CSC615M Machine Project Specifications
Abstract Machine Interpreter

1 Project Description

This project involves you creating an interpreter for various abstract models of computation. These models involve different types of memory storage, memory access, and input-output access.

2 Input Specifications

Input consists of two primary sections when defining the abstract machine: the DATA section and the LOGIC section.

2.1 Auxiliary Memory

In the DATA section, any auxiliary memory is declared. The section begins with a line containing `.DATA` The following are the types of memory that should be supported.

- `STACK <stack_name>`
  This declares a stack with the given identifier. For example `STACK S1` declares a stack named `S1`. The stack accesses memory in a Last-In-First-Out (LIFO) order.
- `QUEUE <queue_name>`
  This declares a queue with the given identifier. For example `QUEUE Q1` declares a queue named `Q1`. The queue accesses memory in a First-In-First-Out (FIFO) order.
- `TAPE <tape_name>`
  This declares tape memory with the given identifier. For example `TAPE T1` declares a tape named `T1`. The tape can be accessed freely by scanning it left and right, ala Turing Machine. **For machines with at least one tape, the first tape declared is also designated as the input tape.** `2D_TAPE <2D_tape_name>` This declares tape memory with the given identifier. For example `2d_TAPE P1` declares a queue named `P1`. The tape can be accessed freely by scanning it left and right, ala Turing Machine, or up and down. **For machines with at least one tape or 2D tape, the first tape-type memory declared is also designated as the input tape.** If the first tape-type memory is a 2D Tape, the input is on the first/topmost row.

2.2 Logic

In the LOGIC section, each line defines a state's behavior using the following syntax:

```
<SOURCE_STATE_NAME>] COMMAND (<SYMBOL_1>,<DESTINATION_STATE_NAME_1>),
     (<SYMBOL_2>,<DESTINATION_STATE_NAME_2>),
     ...,
     (<SYMBOL_N>,<DESTINATION_STATE_NAME_N>)
```

Only the commands `LEFT` and `RIGHT` do not follow this syntax (defined below).
For example

```
A] SCAN (1,A), (2,B), (3,A)
```

Defines the behavior of state A. Upon scanning a 1, the machine transitions into state A. Upon scanning a 2, the machine transitions into state B. Upon scanning a 1, the machine transitions into state A.
The following commands are supported by the machine definition language:

- `SCAN` - this command reads one symbol from the input
- `PRINT` - this command produces one output symbol
- `SCAN RIGHT` - this command is identical to `SCAN`. Specifically, this command reads one symbol from the input to the right of the tape head. The tape head then moves to that location.
- `SCAN LEFT` - this command reads one symbol from the input to the left of the tape head. The tape head then moves to that location.
- `READ(<memory_object>)` - this command reads one symbol from a given stack or queue. For example, if a `STACK S1` was declared in the DATA section, a valid LOGIC definition is `1] READ(S1) (X,1), (Y,2)`. This means if X is "popped" from the stack, the machine stays in state 1, but if Y is popped, the machine moves to state 2.
- `WRITE(<memory_object>)` - this command reads one symbol from a given stack or queue. For example, if a `QUEUE Q1` was declared in the DATA section, a valid LOGIC definition is `1] WRITE(Q1) (X,1), (Y,2)`. This is an example of nondeterminism in the machine. If the machine enqueues X in $Q1$, it will stay in state 1, but if it nondeterministically decides to enqueue Y instead, it will move to state 2.
- `RIGHT(<tape_name>)` - this command reads one symbol on an input tape to the right of the tape head and moves to that location. It also changes the state and overwrites that symbol with a new symbol. This instruction only applies to tapes or 2D tapes. This command uses a unique syntax:

```
<SOURCE_STATE_NAME>] RIGHT(<tape_name>) (<SYMBOL_1>/<REPLACEMENT_SYMBOL_1>,<DESTINATION_STATE_NAME_1>),
     (<SYMBOL_2>/<REPLACEMENT_SYMBOL_2>,<DESTINATION_STATE_NAME_2>),
     ...,
     (<SYMBOL_N>/<REPLACEMENT_SYMBOL_N>,<DESTINATION_STATE_NAME_N>)
```

For example, `1] RIGHT(T1) (0/X,1),(1/Y,2)` means if the machine finds a 0 to the right of the tape head on $T1$, replace it with X, move the tape head to that location, and stay in state 1. If the machine instead finds a 1 to the right of the tape head, replace it with Y, move the tape head to that location, and move to state 2.

- `LEFT(<tape_name>)` - this command reads one symbol on an input tape to the left of the tape head and moves to that location. It also changes the state and overwrites that symbol with a new symbol. This instruction only applies to tapes or 2D tapes. This command uses a unique syntax:

```
<SOURCE_STATE_NAME>] LEFT(<tape_name>) (<SYMBOL_1>/<REPLACEMENT_SYMBOL_1>,<DESTINATION_STATE_NAME_1>),
    (<SYMBOL_2>/<REPLACEMENT_SYMBOL_2>,<DESTINATION_STATE_NAME_2>),
    ...,
    (<SYMBOL_N>/<REPLACEMENT_SYMBOL_N>,<DESTINATION_STATE_NAME_N>)
```

  For example, `1] LEFT(T1) (0/X,1),(1/Y,2)` means if the machine finds a 0 to the left of the tape head of $T1$, replace it with X, move the tape head to that location, and stay in state 1. If the machine instead finds a 1 to the left of the tape head, replace it with Y, move the tape head to that location, and move to state 2.

- `UP(<2D_tape_name>)` - this command reads one symbol on an input tape to the north of the tape head and moves to that location. It also changes the state and overwrites that symbol with a new symbol. This instruction only applies to 2D tapes. This command uses a unique syntax:

```
<SOURCE_STATE_NAME>] UP(<2D_tape_name>) (<SYMBOL_1>/<REPLACEMENT_SYMBOL_1>,<DESTINATION_STATE_NAME_1>),
    (<SYMBOL_2>/<REPLACEMENT_SYMBOL_2>,<DESTINATION_STATE_NAME_2>),
    ...,
    (<SYMBOL_N>/<REPLACEMENT_SYMBOL_N>,<DESTINATION_STATE_NAME_N>)
```

  For example, `1] UP(P1) (0/X,1),(1/Y,2)` means if the machine finds a 0 to the north of the tape head of $P1$, replace it with X, move the tape head to that location, and stay in state 1. If the machine instead finds a 1 to the north of the tape head, replace it with Y, move the tape head to that location, and move to state 2.

- `DOWN(<2D_tape_name>)` - this command reads one symbol on an input tape to the south of the tape head and moves to that location. It also changes the state and overwrites that symbol with a new symbol. This instruction only applies to 2D tapes. This command uses a unique syntax:

```
<SOURCE_STATE_NAME>] DOWN(<2D_tape_name>) (<SYMBOL_1>/<REPLACEMENT_SYMBOL_1>,<DESTINATION_STATE_NAME_1>),
    (<SYMBOL_2>/<REPLACEMENT_SYMBOL_2>,<DESTINATION_STATE_NAME_2>),
    ...,
    (<SYMBOL_N>/<REPLACEMENT_SYMBOL_N>,<DESTINATION_STATE_NAME_N>)
```

  For example, `1] DOWN(P1) (0/X,1),(1/Y,2)` means if the machine finds a 0 to the south of the tape head of $P1$, replace it with X, move the tape head to that location, and stay in state 1. If the machine instead finds a 1 to the south of the tape head, replace it with Y, move the tape head to that location, and move to state 2.

Note that each state's behavior is completely defined in a single line. This means only one command can be associated with any of the states.

The first line in the `.LOGIC` section defines the initial state. There are two special reserved words for state names. `accept` as a state name means that when the machine enters that state, the string is accepted, and the machine halts execution. `reject` as a state name means that when the machine enters that state, the string is rejected, and the machine halts execution.

3 Software Description

Your task is to implement an interpreter for this simple machine description language. A simple GUI is required. Please check this link for a simple example GUI (`https://morphett.info/turing/turing.html`). There must be a text field to input the machine definition and another text field to place an input string. Upon running, you should be able to show the state and values in each memory object during each step of the input. Optionally, you can ask for multiple inputs and show the outputs for each input without showing the machine state step-by-step.

There is no need to visualize the state diagram, but incentives will be given if your system can generate a **readable** state diagram.

The system can be implemented as a desktop application using C++, Java, or Python. You can also implement this as a webpage using any web development framework of your choice. Please do not create a mobile application.

4 Grading

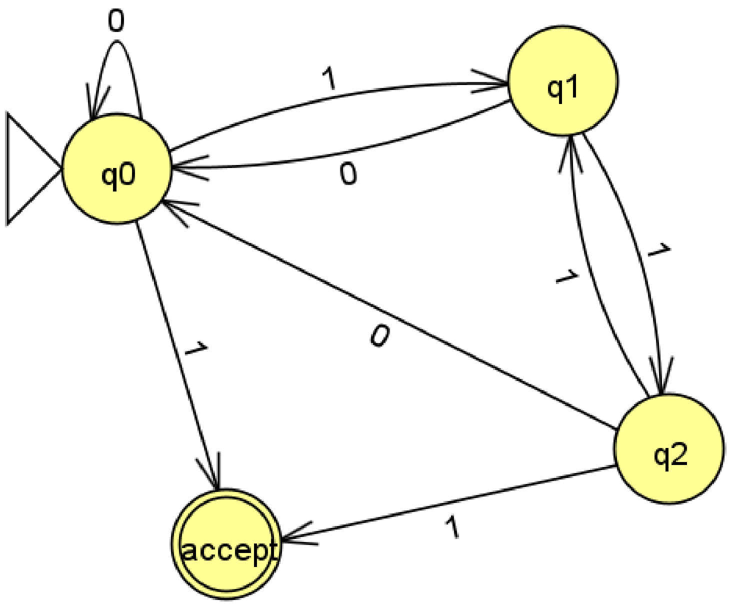
This project is worth 100 points. Correct design and implementation of the following components yield the points shown below:

Module	Points
Proper Input + Output Module For Machine Definition	12
Proper Input Module for Input String	9
Proper Output Module for Accept/Reject	3
Visualization of States and Memory in GUI	6
Stack Memory	12
Tape Memory	12
One-Way Input Instructions (SCAN)	12
Two-Way Input Instructions (SCAN LEFT/SCAN RIGHT)	12
Stack/Queue Memory Instructions (READ/WRITE)	12
1D Tape Instructions (LEFT/RIGHT)	12
Support for Nondeterministic Models	18
Output Tape Instructions (PRINT)	3
2D Tape Instructions (UP/DOWN)	6
Queue Memory	12
Bonus GUI Elements, e.g., State Diagram Generation, Showing all Nondeterministic states	12

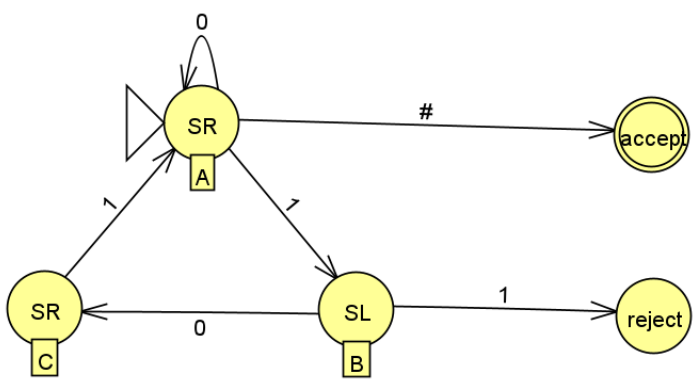
Table 1: Caption

5 Sample Inputs and Corresponding State Diagram

```
.LOGIC
q0] SCAN (0,q0), (1,q1), (1,accept)
q1] SCAN (0,q0), (1,q2)
q2] SCAN (0,q0), (1,q1), (1,accept)
```



```
.LOGIC
A] SCAN RIGHT (0,A), (1,B), (#,accept)
B] SCAN LEFT (0,C), (1,reject)
C] SCAN RIGHT (1,A)
```

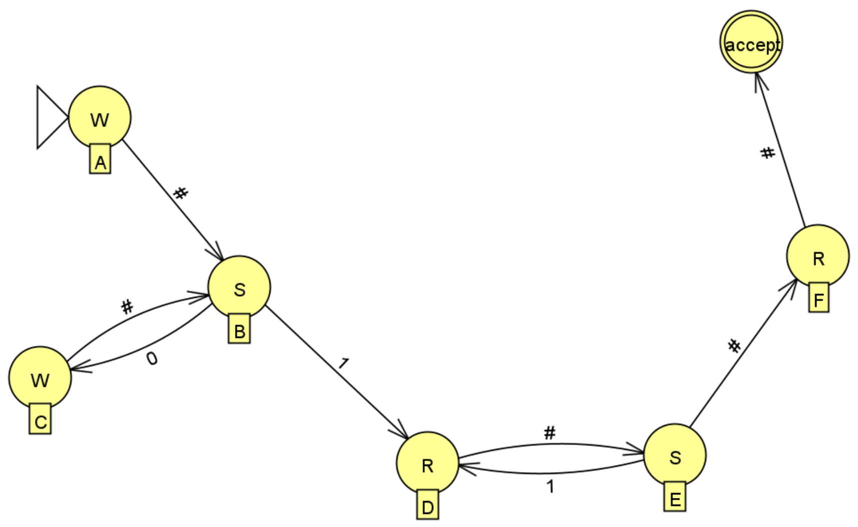


```
.DATA

STACK S1

.LOGIC

A] WRITE(S1) (#,B)
B] SCAN (0,C), (1,D)
C] WRITE(S1) (#,B)
D] READ(S1) (#,E)
E] SCAN (1,D), (#,F)
F] READ(S1) (#,accept)
```

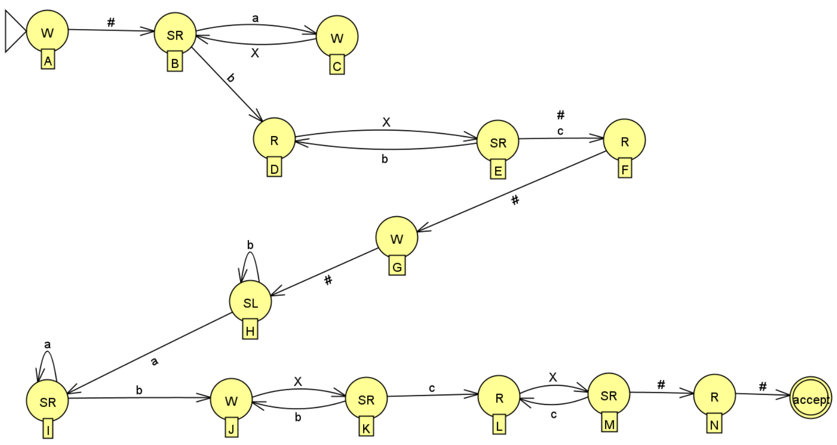


```
.DATA

STACK S1

.LOGIC

A] WRITE(S1) (#,B)
B] SCAN RIGHT (a,C), (b,D)
C] WRITE(S1) (X,B)
D] READ(S1) (X,E)
E] SCAN RIGHT (b,D), (c,F), (#,F)
F] READ(S1) (#,G)
G] WRITE(S1) (#,H)
H] SCAN LEFT (b,H), (a,I)
I] SCAN RIGHT (a,I), (b,J)
J] WRITE(S1) (X,K)
K] SCAN RIGHT (b,J), (c,L)
L] READ(S1) (X,M)
M] SCAN RIGHT (c,L), (#,N)
N] READ(S1) (#,accept)
```

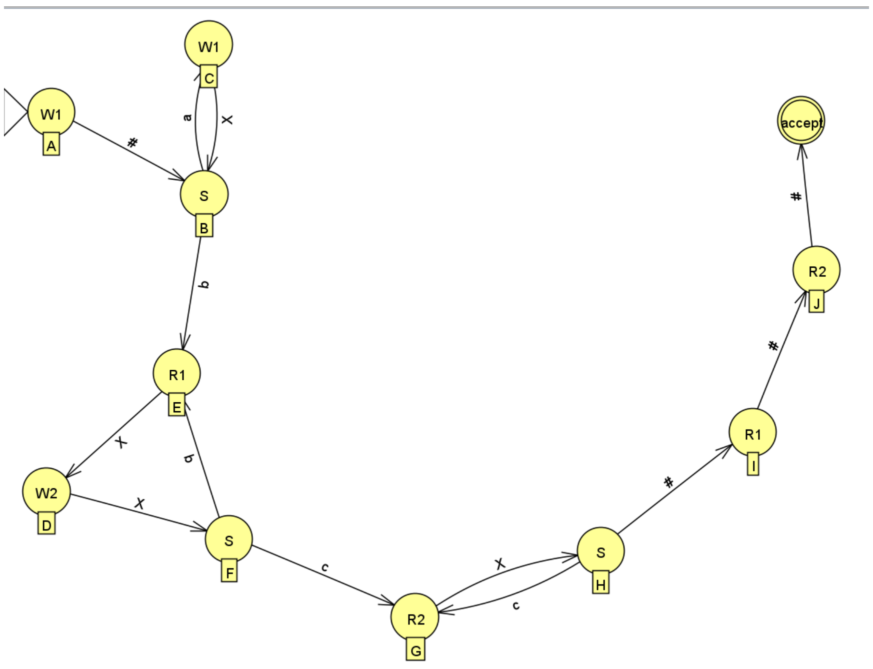


```
.DATA

STACK S1
STACK S2

.LOGIC

A] WRITE(S1) (#,B)
B] SCAN (a,C), (b,E)
C] WRITE(S1) (X,B)
D] WRITE(S2) (X,F)
E] READ(S1) (X,D)
F] SCAN (b,E), (c,G)
G] READ(S2) (X,H)
H] SCAN (c,G), (#,I)
I] READ(S1) (#,J)
J] READ(S2) (#,accept)
```

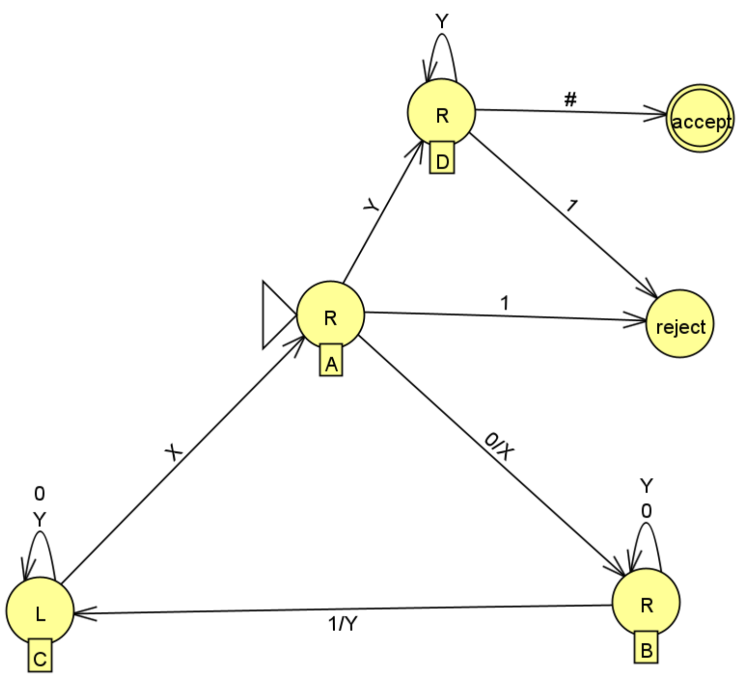


```
.DATA

TAPE T1

.LOGIC

A] RIGHT(T1) (0/X,B), (Y/Y,D), (1/1,reject)
B] RIGHT(T1) (0/0,B), (Y/Y,B), (1/Y,C)
C] LEFT(T1) (0/0,C), (Y/Y,C), (X/X,A)
D] RIGHT(T1) (Y/Y,D), (#/#,accept), (1/1,reject)
```



- - END OF PROJECT SPECIFICATIONS - -