CS 361 Meeting 19 — 4/13/20

Announcements

1. Stay healthy!

Your First Turing Machine?

(Click for video)

1. We have arrived.

2. It is time to talk about Turing machines, the machine model I mentioned on the first day of class, that will give us a "realistic" model of a computer.

3. As we did with DFAs and PDAs, we will start with some examples of diagrams of Turing machines meant to familiarize you with their operation informally. Then, we will give formal definitions for Turing machines and the languages they recognize in the next class.

4. Like DFAs and PDA, a Turing machine is based on a finite set of states, a finite alphabet, and a finite transition function.

5. Unlike DFAs, but like PDAs, a Turing machine has an infinite memory in which it can store symbols from its alphabet.

6. Unlike a PDA this memory is not separate from its input, but is viewed as an extension of the area on which the input is written.

7. With a Turing machine, we imagine that the finite input is written on an infinite tape and we allow the machine both to read and write symbols on the tape both in the original area used for the input and anywhere after the original input.

8. Turing machines don't have "Final states" in the way that DFAs and PDAs do.

9. Instead, a Turing machine has one accept state and one reject state. As soon as it enters one of these two states, the machine stops its processing and either accepts or rejects its input.

   - Note that if the machine never enters one of these states it will never stop.
   - Notationally, it is common to omit the reject state and to assume that if no transition out of a state on some input exists, then the machine goes to its reject state.

10. At each point in the computation a Turing machine is in a particular state of its state diagram and is looking at the position in a particular cell of its tape.

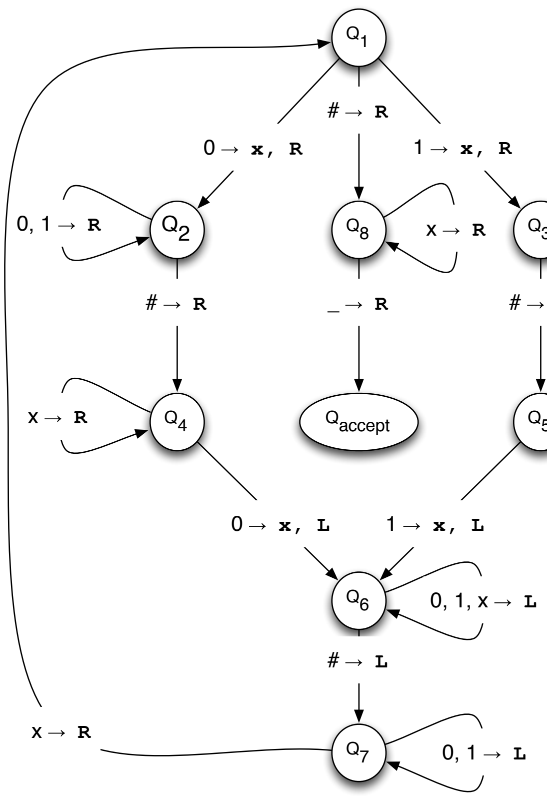
11. With each transition, a Turing machine changes in three ways: a) it can move to a new state, b) it can change the symbol in the cell at which it was positioned, c) it can move to the following or preceding position on its tape.

    - One quirky feature of Sipser's version of the Turing machine formalism is that if the machine ever tries to move to the position preceding the first cell on the tape, it just remains at the first cell.

12. To make this discussion more concrete, consider the machine shown below.

    - In this diagram, Q_1 is the start state, Q_{accept} is the accept state and the reject state is not shown. It is assumed to be the destination of any transition that is missing from the diagram.
    - Each transition is labeled with a string of the form "$s_1, \dots s_n \rightarrow w, D$" or "$s_1, \dots s_n \rightarrow D$".
      - The symbols on the left of the arrow indicating that the labeled transition will only be taken if the symbol at the current position on the tape matches one of the symbols in the list.
      - The "D" will be replaced by L (for move left) or R (for move right).
      - The "w" indicates a symbol to replace the existing contents of the cell at the current position on the tape. If no such symbol is provided, the cell remains unchanged.

Click here to view the slides for this class



13. To better understand how this machine functions, let's examine how it would behave if its input tape initially contained the string "01#01" followed by blank tape.

    - The machine starts looking at the leading 0 in the input in state Q_1.
    - From Q_1 it follows the transition labeled "$0 \to x, R$" to Q_2 since it sees a 0. With this transition, the leading 0 is replaced with an "x". The machine is effectively crossing out the 0 so it will be easily distinguishable as a processed cell if the machine returns to this position later.
    - In state Q_2, the machine will make repeated move right by one step as long as the tape cells contain 0's or 1's. It will move to Q_4 as soon as it reaches the # that is expected in the middle.
    - State Q_4 has a transition on "x", that will not be used on this first pass. Instead, finding a 0 after the #, the machine will cross the 0 out by writing an "x" and move to state Q_6.
    - The loop on Q_6 is designed to let the machine move back until it reaches the #. On this first pass, this happens immediately without using the loop bring the machine to Q_7.
    - The loop on Q_7 is designed to let the machine move over any 0's and 1's to the left of the # until it gets back to the last symbol it replaced by an "x". This will take just one step and then the machine will go to Q_1 while moving over the 1 in the second tape cell.
    - The machine will then pretty much repeat this process to cross off the 1's on either side of the # using states Q_3, Q_5, Q_6 and Q_7 this time. The difference is that while moving through the states on the left, the machine is remembering that it crossed off a 0 as it left state Q_1 while it will only be in the states on thr right if it crossed off a 1.
    - The next time we reach Q_1, the machine will already be on the # and therefore will move to state Q_8. This state makes sure that all of the symbols on the right have been crossed off (indicating that they all matched symbols on the right).
    - If state Q_8 only encounters x's until it reaches the first blank, then the input is accepted.

14. While I love state diagrams, Turing machines quickly get large and complicated, so it is often useful to give a less formal description of their operation. The machine above, for example can be described as follows.

    (a) If the symbol under the read head is a #, scan to the right and if you only find x's until you reach blank tape accept. Otherwise,

    (b) Remember the character you start at and replace it with an x.

    (c) Scan to the right until you find a #.

(d) Continue scanning to the right skipping any x's.

(e) If the first symbol after the x's does not match what you saw in state b, reject.

(f) If the symbol after the x's matches, replace it with an x.

(g) Scan to the left passing all symbols until you get back to the #.

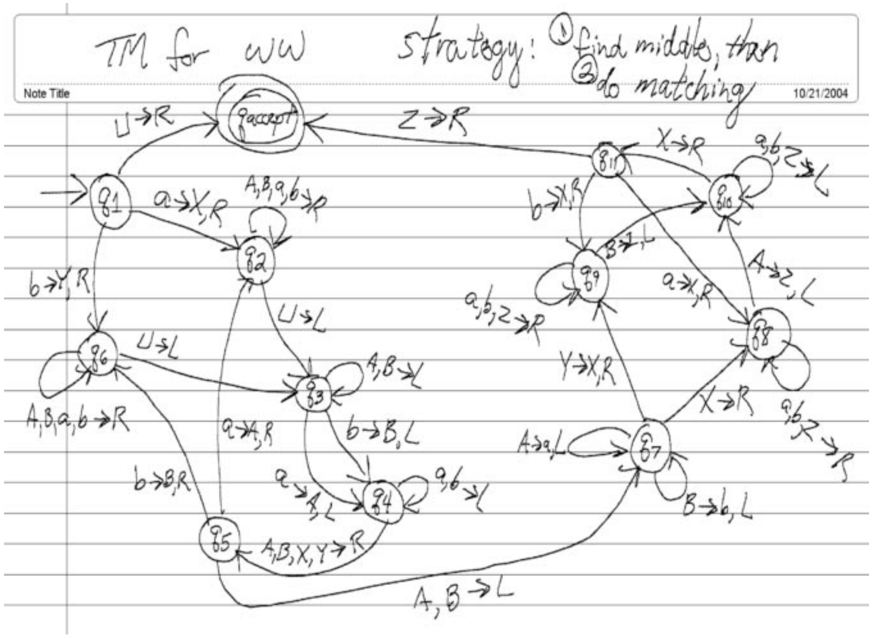
(h) Keep scanning to the left to find the first x.

(i) Move to the right and return to state a.

Another Simple, Similar Turing Machine

(Click for video)

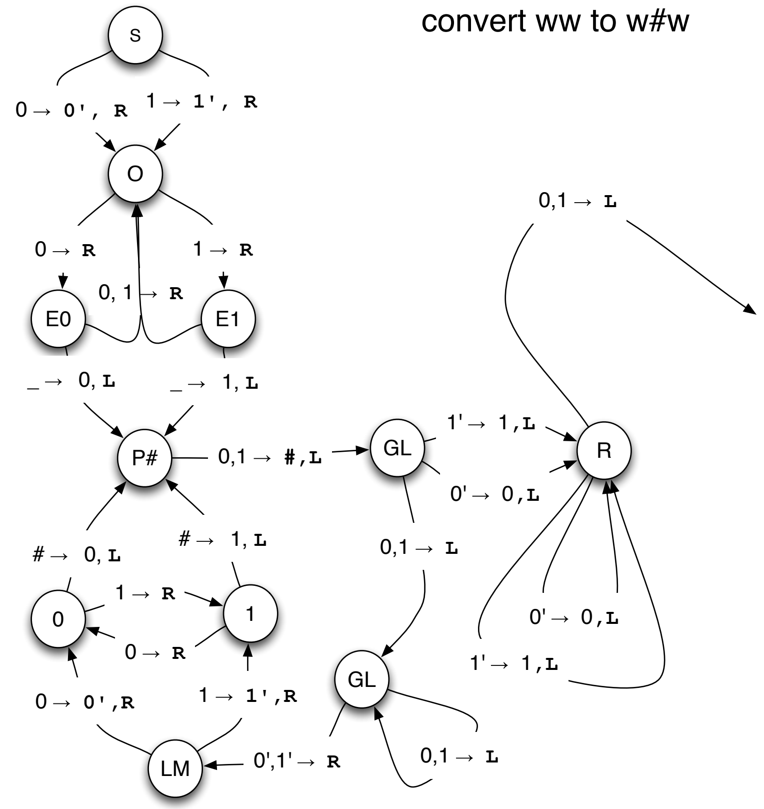
1. I found a hand-drawn Turing machine using Google images that recognizes a language quite similar to the language Sipser's machine recognizes.



- Instead of $w\#w$, this machine recognizes ww.
- Instead of using the binary alphabet $\{0, 1\}$ as Sipser did, this machine uses $\{a, b\}$. Of course, the shapes of the symbols don't really matter.

2. I would like to illustrate another way we could build a machine that recognizes ww, this time over the alphabet $\{0, 1\}$. The idea is to re-use Sipser's machine by doing a little pre-processing.

3. Consider the machine:



4. This machine tries to interpret its input as a string that might be of the form ww and convert it to a string of the form $w\#w$ by inserting a # at the midpoint of its input.

- It first marks and remembers the left-most input character.
    - Since our tape alphabet is unlimited, we can assume the existence of primed, double-primed, hatted, etc. copies of every symbol in our input alphabet.
- It then scans to the end of the tape remembering whether it has seen an even or odd number of symbols, and what the last symbol in an even location was.
- When it hits the end of the tape, it rejects if it has seen an odd number of symbols. If it has seen an even number, it write the last symbol seen over the first blank and then backs up to the original copy of the symbol.
- Next it puts a # in place of what had been the last symbol.
- Now, it enters its main loop. It repeats this as long as there are more unmarked symbols to the left of the position where it has most recently placed the #.
    - It scans left to find the rightmost marked symbol.
    - It then remembers and marks the unmarked symbol just to the right of the rightmost marked symbol and begins a scan to the right.
    - It scan until it hits the # remembering the last symbol it saw
    - When it finds the #, it interchanges the # with the symbol that appeared just before the #.
- When this loop terminates, the # has been positioned correctly, but all the symbols to its left are marked. The final loop makes a pass to the left unmarking all these symbols. It stops by taking advantage that in Sipser's version of TMs, if you attempt to move off the left end of the tape, you just remain on the left-most symbol.

5. Such a machine is called a *transducer*. Its purpose is not really to accept or reject strings. Its purpose is to transform strings. If the arrow leading from the right side is connect to the start state of Sipser's machine, the combined machine will only accepts strings of the form *ww*.

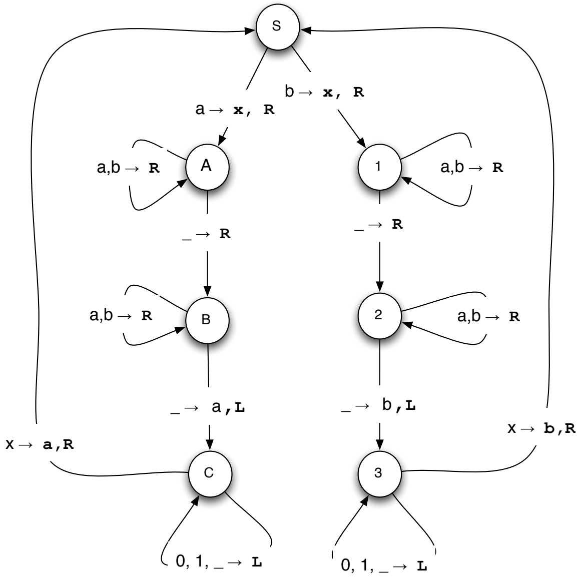
6. This illustrates a useful way to think about designing and describing complex Turing machines. Just as we break real programs up into functions or methods, we can break Turing machines up into sub-modules to perform certain transformations or checks on sub-parts of the input.

Transducers as Components

(Click for video)

1. There are many other examples of Turing Machines that are interesting as transducers. In some sense, they are not complete Turing machines. They don't have accept and reject states so they don't recognize any languages. They can be seen, however, as useful parts when trying to design a Turing Machine that does recognize some language.

2. Here is an interesting example of such a module:



Given any input string of as and bs, this collection of states acts like a Xerox machine. It makes a second copy of the input following the first separated from the first by a space.

   - The states named A, B and C are used to copy an a from the original input to the first blank after the incomplete copy.
   - The states numbered 1, 2 and 3 are used to copy a b.
   - Each set of states first marks the symbol being copied by replacing it with an x and then scans until it finds a blank.
   - Next the machine skips over any copies of letters it has already copied until another blank is found.
   - The second blank is replace by the copied character.
   - Finally, the machine scans back to find the marked symbol that was just copied, unmarks it and repeats the process with the next symbol.

3. With a bit of imagination, you can think of this machine as a primitive assignment statement. After all, "x = y;" makes a copy of y's value.

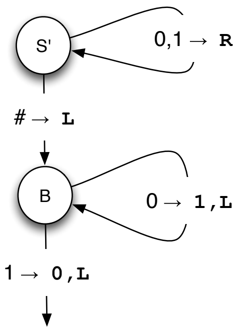
4. This may seem to require a lot of imagination, but it is an important idea to understand to appreciate the power of Turing Machines.

   - A PDA or DFA only gets to look at information once (on a single pass over its input or as it pops it off the stack).
   - A TM can perform one step of computation in one area of the tape (possibly over the input or possibly over some information it copied to another portion of the tape) and then move on to another step.
   - By composing and even repeating such steps, we can implement algorithms in a style much more like traditional programs than PDAs or DFAs.

Counting Down

(Click for video)

1. As another example of a fragment of a Turing machine that might be a useful component in a bigger structure, consider the machine:



   - This machine is simple enough that you should be able to figure out its function on your own?
   - Did you? If not, it interprets its input as a string of the form $n\#\ldots$ where n is the representation of some positive number represented in binary notation, and it transforms its input to be $n-1\#\ldots$.