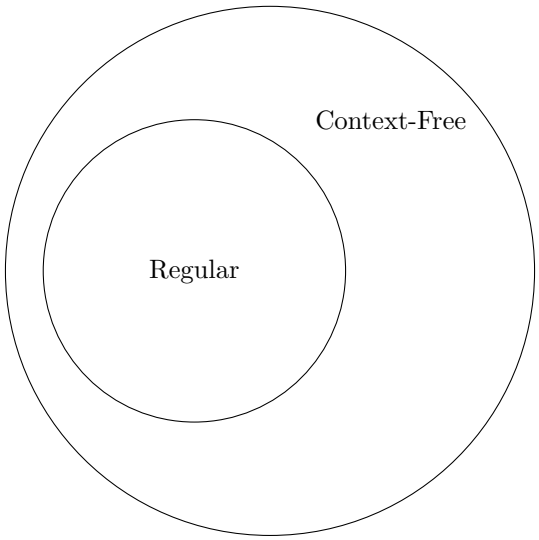
CSCI 341–Fall 2024: Lecture Notes
Set 12: Turing Machines

Edward Talmage

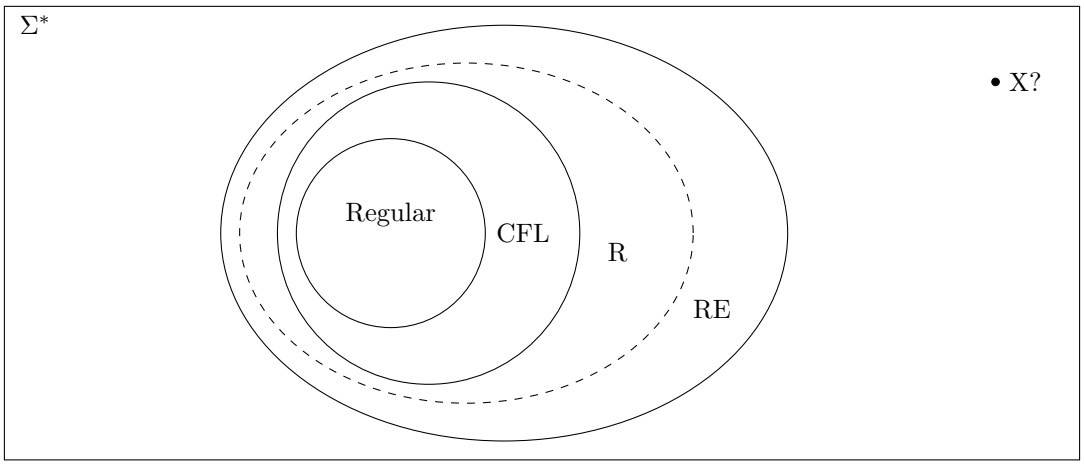
October 25, 2024

1 Setup and Review

So far, we have considered two classes of languages, viewing the world like this:



Now, we will extend one more level, which will be our last in this trajectory:



To define these new classes of languages, we need a new machine. The *Turing Machine* will actually define both of our new classes. A Turing Machine (TM) is a finite state automaton, together with an unlimited, random-access memory. Turing Recognizable (or just Recognizable) languages are those recognized by a TM. Decidable languages are those for which there is a TM that will always halt, either accepting or rejecting (halting to reject is a new feature). The difference here, which is also unlike anything we have seen before, is that TMs can run forever. This is

not acceptance, so a string which causes a TM to run forever is not in its language, but it is not as nice as termination and rejecting the input.

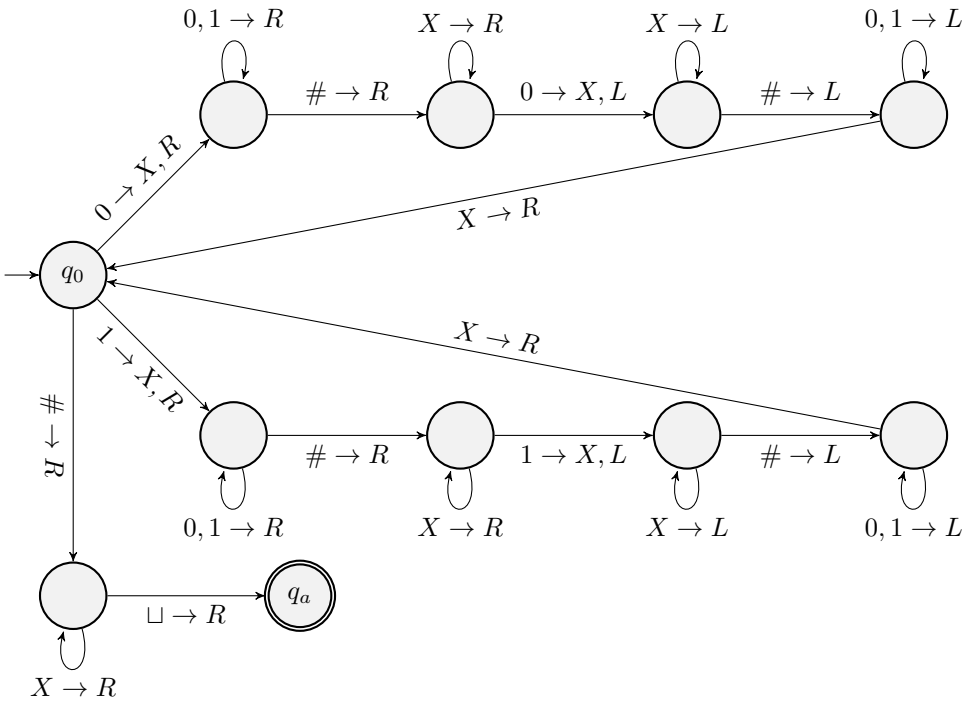
The big question now is whether there are languages outside the class Recognizable. The Church-Turing Thesis, which we will come back to in more detail, says that anything that can be computed (for any reasonable definition of computation), can be computed by a Turing Machine. Specifically, Church and Turing proved that the theory of TMs is equally expressive as lambda calculus. The result has been extended to show that the theory of general recursive functions is also equivalent, and the thesis remains that all other formulations of computation are equivalent. So, then, if TMs completely represent computation, are there languages which cannot be computed? Spoiler: yes, there are.

2 Definition

Definition 1. A *Turing Machine* is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where

- Q is a finite state set
- Σ is a finite input alphabet, which does not contain the symbol $\sqcup$
- Γ is a finite tape (memory) alphabet, with $\sqcup \in \Gamma$ ("empty cell") and $\Sigma \subseteq \Gamma$.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function, with L, R denoting moving left or right on the tape, respectively
- $q_0 \in Q$ is the start state
- $q_{accept} \in Q$ is the (unique) accept state
- $q_{reject} \in Q$ is the (unique) reject state

Example: The following is a TM that recognizes $\{w\#w \mid w \in (0 \cup 1)^*\}$.

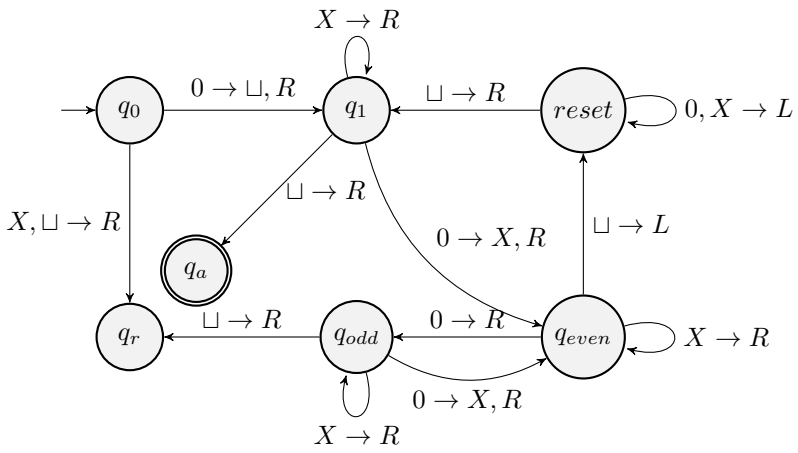


Note that we do not draw all, or in this case any, transitions to q_{reject}. Since TMs are deterministic, there must be an edge from every state on every possible tape symbol, which can be a **lot** of edges. For clarity, any edges not drawn are considered to go to q_{reject}, where the TM immediately halts and rejects the input.

Exercise: Informally (in words) describe a TM recognizing $\{0^{2^n} \mid n \geq 0\}$.

1. Walk right, marking off every second 0.
2. If in step 1, there was only one 0, accept.
3. If in step 1, there was an odd number (> 1) of 0's, reject.
4. Return to left end of tape, repeat from step 1.

Exercise: Draw the corresponding TM



Running a TM: A TM's memory is a one-directionally infinite "tape" (as in, reel-to-reel magnetic tape) or array of memory locations, each of which holds exactly one element of Γ. The tape is read by a "read/write head" (also terminology from magnetic tape storage), which is always at one cell of the tape, starting at the end, which by convention is the left end. When a TM starts, the input $w = w_1 \cdots w_n$ is in the first n cells of the tape, and all other cells contain $\sqcup$. This is why δ no longer takes both an input character and a memory character, as in a PDA. Because the input starts in memory, we only need to input a symbol from memory to determine the next step. It is worth noting that starting with the input in memory is not a significant change, as we could just have our TM first read the input, saving it in memory, then continue. Since we have unrestricted access to memory, we can then return to the beginning of the input and proceed as if the input had been there all along.

To represent the current overall status of a Turing Machine, we introduce a new, concise notation. Observe that we can describe the current status of a TM with a state from the state set, the contents of the tape, and the location of the read/write head. We write this as uqv, where $q \in Q$ is the current state, uv is the contents of memory, and the tape head is at the first character of v. We can then describe transitions as follows:

A configuration $C = uapbv$, where $u, v \in \Gamma^*$, $a, b \in Gamma$, $p \in Q$, yields (denoted $\Rightarrow$) the following:

$$uapbv \Rightarrow \begin{cases} uacqv & \delta(p, b) = (q, c, R) \\ uqacv & \delta(p, b) = (q, c, L) \end{cases}$$

Or, if the head is at the left end of the tape, we have a special case, as the head cannot move left, but stays in the same place if told to move left. Be cautious with this, as the TM does not indicate in any way that this has happened. Thus, $pbv \Rightarrow qcv$ if $\delta(p, b) = (q, c, L)$.

We can now define the rest of the terminology of a Turing Machine:

- A TM *accepts* string w if there is a sequence of configuration $C_0, C_1, \ldots, C_n$ s.t. $C_0 = q_0 w$, C_n is an accept configuration (contains q_{accept}), and $\forall 0 \leq i \leq n-1, C_i \Rightarrow C_{i+1}$.
- TMs *halt* when they reach q_{accept} or q_{reject}, *accepting* or *rejecting*, respectively. If they never reach these states, they *loop*, or run forever.
- The language *recognized* by a TM M is $L(M) := \{w \in \Sigma^* \mid M \text{ accepts } w\}$.
- A language L is *Recognizable* (or *Recursively Enumerable*) if there is some TM M s.t. $L(M) = L$.
- A TM M is a *decider* if $\forall w \in \Sigma^*$, M halts on w.
- A language is *Decidable* (or *Recursive*) if a decider recognizes it.

3 More Examples

Exercise: Informally describe how to build a TM recognizing

$$L = \{a^i b^j c^k \mid i * j = k, i, j, k \geq 1\}$$

1. Scan left to right to ensure w has the form $a^+b^+c^+$. We could do this with just a DFA, since that is a regular expression.
2. Return to left end of tape. Need to mark the end to do this: replace first a with $\hat{a}$ (just extended Γ).
3. Cross off an a, then scan right to the first b. From here, zig-zag, crossing off pairs of b's and c's, until there are no more b's. If there are no more c's while there are still b's, reject.
4. If there are more a's, unmark all the b's, repeat step 3.
5. When all a's are gone, if all c's are gone, accept. Otherwise reject.

Exercise: Informally describe how to build a TM recognizing

$$L = \{\#x_1\#x_2\#\ldots\#x_n \mid x_i \in \{0,1\}^*, i \neq j \Rightarrow x_i \neq x_j\}$$

1. Mark first cell. Reject if not #.
2. Walk across x_1. Mark second #. Accept if there is none ($n = 1$).
3. Zig-zag to compare strings immediately after two marked #'s. If they are equal, reject.
4. Move the second mark right to the next #. If there is not another # before you see $\sqcup$, move the first mark right to the next #, set the second mark to the first # after that, accept if none such exists.
5. Goto step 3.

Exercise: Describe a TM to recognize the language of strings with equal numbers of 0's and 1's.

4 Describing TMs

Formally, we need to give a 7-tuple or state diagram with transitions to describe a Turing Machine. As you can imagine, those very quickly become unmanageable. Instead, we will typically give a description like the above that describes the behavior of how the head interacts with the tape, leaving out details of the state machine. This is called an *implementation-level description*.

The trick to a proper implementation-level description is to make sure that a DFA can control the head in the manner you describe. Primarily, you must be very careful about what you are remembering and how, as DFAs only have finite working memory (states). Later, we can move to a high-level description, which is just an algorithm description (pseudocode), without machine and tape details.