Ryan Darras PJ06 project writeup (Sorry for more than 5 pages, bunch of images though)

Preface

This project was developed using Visual Studio 2017 community edition in C#. There are no customizations made that would interfere with grading or someone else opening and running the project. By default, when the program is run it saves the files into this location ~\bin\Debug\Outputs however I have copied this output into the results folder at the top of the zip file. Similarly, that same location except instead of \Outputs, do \Machines in order to read in the machine files. It is currently hardcoded to read in p01.tm and p01.tm. The projects starting file is program.cs. I have commented most of the code in the solution to where it shouldn't be too difficult to understand the strategy as you read along.

Emulation

I simply used my project from PJ05 to emulate my machines, but I used another program to generate them (manually adding 600 lines of x,y,z,b,R = yuck). While doing my testing, I discovered that I had a few errors in PJ05 so I have fixed them since. Considering the technical approach for emulation was identical to PJ05, I am going to simply copy/paste my documentation from PJ05 here.

All in all, this project was much easier than PJ01 in my opinion. Probably because a majority of the thought processes that went on in PJ01 was the same/similar to this one, so I already had it figured out. The overall goal of this project is as follows:

1) Load and save the input file as a list of strings. Call this "INPUT".
2) Create a new TM and send it a list of strings "MACHINE" that equates to the machine files. Example item in this list would be "0,98,1,97,R".
3) TM constructor reads each line in the list of strings in MACHINE and deconstructs them into FromState, ReadSymbol, ToState, WriteSymbol, and HeadDirection.
4) Uses the above variables to create a Transition struct and stores all of these objects in a Dictionary<int, List<Transition>> which is a dictionary of transitions, and the key is the from state. IE: When we are in state X, we can look up the transitions that are possible from said state.
5) After the TM is built, we run each string in INPUT over the machine and store the results in a List<string> output. (More info in next section)
6) Print output to the output files line by line.

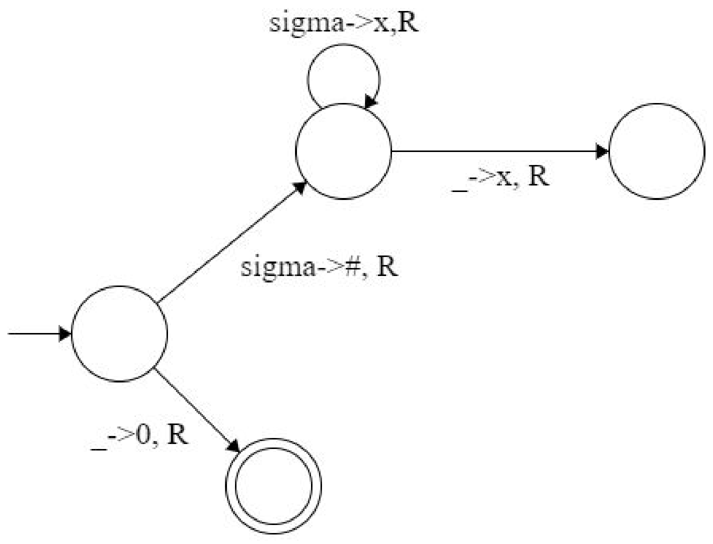
Running the TM

To run the machine, you give it a single string as an input and it will do these steps:

1) Copy the input string onto the tape.
2) Loop until we get to a halting point (accept/reject/no transition)
    a) Figure out what transition we need to make based on current state and tape head.
    b) If we have a valid transition, continue; otherwise reject.
    c) Update our CurrentState to the next state.
    d) Write our new symbol to the tape at tape head location
    e) Update tape head location based on the transition data.
        i) If tape head location == 0, it stays at 0.
        ii) If left, it goes left.
        iii) If right, it goes right.
    f) If we are transitioning right, we might need to add a blank space to our tape because I chose to use a list as the tape. Handle that here.
    g) Check to see if we are in the accept or reject states. If so, halt.
3) Remove any blanks from the end of the tape and return it as the output.

The following is the process at which I generated the machine for p01

This part of the machine is the initial setup. It will take any input string of size > 0 and replace it with "#xxxxxx". For example: Input = "hello", output = "#xxxxx". In the case that the input is of size 0, we will simply put 0 on the output and we are completely finished.



This module is used after the initial setup module to add leading 0’s before the #xxxxx input string. Because we are running this on the same input string as PJ01, we know that the largest input length is < 145. Thus we are going to need to represent up to 255, which is 8 bits. So we are going to run this module 8 times. (NOTE: For a more elegant solution, I would dynamically add range to the binary value as we went along. This would, however, require some tricky manipulation that isn’t necessary for this input file.)

This is an example of the above logic put together to run 8 times. This would be the complete module that ran after the initial setup.

This module is used to handle finding the next index of the input string. If starting on the right hand side of the input, it will continue left until it finds the # symbol, at which point it will move one unit right and start processing on the first index of the input string. It will continue right throughout the string until it finds either the first x, or the first blank space. If it finds the first x, it replaces the x with a y and moves left passed the # symbol into the binary number. If it doesn't find an x, we continue to the module that will remove the y's from the tape so that we can return only the binary value.

This is the counting module. When it is entered, it will add exactly 1 to the binary value.

This module removes the leading 0’s from the binary value. For example, it will change 00011011 to simply 11011 which will result in the final output of the entire system

This is the final product, or at least very close to. I added a few transitions into the project that might not be in this diagram.

The general thought process behind p02 is very similar to p01 so I won’t include every single image as I did for p01. The differences are going to be in the counting process and the “remove leading 0’s” process.

The below image is a “decent” representation of what I did. There are a few places I modified that aren’t in the image below. The code generating file shows the true representation of what this TM should look like.