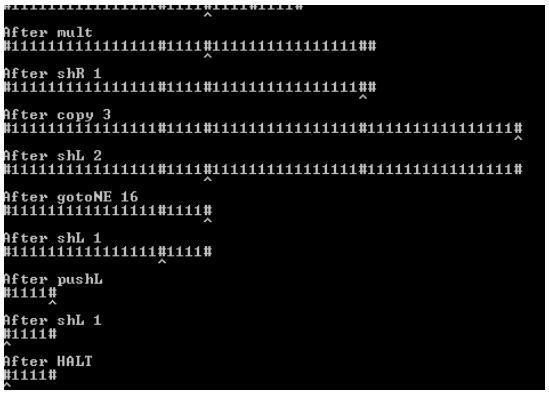

Figure 4.8. – Last lines for Square Root Program with 16 as input

The actual output matches the expected output.

Since all of the component Turing Machines have been found to be working and the system is capable of simulating complex Turing machines such as GCF and Square Root, it can be said that the system is in working condition.

V. CONCLUSION

In conclusion, the project was successfully implemented. The program can successfully simulate a complex Turing Machine from component Turing Machines.

This project is a guide to understanding Turing's thesis in that, if a function is computable, it should be possible to design a Turing Machine for it.

APPENDIX A – SELF-ASSESSMENT RUBRIC

Criterion	Grade
I/O Modules	30
Core Process	30
Quality of Testing	19
Documentation	20
TOTAL	99

REFERENCES

[1] Denning, P.J., Dennis, J.B., & Qualitz, J.E. (1978). *Machines, languages, and computation*. New Jersey: Prentice Hall.