AUTOMATA AND FORMAL LANGUAGE THEORY

Turing Machine Lab

2018-19 GROUP 88

AUTHORS:

Rafael Pablos Sarabia (100372175) 100372175@alumnos.uc3m.es

Juan Sánchez Esquivel (100383422) 100383422@alumnos.uc3m.es

TABLE OF CONTENTS

1. UNDERSTANDING AND PLANNING

1.1. GENERAL CONCEPTS

For the final practice in Automata and Formal Languages Theory class, we have had to develop a Turing Machine to calculate the square root of a rational number as explained in the guide provided for the assignment.

In order to be able to start this practice, we first had to carefully read and understand the provided guide. The explanations provided in the guide allowed us to develop an algorithm and think about the tape organization for the implementation of the Turing Machine. These two important aspects are detailed below. Once we were clear on the possible algorithm and tape organization, we started to create submachines which performed a clear task and could be of use in the final Turing Machine. The different submachines, as well as the final Turing Machine made up of many submachines, will be explained in the following sections on the document.

1.2. ALGORITHM

Since the input tape can have a decimal number, we considered to make a copy (X') of the initial value of X in which the dot was not included. Therefore, if X is 11.0011 (3.1875), then X' will be 110011 (51). This allows us to calculate the square root of X' which has no decimal part, instead of the square root of X which has a decimal part that has to be taken care of. Once we have obtained the square root of X', we fix it so that the decimal part is accounted for. For the same values of X and X' previously given, we would obtain that sqrt(X') = 111 (7) and therefore, sqrt(X) = 1.1100 (1.75). This is true because of the following properties:

$$sqrt(11.0011_2) = sqrt(110011_2 * 2^{-4}) = sqrt(110011_2) * sqrt(2^{-4})$$

$$= sqrt(110011_2) * 2^{-2} = 111_2 * 2^{-2} = 1.11_2$$

After reading and understanding the given guide with instructions, we developed an initial algorithm in order to plan the needed Turing Machines and organize the tape. The algorithm is as follows:

1. Check value of N and finish with output = 0 if N = 0
2. Copy the value of X to X' without the dot
3. Store ε_0 on the tape
4. Store Z_0 on the tape
5. Calculate $Z_i + \varepsilon_i$
6. Calculate $(Z_i + \varepsilon_i)^2$
7. Calculate X'- $(Z_i + \varepsilon_i)^2$
    - If result = 0 → X' = $(Z_i + \varepsilon_i)^2$ → sqrt(X') = $Z_i + \varepsilon_i$ → Go to 9
    - If result < 0 → X' < $(Z_i + \varepsilon_i)^2$ → $Z_{i+1} = Z_i$
    - If result > 0 → X' > $(Z_i + \varepsilon_i)^2$ → $Z_{i+1} = Z_i + \varepsilon_i$
8. Subtract one to counter N (number of iterations left)
    - If N = 0 → Z_{i+1} = sqrt(X')
    - If N ≠ 0 → Calculate ε_{i+1} → Go to 5
9. Adapt sqrt(X') to obtain sqrt(X)

1.3. TAPE ORGANIZATION

The tape has been organized in the following way in order to have all parameters organized and in a sequential order, so transitions are the shortest possible. We believe that this arrangement is quite efficient and helps to have an organized tape.

xx.xx	$	xxx	X	xxxx	E	xxxxx	Z	xxxxx	S	xxxxx	C	xxxxxxxxxx	R	xxxxxxxxxx
X		N		X’		ε_i		Z_i		ε_i+Z_i		$(\varepsilon_i+Z_i)^2$		$X’- (\varepsilon_i+Z_i)^2$

Initial tape configuration from input

- **X:** Any rational number bigger than or equal to 1 for which the user wants to obtain the square root. It includes implicit accuracy since the result must have the same number of significant decimal positions.
- **N:** Number of iteration for the Turing Machine to obtain the closest value possible to the actual value of the square root.
- **X’:** Value of X without the decimal dot. This copy of the value of X is created in order to ease the calculation of the square root of a decimal number by calculating first the square root if the number has no decimal dot.
- **ε_i:** Value that always satisfies $(Z_i)^2 < X < (Z_i+ \varepsilon_i)^2$ and $\varepsilon_{i+1} = \varepsilon_i/2$. Initial value given ε_0 determines the efficiency and precision of the algorithm. It was initially set equal to X to ease the implementation of the Turing Machine. Once we had a successful Turing Machine, we changed it to 2^{k+1} (with k being number of bits of X’) in order to improve precision without greatly affecting efficiency.
- **Z_i:** Value that always satisfies $(Z_i)^2 < X < (Z_i+ \varepsilon_i)^2$ and is updated according to the following equations: - If $X’ < (Z_i + \varepsilon_i)^2 \rightarrow Z_{i+1} = Z_i$ - $X’ > (Z_i + \varepsilon_i)^2 \rightarrow Z_{i+1} = Z_i + \varepsilon_i$

  Initial value for Z_i, similar to ε_i, will affect the efficiency and accuracy of the implementation. We have decided to initialize Z_0 as 0, as initially recommended. It has the same number of bits as ε_i.
- **ε_i+Z_i:** Sum of the two parameters to later calculate the square. It has the same number of bits as Z_i and ε_i. This will not be a problem because if we need one more bit, we would not be satisfying the condition $(Z_i)^2 < X < (Z_i+ \varepsilon_i)^2$.
- **$(\varepsilon_i+Z_i)^2$:** Square of the sum in order to compare it to X’ later. When calculating the square of a decimal number, we must consider that the number of decimals will be doubled. Therefore, we must truncate the number in order to compare it correctly to X’ as it will be explained later.
- **X’ - $(\varepsilon_i+Z_i)^2$:** Value used to check how the value of Z_{i+1} should be updated. If the result is positive, a binary number will be in this section. If result is 0, 0s will be on the tape. In case result is negative, the tape will be empty after the R.

2. TURING MACHINE AND SUBMACHINES

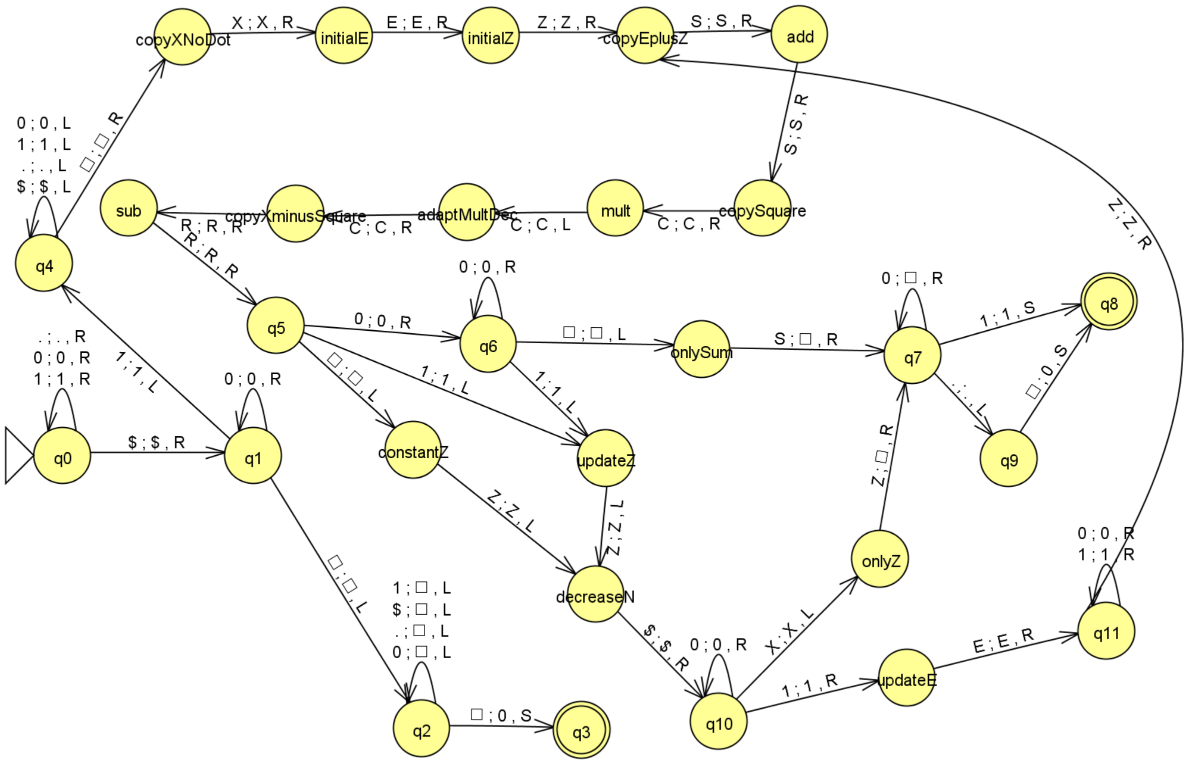
This section of the report details all Turing Machines used in order to obtain the square root of a rational number according to the algorithm and tape organization previously defined. They are listed and explained in a logical order that follows the steps of the algorithm and therefore, the execution steps of the complete Turing Machine (final.jff).

2.1. final.jff

This Turing Machine is the complete version that uses all other submachines that will be explained later. It receives an input of the form 101.11$111 where 101.11 represents X and 111 N. The result is the square root of X after N iterations of the algorithm with the same number of decimal digits and without 0s to the left of the number. The result for the previous input according to the Turing Machine we have developed is 10.01 which seems valid since 101.11 is 5.75, whose square root is 2.40, and 10.01 is 2.25.

final.jff implements the algorithm previously explained by combining all other submachines which will be detailed later. As explained, it first checks if N is not 0 (states q0 and q1). If N is 0, output on the tape is 0 (states q2 and q3). If N is not 0, it moves the header back to the left-most digit (state q4). Then, X' is added to the right of the tape after an X digit (block copyXNoDot). Similarly, initial values for ε_0 (block initialE) and Z_0 (block initialZ) are copied to the right of the tape after E and Z respectively. Then, the tape is prepared for calculating ε_i+Z_i (block copyXNoDot.jff) and the sum is performed (block add.jff). Next, for the calculation of $(\varepsilon_i+Z_i)^2$, the tape is prepared (block copySquare.jff) and the multiplication performed (block mult.jff). The result of the square must be adapted as explained in Section 2.11 (block adaptMultDec.jff). To compare the obtained value with X' in order to determine how to update Z_i, we calculate X' - $(\varepsilon_i+Z_i)^2$ by preparing the tape (block copyXminusSquare.jff) and performing the subtraction (block sub.jff). Then, result is analyzed, and different steps are taken if result is 0, negative or positive (states q5 and q6). If result is 0, ε_i+Z_i is left only on the tape (block onlySum.jff) and non-significant 0s on the left are removed (states q7 to q9). If result is negative, Z value is not updated but the tape is updated for new iteration (block constantZ.jff). On the other hand, if result is positive, Z value is updated to ε_i+Z_i and tape is cleared for new iteration (block updateZ.jff). Then, N is decreased since a new iteration has been performed (block decreaseN.jff). The value of N is checked to decide when to stop algorithm (state q10). If N is 0, the current approximation stored in Z is shown and adapted (block onlyZ.jff) and non-significant 0s on the left are removed. If N is not 0, value of ε_i is updated by dividing current ε_i value by 2 (block update.jff) and header is moved back to Z delimiter (state q11) in order to start a new iteration calculating sum, square, subtraction…

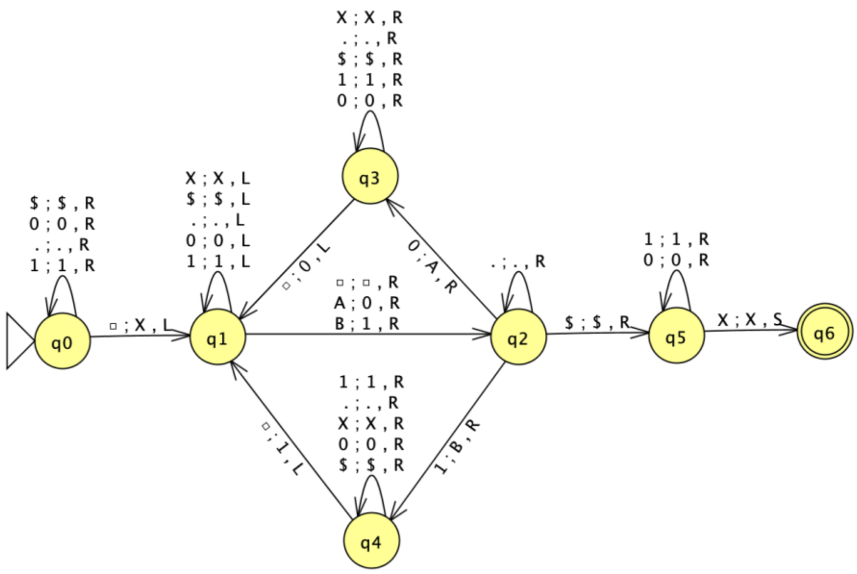
Detailed tests on Section 3 may help to understand the implementation. These tests show the values of each parameter for each iteration until the result is found.



2.2. copyXNoDot.jff

This Turing Machine makes a copy on the right of the tape of the value of X on the input tape without copying the dot. Before copying the value, it adds an X to use it as a delimiter. When this block is accessed from final.jff, the tape is xx.xx$xxx and the header points to the left most digit on the tape. When this block is complete, the tape contains xx.xx$xxxXxxxx and header point at the X delimiter.

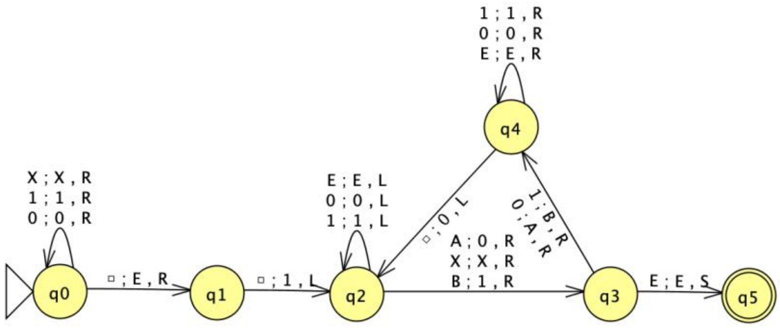
copyXNoDot.jff first goes to the right of the tape and adds the X delimiter (state q0). Then, it copies the value of X skipping the decimal dot (states q1 to q4) and moves the header to the X delimiter (states q5 and q6).



2.3. initialE.jff

This Turing Machine initializes the value of ε_0 after adding a delimiter, E, to the right of the tape. After some improvements, we decided to initialize ε_0 to 2^{k+1}, with k being the number of bits of X', in order to improve precision and efficiency as explained in section 3. When this block is accessed from final.jff, the tape is xx.xx$xxxXxxxx and the header points at delimiter X. When the block is complete, the tape contains xx.xx$xxxXxxxxEeeeee and header points at delimiter E.

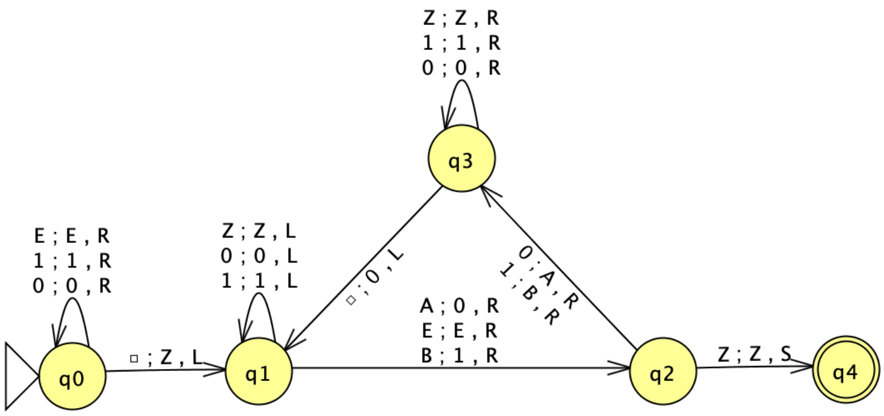
initialE.jff first goes to the right of the tape and adds the E delimiter followed by a 1 (states q0 and q1). Then, it adds as many 0s as digits that X has (states q2 to q4) and moves stops header at E delimiter (state q5).



2.4. initialZ.jff

This Turing Machine initializes the value of Z_0 after adding a delimiter, Z, to the right of the tape. As recommended on the guide, Z_0 is initialized to 0, with the same number of bits as ε_0. When this block is accessed from final.jff, the tape is xx.xx$xxxXxxxxEeeeee and the header points at delimiter E. When the block is complete, the tape contains xx.xx$xxxXxxxxEeeeeeZzzzzz and header points at delimiter Z.

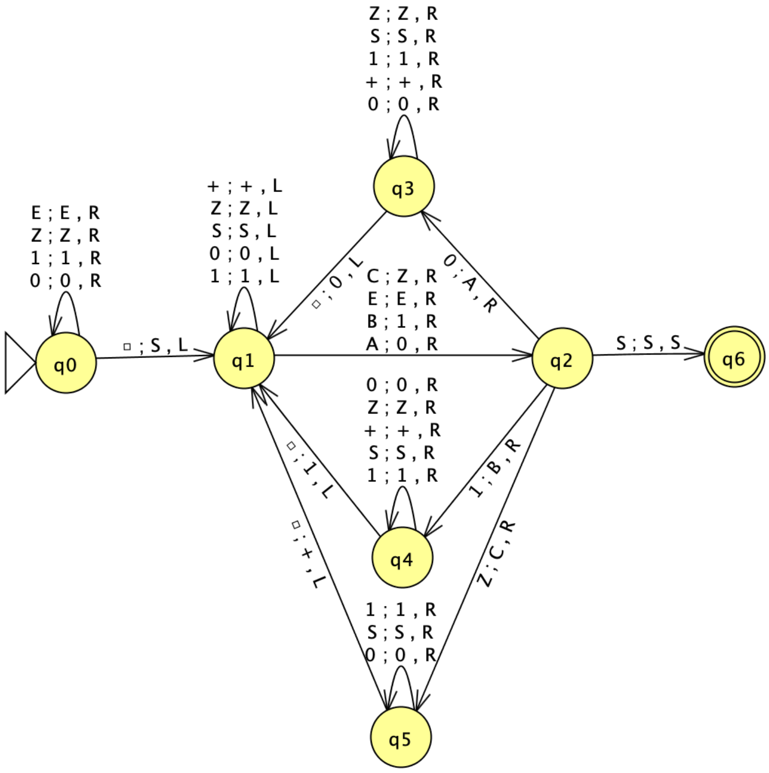
initialZ.jff first goes to the right of the tape and adds the Z delimiter (state q0). Then, it adds as many 0s as digits that X has (states q1 to q3) and moves stops header at Z delimiter (state q4).



2.5. copyEplusZ.jff

This Turing Machine initializes prepares the tape to calculate $\varepsilon_i + Z_i$. Since we want to preserve the values of ε_i and Z_i, then we make a copy of them after a delimiter S and add a '+' between them. When this block is accessed from final.jff, the tape is xx.xx$xxxXxxxxEeeeeeZzzzzz and the header points at delimiter Z. When the block is complete, the tape contains xx.xx$xxxXxxxxEeeeeeZzzzzzSeeeee+zzzzz and the header points at delimiter S.

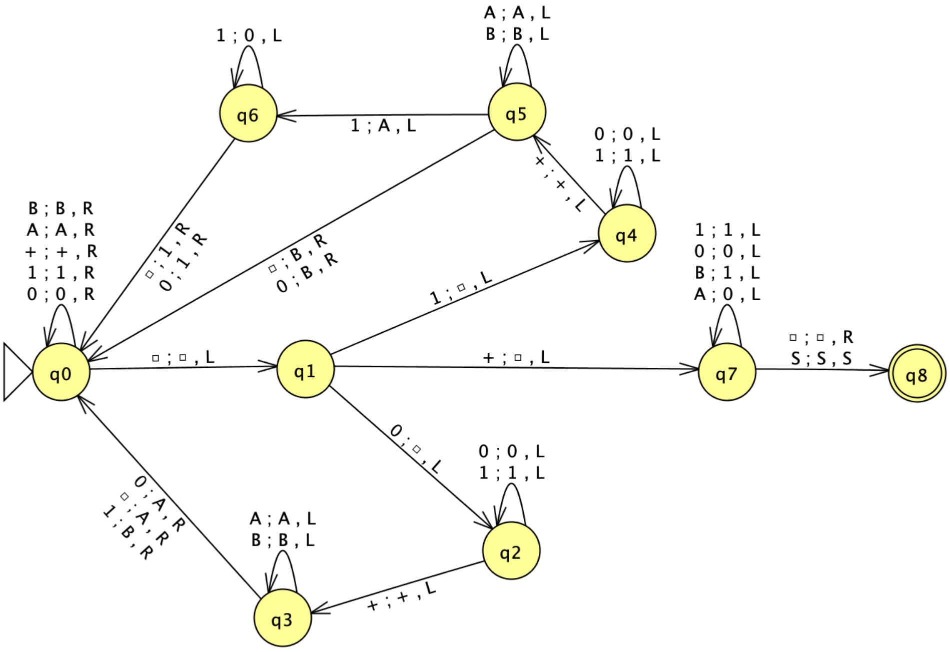
copyEplusZ.jff first goes to the right of the tape and adds the S delimiter (state q0). Then, it copies the value of ε_i after the S delimiter (states q1 to q4). After that, the plus sign is added (states q2 and q5), Z_i is copied (states q1 to q4) and the header stopped at delimiter S (state q6).



2.6. add.jff

This Turing Machine calculates the sum of two binary numbers, specifically $\varepsilon_i + Z_i$. In order to satisfy the condition $(Z_i)^2 < X < (Z_i + \varepsilon_i)^2$, this sum will never need one more bit than the number of bits of ε_i or Z_i, so we don't need to worry about the fact that the tape is occupied on the left. When this block is accessed from final.jff, the tape is xx.xx$xxxXxxxxEeeeeeZzzzzzSeeeee+zzzzz and the header points at the digit on the right of delimiter S. When the block is complete, the tape contains xx.xx$xxxXxxxxEeeeeeZzzzzzSsssss and the header points at delimiter S.

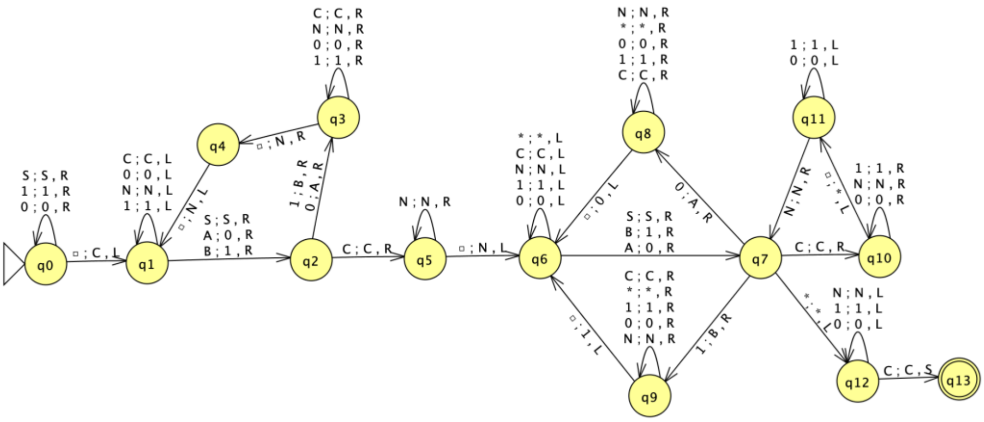
add.jff is an adapted version of the one given in class. It takes the right most digit not added yet from the second operand and adds it to the first one. If the digit to add is a 0, only values are marked as added (states q0 to q3). If the digit to add is a 1, it will be added differently to 0 (states q0, q1, q4 and q5) and to 1 (states q0, q1, q4 – q6). Marked digits are changed back to 0s and 1s (state q7) and header is stop at delimiter S (state q8).



2.7. copySquare.jff

This Turing Machine prepares the tape to calculate the square of two binary numbers, specifically $(\varepsilon_i + Z_i)^2$. Since mult.jff calculates the multiplication of two binary numbers but needs to have blank spaces of the left, this submachine reserves space on the tape for the multiplicator to be able to operate and sets up the multiplication expression. When this block is accessed from final.jff, the tape is xx.xx$xxxXxxxxEeeeeeZzzzzzSsssss and the header points at delimiter S. When the block is complete, the tape contains xx.xx$xxxXxxxxEeeeeeZzzzzzSsssssCNNNNNNNNNNNsssss*sssss and the header points at delimiter C.

copySquare.jff first goes to the right of the tape and adds delimiter C (state q0). Then, it adds (2k+1) Ns (states q1 to q5) on the right of the tape to reserve the needed space for the multiplication (with k being the number of digits of the sum $\varepsilon_i + Z_i$). This is due to the fact that the result can have 2k digits and the multiplicator needs one more digit for the '+' sign when operating. Then, sssss*sssss $((\varepsilon_i + Z_i)*(\varepsilon_i + Z_i))$ is copied on the right (states q6 to q11) and header is moved back to delimiter C (states q12 and q13).

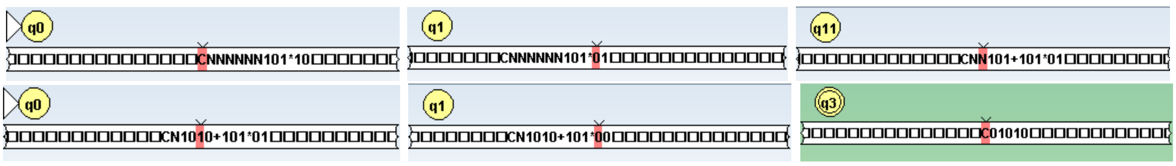


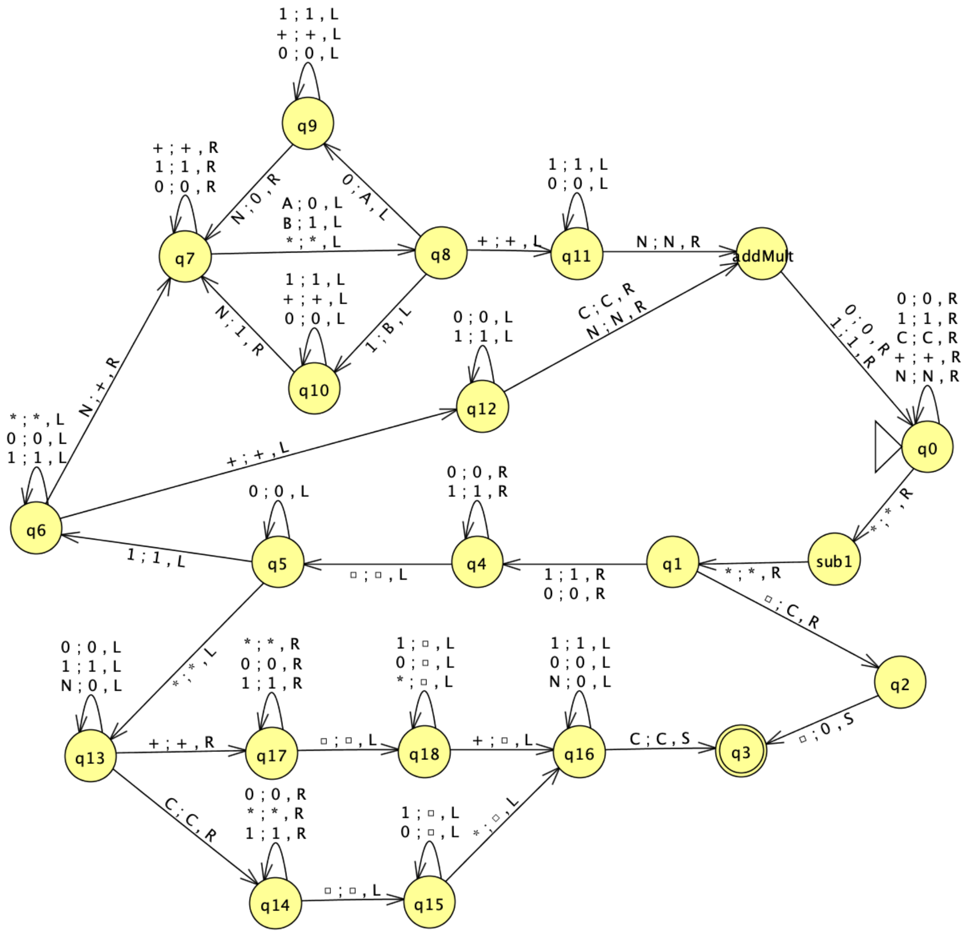
2.8. mult.jff

This Turing Machine calculates the multiplication of two binary numbers, specifically $(\varepsilon_i + Z_i)^2$. It needs to have at least (a+b+1) N digits on the left to be able to operate (with a and b being the number of digits of each operand). When this block is accessed from final.jff, the tape is xx.xx$xxxXxxxxEeeeeeZzzzzzSsssssCNNNNNNNNNNNsssss*sssss and the header points at delimiter C. When the block is complete, the tape contains xx.xx$xxxXxxxxEeeeeeZzzzzzSsssssCcccccccccccc and the header points at delimiter C.

mult.jff obtains the result of A*B by adding A+A…+A (B times). It first goes to the right of the tape and subtracts one unit from the second operand (state q0 and block sub1.jff). If the second operand was 0, it will become □, and the machine will show a 0 on the tape (states q1 to q3). If the second operand was not a 0, it checks if its current value after subtracting 1 is 0 (states q4 and q5). If the second operand is now 0, multiplication has been completed. Otherwise, it continues with the multiplication process. If it is the first iteration of the multiplication and the sum has not yet been initialized, tape is prepared as A+A*(B-1) by adding the '+' sign (states q6 and q7), copying the value of A on the left of the '+' sign (states q7 to q10) and moving the header to the left-most digit of A (state q11). Then, A+A is calculated (block addMult.jff) and tape looks like S+A*(B-1) where S is A+A. Then, the loop is started again by decreasing B, checking if it is 0 and continuing the multiplication process if not. If B is not 0, since '+' sign has been added, state q12 moves header so that block addMult.jff can add the values and continue the process. Once the second operand becomes 0, result must be shown. If initial value of B was 1 and '+' sign was never added, A is left on the tape (states q13 to 15). If initial value of B was not 0 and sums have been calculated, S is left on the tape (states q16 to q18). Finally, header is moved back to the C delimiter.

The following Step By Building Block test to multiply 101*10 may help clarify the implementation.

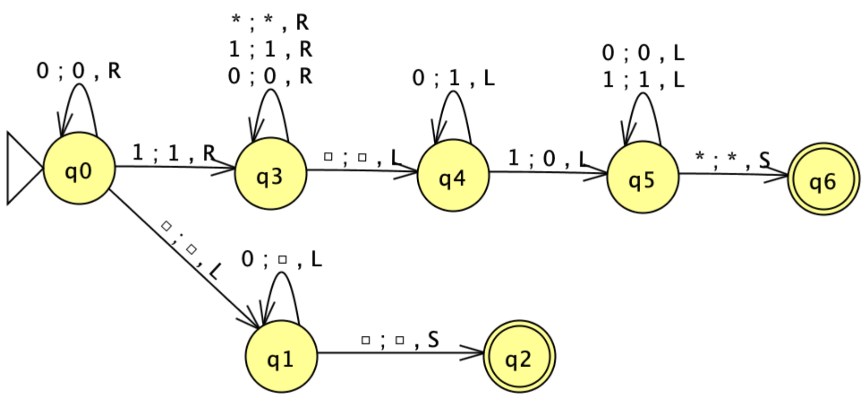




2.9. sub1.jff

This Turing Machine decreases the value of a binary number by one unit. If the input is 0, the result will be blank spaces. When this block is accessed from mult.jff, the tape is […]*aaaaa and the header points at the digit on the right of ‘*’. When the block is complete, the tape contains […]*bbbbb and the header points at ‘*’ with aaaaa=bbbbb+1.

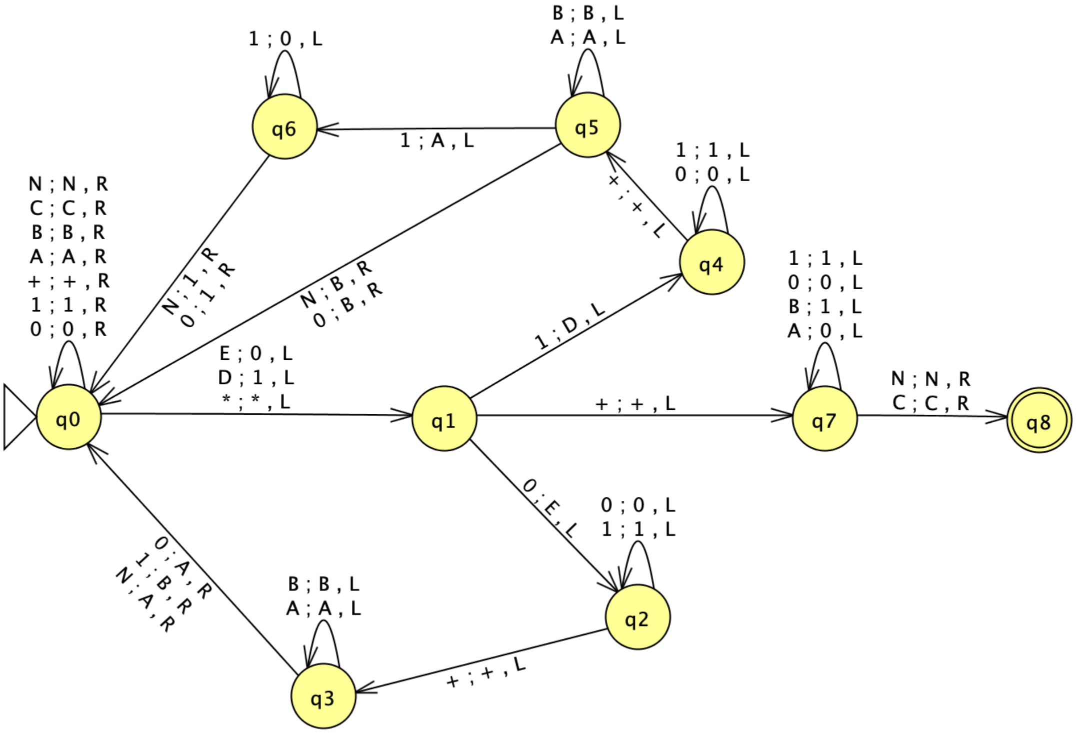
sub1.jff calculates the predecessor of a binary number by first checking if the number is 0 and returning blank spaces if it is (states q0 to q2). If number is not 0, the right-most number is found (state q3). Every 0 is translated into a 1 (state q4) until a 1 is found, which is translated into a 0 (states q4 and q5). Once the 1 has become a 0, the header is move back to the ‘*’ digit (states q5 and q6).



2.10. addMult.jff

This Turing Machine calculates the sum of two binary numbers for the multiplication Turing Machine. When this block is accessed from mult.jff, the tape is […]CNNNaaa+bbb*[…] and the header points at the left most digit on the left operand (a). When the block is complete, the tape contains […]CNNcccc+bbb*[…] and he header points at the left-most digit of c, with cccc = aaa+bbb.

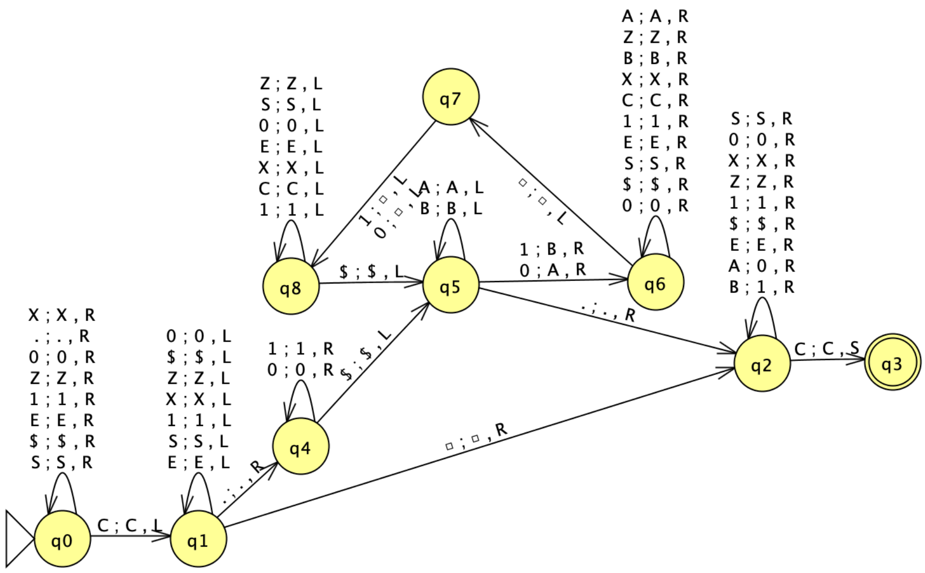
addMult.jff calculates the sum of the two binary numbers during the multiplication process in a similar way to add.jff. The main differences are the digits on the tape that can be replaced and bound the numbers to be added, as well as the fact that this adder does not only show the result, but also the second operand. The Turing Machine start by adding digit by digit from right to left. If digit to be added is 0, digits are marked and next digit to add is found (states q0 to q3). If digit to be added is a 1, it is added differently to a 0 (states q0, q1, q4 and q5) than to a 1 (states q0, q1, q4 to q6). Then, all marked digits are changed back to binary and header is moved to the left-most digit of the result (states q7 and q8).



2.11. adaptMultDec.jff

This Turing Machine truncates the square previously calculated and stored after delimiter C if X is a decimal number. This is due to the fact that when a decimal number is squared, the number of decimal positions is doubles. Since we want to have certain implicit number of decimals and we are not using a dot to represent it, we will remove the extra decimals generated using this Turing Machine. When this block is accessed from final.jff, the tape is xx.xx$xxxXxxxxEeeeeeZzzzzzSsssssCccccccccccc and the header point one digit to the left of delimiter C. After the block is complete, ccccccccc will not contain the k least significant digits (with k being the decimal digits of X). So, the tape will look like xx.xx$xxxXxxxxEeeeeeZzzzzzSsssssCccccccccc and the header will be at the C delimiter.

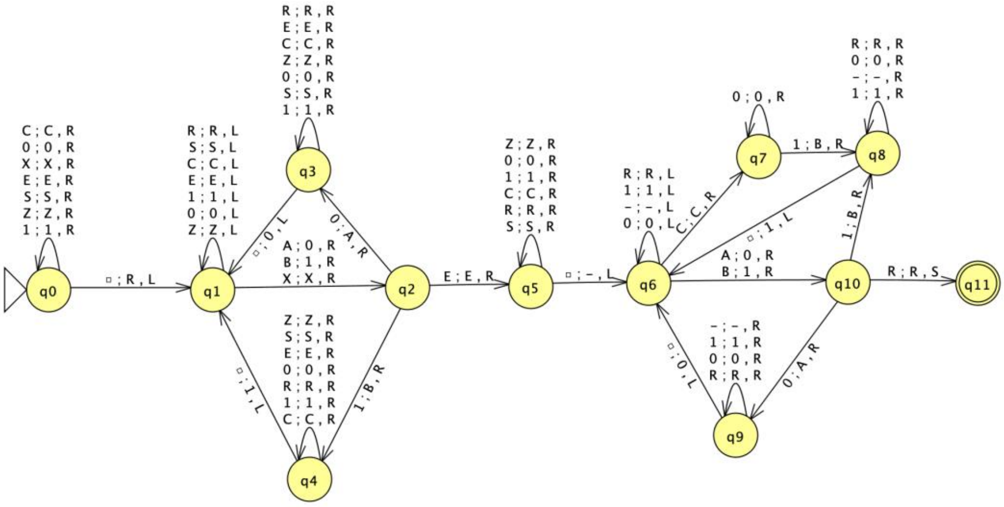
adaptMultDec.jff does not change the input tape if X has no decimal digits (states q0 to q3). If X has a decimal dot, least significant bit of X is found (state q4). For each digit of X until the decimal dot is found (starting from the least significant), the digit is marked on X and the least significant bit of cc…cc is deleted (states q5 to q8). Once the decimal dot is found, header is moved back to delimiter C (states q2 and q3).



2.12. copyXminusSquare.jff

This Turing Machine prepares the tape to obtain X’- $(Z_i + \varepsilon_i)^2$. When this block is accessed from final.jff, the tape is xx.xx$xxxXxxxxEeeeeeZzzzzzSsssssCcccccccccc and the header is at the C delimiter. Once the block is complete, the tape will be like xx.xx$xxxXxxxxEeeeeeZzzzzzSsssssCcccccccccRxxxx-ccccccccc with the header at delimiter R.

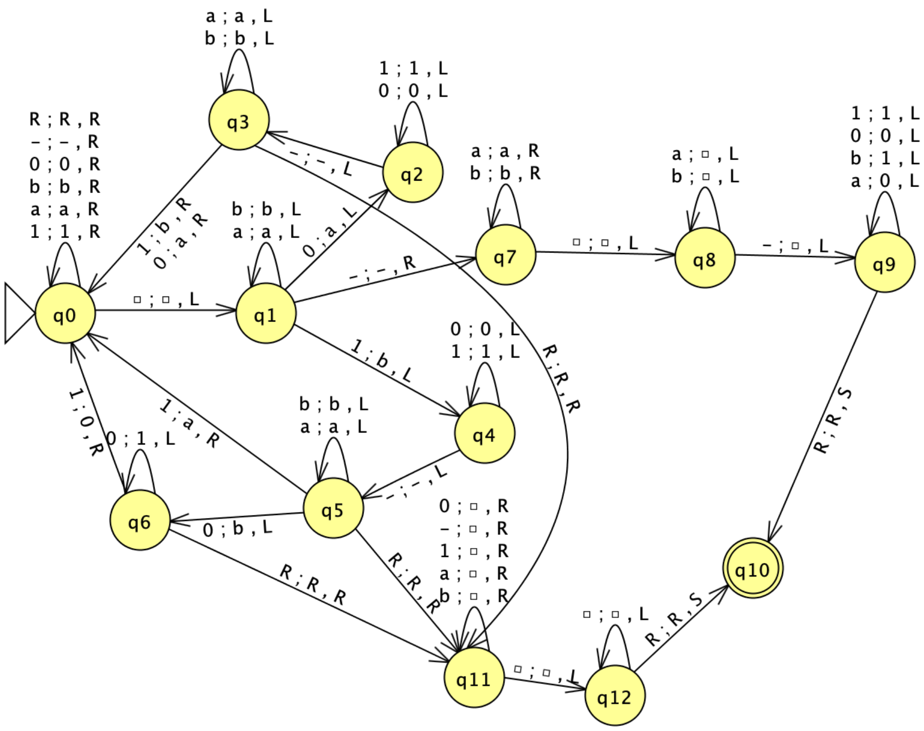
copyXminusSquare.jff first adds the R delimiter on the right of the tape (state q0). Then, it copies the value of X’ at the end of the tape using a simple copier similar to others explained (states q1 to q4). Once X’ is copied, it adds the minus on the right (state q5) and copies the value of C without 0s on the left (state q7) just like the value of X (states q6 to q10). At the end, header is just moved back to delimiter R (state q11).



2.13. sub.jff

This Turing Machine, which is an adapted version of the one provided, calculates the subtraction of two binary numbers. When this block is accessed from final.jff, the tape is xx.xx$xxxXxxxxEeeeeeZzzzzzSsssssCcccccccccRxxxx-ccccccc and the header is at the digit on the right of the R delimiter. When the block is finished, the tape contains xx.xx$xxxXxxxxEeeeeeZzzzzzSsssssCcccccccccRrrrr and the header is at the R delimiter. If subtraction is positive, rrrr will represent the result. If result is 0, rrrrr may look like 00000. In the case of the result being negative, rrrr will not contain anything other than blank spaces.

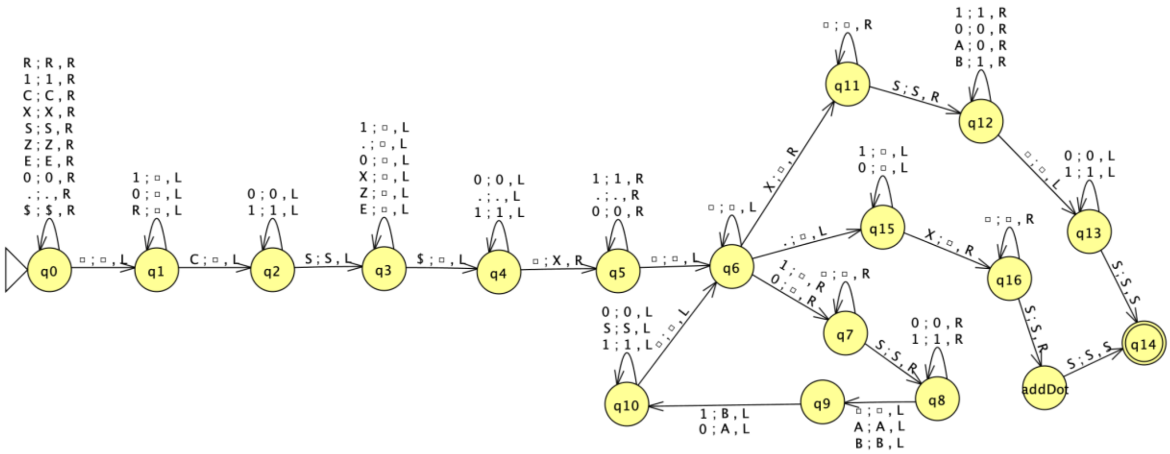
sub.jff is similar to the subtractor provided in class with the only difference that it does not give an error when result is negative. To do this, states q11 and q12 have been added. Similar to the adder, digits already considered re marked and digits are treated differently depending on their values. When subtracting a 0 (states q0 to q3) the digits are only marked. When subtracting a 1 to another 1 (states q0, q1, q4 and q5) machine operates differently than when subtracting to a 0 (states q0, q1, q4 to q6). When complete, header is moved back to R delimiter and digits other than the result removed (states q7 to q10).



2.14. onlySum.jff

This Turing Machine is accessed from final.jff when X'- $(Z_i + \varepsilon_i)^2$ is 0. Therefore, the square root of X' is the sum $Z_i + \varepsilon_i$. This Turing Machine clears the tape and adapts the result to account for the decimal digits of X and show the correct result for the square root of X. When this block is accessed, the tape looks like xx.xx$xxxXxxxxEeeeeeZzzzzzSsssssCcccccccccRrrrr and the header is at the right-most digit. When this block is finished, tape only contains the result in the form x.xx with the correct number of decimal places and the header on the left-most digit.

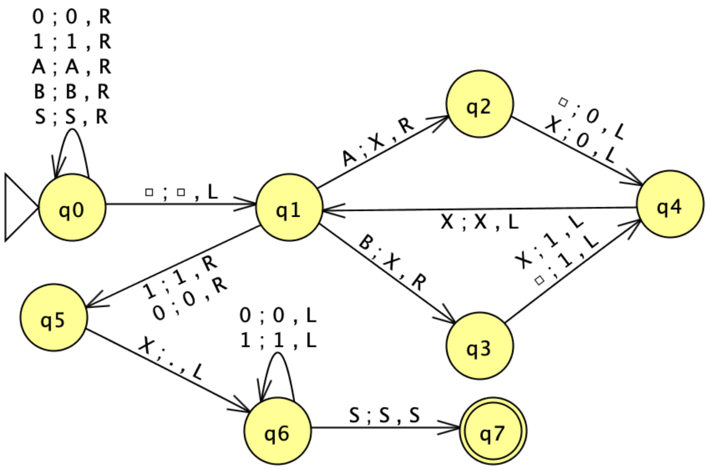
onlySum.jff removes everything on the right of the sum (states q0 and q1). Then, everything between X and the sum is removed (states q2 and q3) and an X is added on the left of the tape (states q4 and q5) so that the tape looks like Xxx.xx□□□□□□□□□□□Ssssss. Then, less significant digits are removed from X and marked in S (states q6 to q10). If header reaches X without finding the decimal dot on X (state q11), X was a whole number without decimal digits and result is sssss. Therefore, marks are deleted from sssss and result is shown on tape (states q11 to 14). If the decimal dot is found, sssss is adapted to insert the decimal dot in the proper place using addDot.jff (states q15, q16 and block addDot.jff).



2.15. addDot.jff

This Turing Machine adds a dot in the indicated place by moving the digits after the dot one position to the right. When this block is accessed from onlySum.jff or onlyZ.jff, the tape looks like […]S101AABA with the header at delimiter S for example. When the block is finished, tape looks like […]S101.0010 with header at S delimiter.

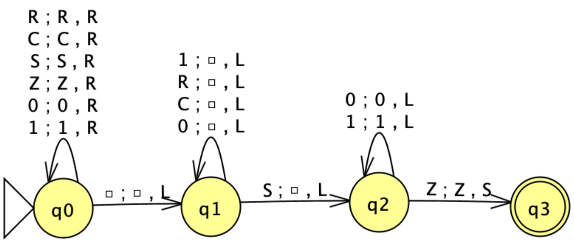
addDot.jff goes to the right most digit of the tape, replaces it by X and copies it at the end of the tape. Then, it finds every element on the left of X and swaps it with X to move each digit one position to the right until all As and Bs have been moved (states q0 to q4). Finally, the X is replaced by the decimal dot and the header moved back to the S delimiter (states q5 to q7).



2.16. constantZ.jff

This Turing Machine removes everything on the right of the Z value in order to start a new iteration. When this block is accessed from final.jff, the tape is xx.xx$xxxXxxxxEeeeeeZzzzzzSsssssCcccccccccR and the header is at the R delimiter. When the block is finished, tape looks like xx.xx$xxxXxxxxEeeeeeZzzzzz and header is at the Z delimiter.

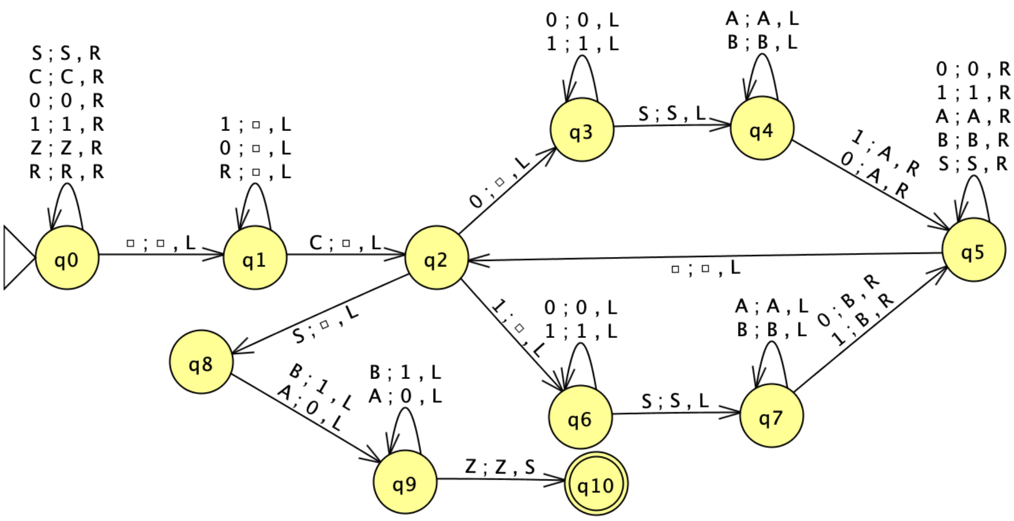
constantZ.jff goes to the right of the tape (state q0) and start removing everything towards the right until Zzzzzz (state q1). Then, the header is moved back to Z delimiter (states q2 and q3).



2.17. updateZ.jff

This Turing Machine removes everything on the right of the Z value and updates the Z value in order to start a new iteration. When this block is accessed from final.jff, the tape is xx.xx$xxxXxxxxEeeeeeZzzzzzSsssssCcccccccccRrrrrrr and the header is at the right-most 1 in rrrrrr. When the block finishes, tape looks like xx.xx$xxxXxxxxEeeeeeZzzzzz with the header at the Z delimiter.

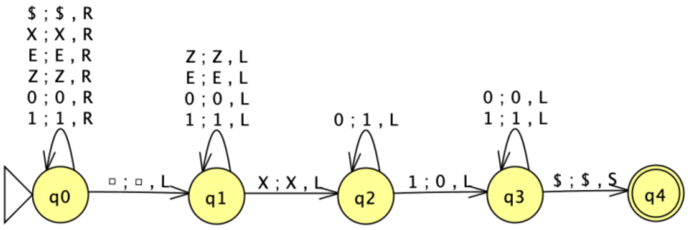
updateZ.jff goes to the right of the tape (state q0) and removes everything until the sum (state q1), which will be the new Z value. Then, the value of the sum is copied in the space reserved for Z. Digits are copied from right to left differently depending on wheter they are a 0 (states q2 to q5) or a 1 (states q2, q5 to q7). Once the value is copied and sum has been deleted, S delimiter is removed and header is moved back to Z delimiter (states q8 to q10).



2.18. decreaseN.jff

This Turing Machine decreases the number of iterations left, N, by one unit after completing the current iteration. When this block is accessed from final.jff, tape contains xx.xx$xxxXxxxxEeeeeZzzzzz with the header at the Z delimiter. When the block is completed, tape contains the same (with a value of N updated) and the header is at the $ delimiter.

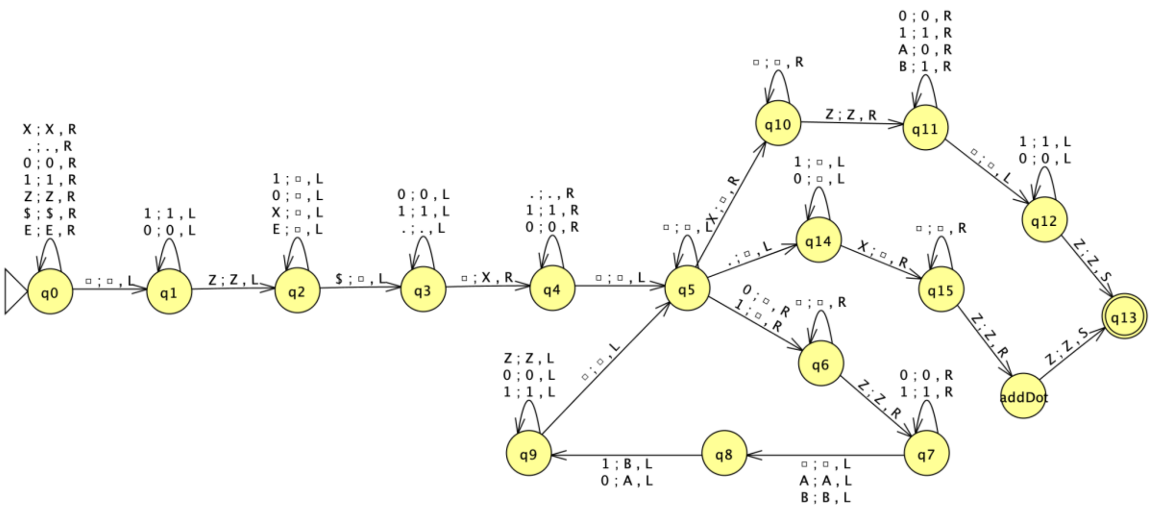
decreaseN.jff goes to the right of the tape (state q0) for testing purposes. Then, it finds the least significant bit of N (state q1) and decreases it by one unit by changing 0s by 1s until a 1 is found (state q2), which is changed for a 0. Finally, header is moved back to the $ delimiter (states q3 and q4).



2.19. onlyZ.jff

This Turing Machine shows the result of the square root of X after completing the given number of iterations. The result after N iterations is approximated by Z, so Z has to be adapted to account for decimal digits and left alone on the tape. When this block is accessed from final.jff tape looks like xx.xx$xxxXxxxxEeeeeZzzzzz with the header at the X delimiter. When the block finishes, tape looks like x.xx with the header on the left-most digit (x.xx being the z value adapted to account for decimal digits).

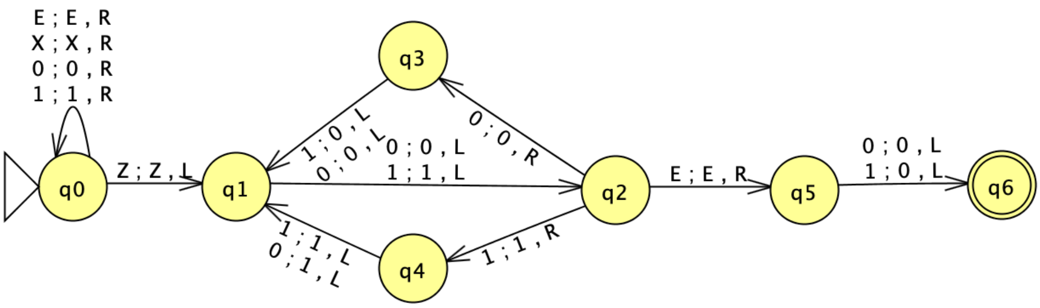
onlyZ.jff is very similar to block onlySum.jff. It goes the left of the Z value (states q0 and q1) for testing purposes. Then, everything between X and the Z value is removed (state q2) and an X is added on the left of the tape (state q3) so that the tape looks like Xxx.xx□□□□□□□□□□□Zzzzz. Then, less significant digits are removed from X and marked in Z (states q5 to q9). If header reaches X delimiter without finding the decimal dot on X (state q10), X was a whole number without decimal digits and result is zzzzz. Therefore, marks are deleted from zzzzz and result is shown on tape (states q10 to 13). If the decimal dot is found, zzzzz is adapted to insert the decimal dot in the proper place using addDot.jff (states q14, q15 and block addDot.jff).



2.20. updateE.jff

This Turing Machine updates the value of ε_i according to the given formula $\varepsilon_{i+1} = \varepsilon_i/2$. Since the number is in binary, the only thing in order to divide a binary number by 2 is to shift all digits one position to the right. Since we want to have a given precision in ε_i, we keep the number of digits constant, shift all digits one position to the right and remove the last one. When this block is accessed from final.jff, the tape contains xx.xx$xxxXxxxxEeeeeeZzzzzz with the header on N value. When the block is finished, tape looks the same with an updated ε value and header at E delimiter.

update.jff first goes to the least significant bit of ε (state q0) and replaces it with the bit on its right (states q1 to q4) until E delimiter is reached. Then, most significant bit is updated to 0 and header moved back to E delimiter (states q5 and q6).



3. IMPROVEMENTS

This project has been really extensive and complex in comparison with the previous laboratory assignments we had done. Each one of us approached the practice differently. Designing two different Turing machines and observing the strengths and weaknesses of each one led to a better approach in order to obtain an overall well-functioning machine. The approach we finally decided to submit had the tape organized like explained in section 1 and shown below:

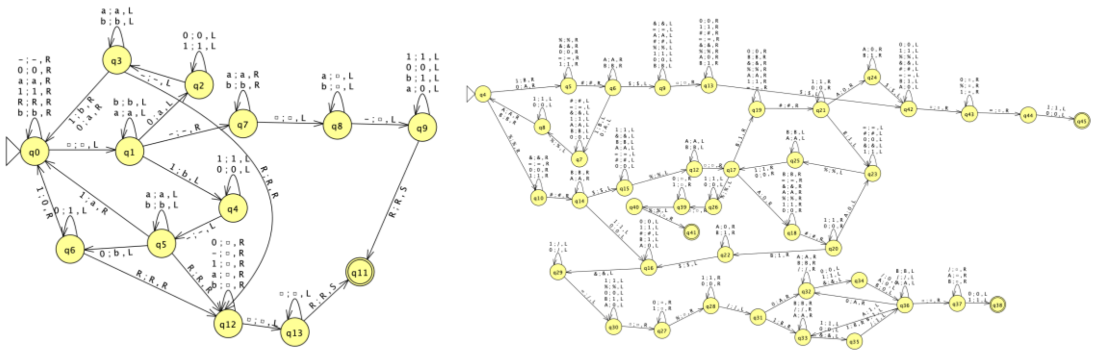
xx.xx	$	xxx	X	xxxx	E	xxxxx	Z	xxxxx	S	xxxxx	C	xxxxxxxxxx	R	xxxxxxxxxx
X		N		X’		ε_i		Z_i		ε_i+Z_i		$(\varepsilon_i+Z_i)^2$		$X'-(\varepsilon_i+Z_i)^2$

While our other approach had the tape organized this other way

xxxx	%	xxx	=	xxxx	&	111	#	xxxxx	$	xxxxx	@	xxxxxxxxxx
$(\varepsilon_i+Z_i)^2$		ε_i+Z_i		Z		D		X		N		ε_i

D being the unary representation of the decimal digits of X

Despite having our second approach a shorter tape and not having to deal with the decimal point through operations, but only at the beginning and the end, it also had some major flaws in comparison with our final machine. For instance, the comparison phase had much greater transitions and states which significantly slowed the overall machine as comparisons have to be done constantly.



Submitted TM on the left, alternative TM approach on the right

These factors along with being the submitted TM better debugged led to our choice. However, it taught us about how could potentially refine our machine. Very likely the tape of our submission could be reduced in size, not storing X (only X’) or not storing $(\varepsilon_i+Z_i)^2$ but only compute it at the specific moment that they are needed and then deleting them from the tape. Moreover, even though we believe that our initial ε_0 is extremely efficient and precise compared to when we first set it equal to X, we believe that there may be a better one which would reduce the number of cycles and therefore improve overall performance.

4. TESTS PERFORMED

Once we thought we had succeeded with the implementation of our machine, we run several tests thinking about different cases, especially the ones that could give us some problems and unexpected errors. These tests helped us to determine with confidence that our implementation works for any given case. We believe we have covered specific and some extreme cases where unexpected behavior could have happened. Testing inputs as numbers with a decimal part equal to 0, inputs with N=0, big and small numbers, numbers with an exact square root… The expected output has been determined based on the algorithm and using the calculator using number in decimal format.

Input	Expected Output	Obtained Output	Success
11.011101$11011	1.110111	1.110111	**Yes**
1.0$1	0.0	0.0	**Yes***
1.0$10	1.0	1.0	**Yes**
1.00$10	1.00	1.00	**Yes**
1$11	1	1	**Yes**
101$0	0	0	**Yes**
1011.00$111	11.01	11.01	**Yes**
10101.11$111	100.10	100.10	**Yes**
1111001$111110001	1011	1011	**Yes**
1111001.00$1111	1011.00	1011.00	**Yes**
100000.0$1111	101.1	101.1	**Yes**
100000.00$0011011	101.10	101.10	**Yes**
1000010001$11111	10111	10111	**Yes**

**Despite what it may look like, the machine is behaving as expected. As we set N=1, the machine in the first loop performs 10*10 and as it is greater than X when truncated, Z remains equal to 0. We end the machine as N is now equal to zero and Z value is shown. On the following test we can see how a greater N obtains the desired solution*.

Also, since we completed the assignment before the lab session we had, we were able to test our Turing Machine with the test suit provided by the professor. Details on those tests are shown just below. We obtain the expected results, but not with the precision of the last digit. This is due to the low number of iterations and the initial configuration of $\varepsilon_i = 2^{k+1}$ where k is the digits of X.

Input	Output	Result
100.1101011$1000	10.0011000	Accept
101.1100001$1000	10.0110000	Accept
110.1100001$1000	10.1001000	Accept
111.1101011$1000	10.1100000	Accept
1001.000000$1000	11.000000	Accept
1010.0011110$1000	11.0010000	Accept
1011.1000111$1000	11.0110000	Accept
1100.1111010$1000	11.1000000	Accept
1110.0111000$1000	11.1100000	Accept
10000.0000000$1000	100.0000000	Accept

	x	raiz(x)	x binario	raiz(x) binario
1	4.84	2.2	100.1101011	10.0011
2	5.76	2.4	101.1100001	10.0110
3	6.76	2.6	110.1100001	10.1001
4	7.84	2.8	111.1101011	10.1100
5	9	3	1001	11
6	10.24	3.2	1010.0011110	11.0011
7	11.56	3.4	1011.1000111	11.0110
8	12.96	3.6	1100.1111010	11.1001
9	14.44	3.8	1110.0111000	11.1100
10	16	4	10000	100

Tests Explained:

[] Result given by algorithm without adapting decimals because X'- $(E+Z)^2$ = 0 or N=0

[] Value of Z has been updated because X'- $(E+Z)^2 > 0$

[] Result given without adapting decimals and Z has been updated

X	X'	N	E	Z	E+Z	Fixed $(E+Z)^2$	X'- $(E+Z)^2$
11.011101	11011101	11011	100000000	000000000	100000000	10000000000	(-)
		11010	010000000	000000000	010000000	100000000	(-)
		11001	001000000	000000000	001000000	1000000	(+)
		11000	000100000	001000000	001100000	10010000	(+)
		10111	000010000	001100000	001110000	11000100	(+)
		10110	000001000	001110000	001111000	11100001	(-)
		10101	000000100	001110000	001110100	11010010	(+)
		10100	000000010	001110100	001110110	11011001	(+)
		10011	000000001	001110110	001110111	11011101	0
		10010	000000000	--	--	--	--
		. . . .					
		00000	000000000	--	--	--	--
001110111 is adapted to 1.110111							

X	X'	N	E	Z	E+Z	Fixed $(E+Z)^2$	X'- $(E+Z)^2$
1.0	10	1	100	000	100	1000	(-)
		0	0100	0000	--	--	--
0000 is adapted to 0.0							

X	X'	N	E	Z	E+Z	Fixed $(E+Z)^2$	X'- $(E+Z)^2$
1.0	10	10	100	000	100	1000	(-)
		01	010	000	010	10	0
		00	001	--	--	--	--
010 is adapted to 1.0							

X	X'	N	E	Z	E+Z	Fixed $(E+Z)^2$	X'- $(E+Z)^2$
1.00	100	10	1000	0000	1000	10000	(-)
		01	0100	0000	0100	100	0
		00	0010	--	--	--	--
0100 is adapted to 1.00							

X	X’	N	E	Z	E+Z	Fixed $(E+Z)^2$	X’- $(E+Z)^2$
1	1	11	10	00	10	100	(-)
		10	01	00	01	1	0
		01	--	--	--	--	--
		00	--	--	--	--	--
01 is adapted to 1							

X	X’	N	E	Z	E+Z	Fixed $(E+Z)^2$	X’- $(E+Z)^2$
101	101	0	1000	0000	--	--	--
0000 is adapted to 0							

X	X’	N	E	Z	E+Z	Fixed $(E+Z)^2$	X’- $(E+Z)^2$
1011.00	101100	111	1000000	0000000	1000000	10000000000	(-)
		110	0100000	0000000	0100000	100000000	(-)
		101	0010000	0000000	0010000	1000000	(-)
		100	0001000	0000000	0001000	10000	(+)
		011	0000100	0001000	0001100	100100	(+)
		010	0000010	0001100	0001110	1110001	(-)
		001	0000001	0001100	0001101	101010	(+)
		000	0000000	0001101	--	--	--
0001100 is adapted to 11.01							

X	X’	N	E	Z	E+Z	Fixed $(E+Z)^2$	X’- $(E+Z)^2$
10101.11	1010111	111	10000000	00000000	10000000	1000000000000	(-)
		110	01000000	00000000	01000000	10000000000	(-)
		101	00100000	00000000	00100000	100000000	(-)
		100	00010000	00000000	00010000	1000000	(+)
		011	00001000	00010000	00011000	10010000	(-)
		010	00000100	00010000	00010100	1100100	(-)
		001	00000010	00010000	00010010	1010001	(+)
		000	00000001	00010010	--	--	--
00010010 is adapted to 100.10							

X	X’	N	E	Z	E+Z	Fixed $(E+Z)^2$	X’-$(E+Z)^2$
1111001	1111001	111110001	10000000	00000000	10000000	100000000000000	(-)
		111110000	01000000	00000000	01000000	1000000000000	(-)
		111101111	00100000	00000000	00100000	10000000000	(-)
		111101110	00010000	00000000	00010000	100000000	(-)
		111101101	00001000	00000000	00001000	1000000	(+)
		111101100	00000100	00001000	00001100	10010000	(-)
		111101011	00000010	00001000	00001010	1100100	(+)
		111101010	00000001	00001010	00001011	1111001	0
		111101001	00000000	--	--	--	--
		. . . .					
		000000000	00000000	--	--	--	--
00001011 is adapted to 1011							

X	X’	N	E	Z	E+Z	Fixed $(E+Z)^2$	X’-$(E+Z)^2$
1111001.00	111100100	1111	1000000000	0000000000	1000000000	10000000000000000	(-)
		1110	0100000000	0000000000	0100000000	100000000000000	(-)
		1101	0010000000	0000000000	0010000000	1000000000000	(-)
		1100	0001000000	0000000000	0001000000	10000000000	(-)
		1011	0000100000	0000000000	0000100000	100000000	(+)
		1010	0000010000	0000100000	0000110000	1001000000	(-)
		1001	0000001000	0000100000	0000101000	110010000	(+)
		1000	0000000100	0000101000	0000101100	111100100	0
		0111	0000000010	--	--	--	--
		. . . .					
		0000	00000000	--	--	--	--
0000101100 is adapted to 1011.00							

X	X’	N	E	Z	E+Z	Fixed $(E+Z)^2$	X’- $(E+Z)^2$
100000.0	1000000	1111	10000000	00000000	10000000	10000000000000	(-)
		1110	01000000	00000000	01000000	100000000000	(-)
		1101	00100000	00000000	00100000	1000000000	(-)
		1100	00010000	00000000	00010000	10000000	(-)
		1011	00001000	00000000	00001000	100000	(+)
		1010	00000100	00001000	00001100	1001000	(-)
		1001	00000010	00001000	00001010	110010	(+)
		1000	00000001	00001010	00001011	111100	(+)
		0111	00000000	00001011	00001011	111100	(+)
		. . . .					
		0000	00000000	00001011	--	--	--
00001011 is adapted to 101.1							

X	X’	N	E	Z	E+Z	Fixed $(E+Z)^2$	X’- $(E+Z)^2$
100000.00	10000000	0011011	100000000	000000000	100000000	1000000000000000	(-)
		0011010	010000000	000000000	010000000	1000000000000	(-)
		0011001	001000000	000000000	001000000	10000000000	(-)
		0011000	000100000	000000000	000100000	100000000	(-)
		0010111	000010000	000000000	000010000	1000000	(+)
		0010110	000001000	000010000	000011000	10010000	(-)
		0010101	000000100	000010000	000010100	1100100	(+)
		0010100	000000010	000010100	000010110	1111001	(+)
		0010011	000000001	000010110	000010111	10000100	(-)
		. . . .					
		0000	00000000	000010110	--	--	--
000010110 is adapted to 101.10							

X	X’	N	E	Z	E+Z	Fixed $(E+Z)^2$	X’- $(E+Z)^2$
1000010001	1000010001	11111	10000000000	00000000000	10000000000	$1*2^{20}$	(-)
		11110	01000000000	00000000000	01000000000	$1*2^{18}$	(-)
		11101	00100000000	00000000000	00100000000	$1*2^{16}$	(-)
		11100	00010000000	00000000000	00010000000	$1*2^{14}$	(-)
		11011	00001000000	00000000000	00001000000	$1*2^{12}$	(-)
		11010	00000100000	00000000000	00000100000	10000000000	(-)
		11001	00000010000	00000000000	00000010000	100000000	(+)
		11000	00000001000	00000010000	00000011000	1001000000	(-)
		10111	00000000100	00000010000	00000010100	110010000	(+)
		10110	00000000010	00000010100	00000010110	111100100	(+)
		10101	00000000001	00000010110	00000010111	1000010001	(-)
		10100	00000000000	--	--	--	--
		. . . .					
		00000	00000000000	--	--	--	--
00000010111 is adapted to 10111							

5. CONCLUSSION

This project took much more time and effort than expected. We believe the way this practice ramped up difficulty was challenging in comparison with how previous practices were and we believe an intermediate step (as an extra practice or some exercises done at class) would have been really helpful. Nevertheless, we believe to have improve our skills regarding Turing machines deeply from when we began the practice. Developing two functioning Turing machines for a later comparison was highly time consuming but led to what we believe to be a good machine that works as expected on all the cases we have been able to test. We also learnt about building blocks, how they work and their potential when building a more complex Turing machine as this one was. Lastly, we improved our knowledge on JFLAP software and how to “debug” a machine when it is not working as expected to identify and correct any mistake. We are satisfied with our Turing Machine but would have liked more time to be able to perform some improvements regarding efficiency.