Project 2: Turing's Assembler
Theory of Computation (CSCI 3500)
Total Points: 75

Due: December 5 by 11:59pm

This project is to be turned via your Git repo.

An assembly language is a low-level programming language for programming computational devices. A low-level programming language is one that has a tight correspondence with the hardware of the machine. In this project we develop an assembly language for Turing Machines. The following is an example TM assembler program:

```
-- Initialization:
{states: Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7,A,R}
{start: Q0}
{accept: A}
{reject: R}
{alpha: 0,1,#}
{tape-alpha: 0,1,#,x}

-- Main Algorithm:
--   0:
rwRt Q0 0 x Q1;              -- Read, Write, Right, Transition
rRl Q1 0;                    -- Read, Right, Loop.
rRl Q1 1;
rRt Q1 # Q3;                 -- Read, Right, transition.
rRl Q3 x;
rwLt Q3 0 x Q5;              -- Read, Write, Left, Transition.

--   1:
rwRt Q0 1 x Q2;
rRl Q2 0;                    -- Read, Right, Loop.
rRl Q2 1;
rRt Q2 # Q4;                 -- Read, Right, transition.
rRl Q4 x;
rwLt Q4 1 x Q5;              -- Read, Write, Left, Transition.

--  Find #:
rLl Q5 x;                    -- Read, Left, Loop.
rLt Q5 # Q6;                 -- Read, Left, Transition.

--   Reset:
rLl Q6 0;
rLl Q6 1;
rRt Q6 x Q0;

--   Accept:
rRt Q0 # Q7;
rRl Q7 x;
rRt Q7 _ A;
```

Comments start with a -- and can appear anywhere. The first part of each TM source file is the initialization of the hardware which is a list of configurations. Then the rest of the file is a series of commands that defines the algorithm of the TM. The previous example is the definition of the TM that accepts the language $L = \{w\#w \mid w \in \{0,1\}^*\}$.

The main part of this project is to define the compiler for TM source files. It is to prompt the user for the path to the TM source file, and for a word that is to be tested for acceptance with respect to the TM defined in the TM source file. Then output a complete trace of the machine. For example,

```
Please enter a path to the source file: copy.TM
Please enter a word: 01#01
[Q0]01#01
x[Q1]1#01
x1[Q1]#01
x1#[Q3]01
x1[Q5]#x1
x[Q6]1#x1
[Q6]x1#x1
x[Q0]1#x1
xx[Q2]#x1
xx#[Q4]x1
xx#x[Q4]1
xx#[Q5]xx
xx[Q5]#xx
x[Q6]x#xx
xx[Q0]#xx
xx#[Q7]xx
xx#x[Q7]x
xx#xx[Q7]
Accept: xx#xx[A]
```

There are several steps to the project which we describe below.

1 Preliminaries

You are allowed to use any programming language you like, and any libraries you like, except any libraries that implement Turing machines. You must implement those yourself.

Your project will be graded using testing, and not by examining your code. **Thus, while I will try to give partial credit when possible, if I cannot conduct the testing process, then a zero will be given.** Therefore, it is of the utmost importance that your project compiles and runs when you turn it in to get a passing grade.

You will need to turn in the following:

- All your source code, and
- instructions for building and running your project – this must include details of all the tools and software that I need to install to build your project.

Place your project in a directory (or folder) called project2 inside your Git repo.

While testing I will distinguish between two types of errors:

- A **graceful failure** which is one issued by the programmer and is not an exception due to the incorrect usage of some internal action. For example, if you call some library function and it triggers an exception that was not intentionally triggered by the programmer.
- All other errors are considered true errors.

If your program is given some input that it should reject it must do so using a graceful error. If not, then the current test will be considered a failure and no points will be rewarded for that test case.

2 Part 0: Implementing Turing Machines

The first part requires the implementation of Turing machines. To complete this part follow the definitions in the lecture notes. It might also be useful to create helper functions (or methods) that help keep track of the configurations of the machine as it runs. Lastly, in order for the Turing machine implementation to be complete a tape and a head pointing to a cell on the tape must be combined with the definition from the lecture notes. Then the input word must be written to the tape before running the machine.

3 Part 1: The TM Source Parser

Now the parser for a TM source file must be implemented. Every TM source file must contain two parts: the TM configuration and the algorithm for the transition function. The file must contain the configuration first. Comments can appear on any line and start with -- (including before the configuration).

The configuration of the TM consists of the following lines (they must appear in this order at the beginning of the file):

- States: `{states: q0,...,qn}` where each `qi` can be any string of alpha or numerical symbols.
- Start state: `{start: q}` where `q` is a state from the list of states given in the states configuration.
- Accept state: `{accept: A}` where `A` is a state from the list of states given in the states configuration.
- Reject state: `{reject: R}` where `R` is a state from the list of states given in the states configuration, and is distinct from the specified start state,
- Input alphabet: `{alpha: a1,...,am}` where each ai is any character at all.
- Tape alphabet: `{tape-alpha: t1,...,tm,_}` where each ti is any character at all, and the underscore (_) represents the blank symbol, and each input character in the input alphabet is an element of the tape alphabet.

The remainder of the file contains a list of commands describing the transitions of the Turing machine. These may occur in any order within the file. Each command ends with a ;. The following lists the possible commands and their intended semantics:

- `rwRt q t t' q';` : in state q where the head is reading `t` off of the tape, write `t'`, transition to state `q'` and move the head right.
- `rwLt q t t' q';` : in state q where the head is reading `t` off of the tape, write `t'`, transition to state `q'` and move the head left.
- `rRl q t;` : in state q where the head is reading `t` off of the tape, write `t`, transition to state `q` and move the head right.
- `rLl q t;` : in state q where the head is reading `t` off of the tape, write `t`, transition to state `q` and move the head left.
- `rRt q t q';` : in state q where the head is reading `t` off of the tape, write `t`, transition to state `q'` and move the head right.

- `rLt q t q';` : in state `q` where the head is reading `t` off of the tape, write `t`, transition to state `q'` and move the head left.

Finally, the parser then creates the Turing machine the source file describes. Note that each state and character used in the commands must be checked to insure they actually are given in the initialization of the machine.

4 Part 2: Putting it all Together

You are now ready to put all of your hard work together into one glorious function called main. This function should prompt the user for a path to a TM source file, and for a word over the alphabet the machine requires. Then outputs the list of configurations the machine enters. A configuration has the form $t_1 \cdots t_i[q]t_{i+1} \cdots t_n$ where each t_i is an element of the machines tape alphabet and q is a state of the machine. The configuration is a snapshot of the machines state. The tape contents is $t_1 \cdots t_i t_{i+1} \cdots t_n$, q is the state the machine resides in, and the machines head is currently reading the symbol t_{i+1}. The final output should be the complete list of configurations that the machine entered, and whether the word was accepted by the machine or not.

Points will be deducted if the final submission's output contains debugging information.

That's it!

You may prompt the user in anyway you wish. You do not have to use a GUI if you do not want to, but you can if you do.