Due 5/7

Construct a Turing machine that adds two binary real numbers. An input will be of the form X#Y, where X and Y are elements of $\{0, 1\}^+.\{0, 1\}^+$. In particular, $X = x_n x_{n-1} \ldots x_1 x_0 . x_{-1} x_{-2} \ldots x_{-k}$, and $Y = y_m y_{m-1} \ldots y_1 y_0 . y_{-1} y_{-2} \ldots y_{-l}$ with x_i, y_i in $\{0, 1\}$, $X = (x_n \times 2^n) + (x_{n-1} \times 2^{n-1}) + \ldots + (x_1 \times 2^1) + (x_0 \times 2^0) + (x_{-1} \times 2^{-1}) + \ldots + (x_{-k} \times 2^{-k})$, and $Y = (y_m \times 2^m) + (y_{m-1} \times 2^{m-1}) + \ldots + (y_1 \times 2^1) + (y_0 \times 2^0) + (y_{-1} \times 2^{-1}) + \ldots + (y_{-l} \times 2^{-l})$. Your Turing machine must be a single tape, one way infinite, deterministic Turing machine. When the Turing machine completes, the tape should contain Z, where Z = X + Y. You do not need to delete X#Y from the tape, you can simply position the Turing machine's read/write head at the beginning of Z (leftmost symbol of Z). For your result to be correct, it will not have any leading 0s to the left of the decimal point unless Z is less than 1, and the result will not having any trailing 0s to the right of the decimal point, unless Z is an integer. That is, the following are invalid, 01.0 and 1.10, and likewise the following three are valid, 0.1, 1.0, and 0.0. For the trailing 0s, you can move to the right end of Z and move left over 0s, overwriting them with blank spaces, until either a 1 or a decimal point is found. If a decimal point is found, the blank space to the right of it can be changed back to a 0. For leading 0s, when you position the read/write head on the leftmost symbol of Z, you can simply move to the right of any leading 0s until either a 1 or a decimal point is found. If a decimal point is found, simply move left. For this assignment you may want to use blocks (subroutines in JFLAP) to build your Turing machine. You can also make use of the S directive for read/write head motion (L – left, R – right, S – stay). You may use the "~" to match any symbol for reading/writing in a transition. Otherwise any transition must read/write a single symbol. You may not use the JFLAP transitions like "(a,b,c}w; w, R)" (stores a symbol in a JFLAP internal variable). Your Turing machine cannot make use of the blank spaces to the left of the input string. JFLAP has some unhappiness with filenames containing special characters and I don't know all the symbols that cause problems (I stick with alphanumeric and the underscore symbol, and have had problems with $, #, and the blank space in block file names). When you create a block, I would suggest you create it as a separate file, test it, and then insert it into your main program (or a block that goes into your main program). I've heard from students that if you are editing a block from within your main program and you save the block before closing it, it will overwrite your file with the block you are editing (you need to exit block edit mode prior to saving). Keep backup copies of all of your file (often). You can do a save as while editing a block to save the block as a separate file.

It took me about three hours to implement my version of the program, an hour to test/debug it, and probably an hour to write a Java program to verify the output was correct (unfortunately Java's BigDecimal class doesn't support non-base 10 representations). For my final test I generated 1000 random strings of the form X#Y where X and Y have length between 4 and 10 symbols and ran my program against the strings and verified that the result of adding X and Y was correct.

Below is some additional information about my implementation.

My Turing machine uses 4 blocks. The blocks perform the following actions.

- InsertLeftTerminatorAndAppendPoundSign – block that inserts a "$" at the left end of the input, shifting all of the input to the right one position, and appends a "#" at the right end of the string
  - The block has 8 states and rewinds the tape leaving the read/write head under the "$"
  - The "#" appended to the input is to mark where Z (Z = X + Y) begins
- fixOrder – block that reorders X & Y if X has less symbols to the right of the decimal point than Y does
  - The block has 22 states
  - After this block completes we have the first of the two string has at least as many symbols to the right of the decimal point as the second – this is to make is more efficient to pad the

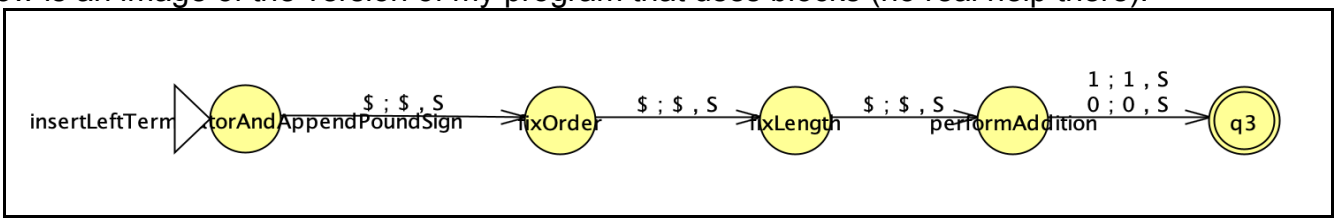
second of the two with 0s such that they have the same number of symbols to the right of the decimal point

- fixLength – block that appends 0s to the end of Y to make X & Y have the same number of digits to the right of the decimal point
  - The block has 12 states
- performAddition – block that actually adds X & Y
  - The block has 38 states
  - We will discuss performing binary addition in class
- main program – uses the four blocks and one accept state

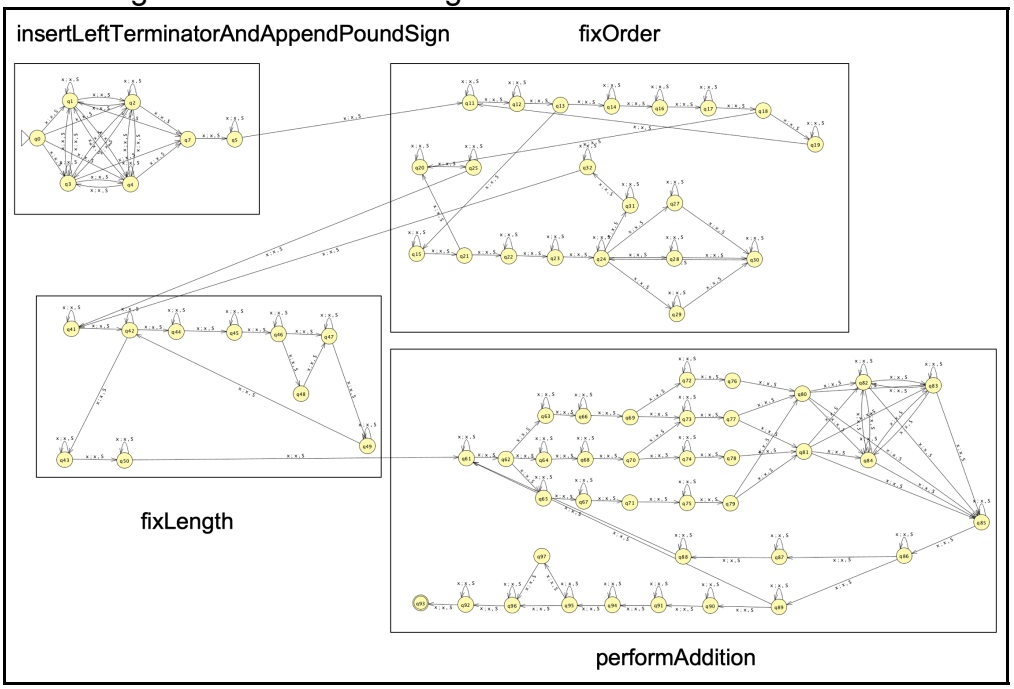
My Turing machine uses an input alphabet of {0, 1, ., #} and a tape alphabet of {0, 1, ., x, a, b, #, $, blank space}. The x is used to mark symbols of X and Y as having been processed and a and b are used to mark 0 and 1 when they need to be restored in the future (in fixOrder & fixLength).

I also did a verion of the program without using blocks. Assuming I counted correctly, it has 75 states.

Below is an image of the version of my program that uses blocks (no real help there).



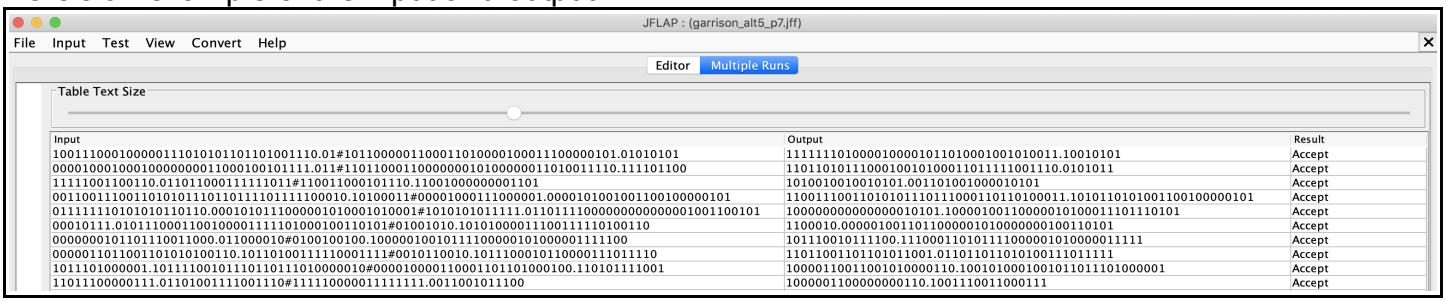
Below is an image of the version of my program that doesn't use blocks. All of the transitions have been replaced with a single transition. The image is to show the same structure works without blocks.



Upload your JFLAP file to the assignment in Brightspace by 11:59pm on the date due. The filename must be your last name followed by "_p7.jff" (as an example, my filename would be "garrison_p7.jff").

Due 5/7

Here's an example of the input and output.



Input	Output	Result
1001110001000001110101011011010011110.01#10110000011000110100001000111000000101.01010101	11111110100001000010110100010010100011.10010101	Accept
0000100010001000000001100010010111 1.011#11011000110000000101000000110100111 10.111101100	11011010111000100101000110111110011110.0101011	Accept
11111001100110.011011000111111011#11001100010111 0.1100100000000 1101	10100100100101 01.001101001000010101	Accept
0011001110011010101110110111101111100010.10100011#0000100011100000 1.000010100100110010000 0101	11001110011010101110111000110110100011.10101101010011001000000101	Accept
0111111101010110110.000101011100000101000101000 1#101010111111.011011110000000000000001001100101	10000000000000000010101.10000100110000010100011101110101	Accept
00010111.010111000110010000111110100010011010 1#01001010.10101000011100111110100110	1100010.000001001101100000101000000100110101	Accept
0000000101101110011000.01100001 0#0100100100.1000001001011100000101000001111100	10111001011110 0.111000110101111000001010000011111	Accept
0000011011001101010100110.101101001111100011 11#0010110010.101110001011000011101111 0	11011001101101011001.011011011010100111011111	Accept
1011101000001.101111001011101101110100000 10#0000100001100011011010000100.110101111001	10000110011001010000110.100101000100101101110100000 1	Accept
11011100000111.011010011110011 10#1111100000111111 1.0011001011100	1000001100000000110.1001110011000111	Accept

I will be testing your program with 20 – 40 strings of varying length, although most of them are fairly long. It takes my program less than ten seconds to process the ten strings shown above. Each of the strings above consists of X and Y having length between 30 and 50 symbols.

If you sumbit something that is partially functional by the due date, you can continue submitting updates for full credit until at least the end of final exams.