Lecture 12: Computability and Turing Machines

9 March 2010

> The Electric Monk was a labor-saving device, like a dishwasher or a video recorder. Dishwashers washed tedious dishes for you, thus saving you the bother of washing them yourself, video recorders watched tedious television for you, thus saving you the bother of looking at it yourself; Electric Monks believed things for you, thus saving you what was becoming an increasingly onerous task, that of believing all the things the world expected you to believe.
> – Dirk Gently's Holistic Detective Agency, Douglas Adams.

This lecture covers the beginning of section 3.1 from Sipser.

1 Computability

For the alphabet

$$\Sigma = \{0, 1, \ldots, 9, +, -\},$$

consider the language

$$L = \left\{ a_n a_{n-1} \ldots a_0 + b_m b_{m-1} \ldots b_0 = c_r c_{r-1} \ldots c_0 \;\middle|\; \begin{array}{l} a_i, b_j, c_k \in [0,9] \text{ and} \\ \langle a_n a_{n-1} \ldots a_0 \rangle \\ \quad + \quad \langle b_m b_{m-1} \ldots b_0 \rangle \\ \qquad = \langle c_r c_{r-1} \ldots c_0 \rangle \end{array} \right\},$$

where $\langle a_n a_{n-1} \ldots a_0 \rangle = \sum_{i=0}^{n} a_i \cdot 10^i$ is the number represented in base ten by the string $a_n a_{n-1} \ldots a_0$. We are interested in the question of whether or not a given string belongs to this language. This is an example of a decision problem (where the output is either `yes` or `no`), which is easy in this specific case, but clearly too hard for a PDA to solve it[1].

Usually, we are interested in algorithms that compute something for their input and output the results. For example, given the strings $a_n a_{n-1} \ldots a_0$ and $b_m b_{m-1} \ldots b_0$ (i.e., two numbers) we want to compute the string representing their sum.

Here is another example for such a decision algorithm: Given a quadratic equation $ax^2 + bx + c = 0$, we would like to find the ***roots*** of this equation. Namely, two numbers r_1, r_2 such that $ax^2 + bx + c = a(x - r_1)(x - r_2) = 0$. Thus, given numbers a, b and c, the algorithm should output the numbers r_1 and r_2.

To see how subtle this innocent question can be, consider the question of computing the roots of a polynomial of degree 5. That is, given an equation

$$ax^5 + bx^4 + cx^3 + dx^2 + ex + f = 0,$$

[1] We use the world clearly here to indicate that the fact that this language is not context-free can be formally proven, but it is tedious and not the point of the discussion. The interested reader can try and prove this using the pumping lemma for CFGs.

can we compute the values of x for which is equation holds? Interestingly, if we limit our algorithm to use only the standard operators on numbers $+, -, *, /, \sqrt{}, \sqrt[k]{}$ then no such algorithm exists.[2]

In the final part of this course, we will look at the question of what (formally) is a computation? Or, in other words, what is (what we consider to be) a computer or an algorithm? A precise model for computation will allow us to prove that computers can solve certain problems but not others.

1.1 History

Early in this century, mathematicians (e.g. David Hilbert) thought that it might be possible to build formal algorithms that could decide whether any mathematical statement was true or false. For obvious reasons, there was great interest in whether this could really be done. In particular, he took upon himself the project of trying to formalize the known mathematics at the time. Gödel showed in 1929 that the project (of explicitly describing all of mathematics) is hopeless and there is no finite description of mathematical models.

In 1936, Alonzo Church and Alan Turing independently showed that this goal was impossible. In his paper, Alan Turing introduced the Turing machine (described below). Alonzo Church introduced the ***λ-calculus***, which formed the starting point for the development of a number of functional programming languages and also formal models of meaning in natural languages. Since then, these two models and some others (e.g. ***recursion*** theory) have been shown to be equivalent.

This has led to the Church-Turing Hypothesis.

> ***Church-Turing Hypothesis***: All reasonable models of (general-purpose) computers are equivalent. In particular, they are equivalent to a Turing machine.

This is not something you could actually prove is true (what is reasonable in the above statement, for example?). It could be proved false if someone found another model of computation that could solve more problems than a Turing machine, but no one has done this yet. Notice that we are ignoring how fast the computation can be done: it is certainly possible to improve on the speed of a Turing machine (in fact, every Turing machine can be speeded up by making it more complicated). We are only interested in what problems the machines can or can not solve.

2 Turing machines

2.1 Turing machines at a high level

So far, we have seen two simple models of computation:

- DFA/NFA: finite control, no extra memory, and

[2] This is the main result of Evariste Galois that died at the age of 20(!) in a duel. Niels Henrik Abel (which also died relatively young) proved this slightly before Galois, but Galois work lead to a more general theory.

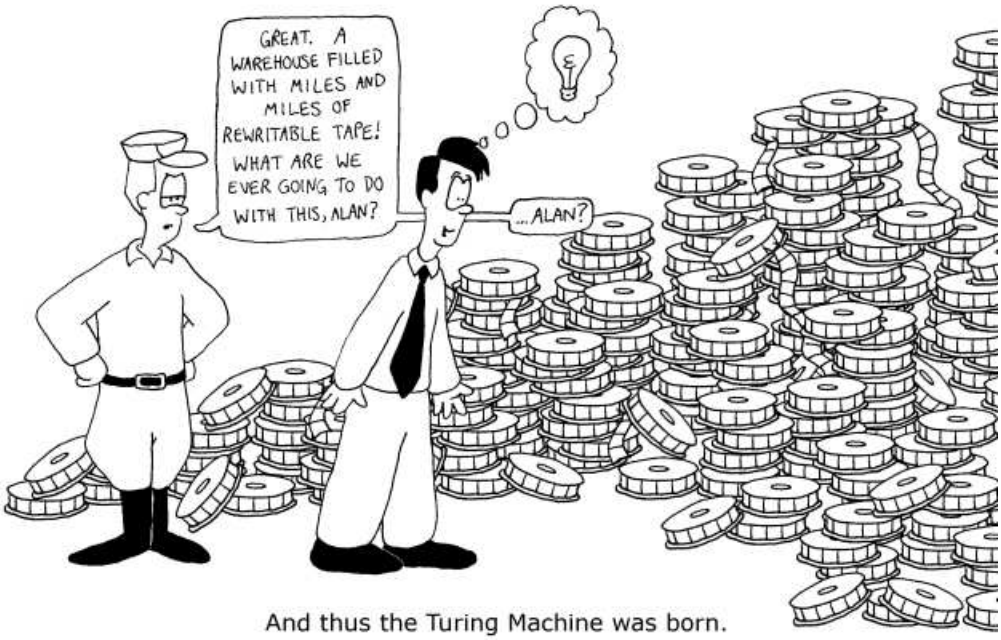


Figure 1: Comic by Geoff Draper.

- Recursive automatas/PDA: finite control, unbounded stack.

Both types of machines read their input left-to-right. They halt exactly when the input is exhausted. Turing machines are like a RA/PDA, in that they have a finite control and an unbounded one dimensional memory tape (i.e., stack). However, a Turing machine is different in the following ways.

(A) The input is delivered on the memory tape (not in a separate stream).

(B) The machine head can move freely back and forth, reading and writing on the tape in any pattern.

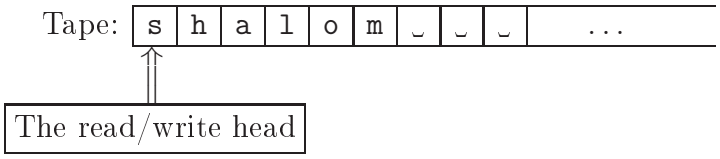
(C) The machine halts immediately when it enters an ***accept*** or ***reject*** state.

Notice condition (C) in particular. A Turing machine can read through its input several times, or it might halt without reading the whole input (e.g. the language of all strings that start with ab can be recognized by just reading two letters).

Moving back and forth along the tape allows a Turing machine to (somewhat slowly) simulate random access to memory. Surprisingly, this very simple machine can simulate all the features of "regular" computers. Here equivalent is meant only in the sense that whatever a regular computer can compute, so can a Turing machine compute. Of course, Turing machines do not have graphics/sound cards, internet connection and they are generally considered to be an inferior platform for computer games. Nevertheless, computationally, TMs can compute whatever a "regular" computer can compute.

2.2 Turing Machine in detail

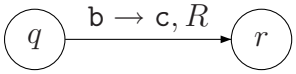
Specifically, a ***Turing machine*** (TM) has a finite control and an infinite tape. In this class, our basic model will have a tape that is infinite only in one direction. A Turing machine starts up with the input string written at the start of the tape. The rest of the tape is filled with a special blank character (i.e., '␣'). Initially, the head is located at the first tape position. Thus, the initial configuration of a Turing machine for the input `shalom` is as follows.



Each step of the Turing machine first reads the symbol on the cell of the tape under the head. Depending on the symbol and the current state of the controller, it then

- (optionally) writes a new symbol at the current tape position,
- moves either left or right, and
- (optionally) changes to a new state.

For example, the following transition is taken if the controller is in state q and the symbol under the read head is `b`. It replaces the `b` with the character `c` and then moves right, switching the controller to the state r.



Note, that Turing machines are deterministic. That is, once you know the state of the controller and which symbol is under the read/write head, there is exactly one choice for what the machine can (and must) do.

The controller has two special states q_{acc} and q_{rej}. When the machine enters one of these states, it halts. It either ***accepts*** or ***rejects***, depending on which of the two it entered.

Note 2.1 If the Turing machine is at the start of the tape and tries to move left, it simply stays put on the start position. This is not the only reasonable way to handle this case.

Note 2.2 Nothing guarantees that a Turing machine will eventually halt (i.e., stop). Like your favorite Java program, it can get stuck in an infinite loop[3]. This will have important consequences later, when we show that deciding if a program halts or not is in fact a task that computers can not solve.

Remark 2.3 Some authors define Turing machines to have a doubly-infinite tape. This does not change what the Turing machine can compute. There are many small variations on Turing machines which do not change the power of the machine. Later, we will see a few sample variations and how to prove they are equivalent to our basic model. The robustness of this model to minor changes in features is yet another reason computer scientists believe the Church-Turing hypothesis.

[3] Or just get stuck inside of Mobile with the Memphis blues again...

2.3 Turing machine examples

2.3.1 The language $w\$w$

For $\Sigma = \{\mathtt{a}, \mathtt{b}, \$\}$, consider the language

$$L = \left\{ w\$w \;\middle|\; w \in \Sigma^* \right\},$$

which is not context-free. So, let describe a TM that accepts this language.

One algorithm for recognizing L works as follows. It first

1. Cross off the first character `a` or `b` in the input (i.e. replace it with `x`, where `x` is some special character)) and remember what it was (by encoding the character in the current state). Let u denote this character.
2. Move right until we see a \$.
3. Read across any `x`'s.
4. Read the character (not `x`) on the tape. If this character is different from u, then it immediately rejects.
5. Cross off this character, and replace it by `x`.
6. Move left past the \$ and then keep going until we see an `x` on the tape.
7. Move one position right and go back to the first step.

We repeat this until the first step can not find any more `a`'s and `b`'s to cross off.

Figure 2 depicts the resulting TM. Observe, that for the sake of simplicity of exposition, we did not include the state q_{rej} in the diagram. In particular, all missing transitions in the diagram are transitions that go into the reject state.

Notice that we did not include the reject state in the diagram, because it is already too messy. If there is no transition shown, we will assume that one goes into the reject state.

Note 2.4 For most algorithms, the Turing machine code is complicated and tedious to write out explicitly. In particular, it is not reasonable to write it out as a state diagram or a transition function. This only works for the relatively simple examples, like the ones shown here. In particular, its important to be able to describe a TM in high level in pseudo-code, but yet be able to translate it into the nitty-gritty details if necessary.

2.3.2 Mark start position by shifting

Let $\Sigma = \{a, b\}$. Write a Turing machine that puts a special character `x` at the start of the tape, shifting the input over one position, then accepting the input.

Accepting or rejecting is not the point of this machine. Rather, marking the start of the input is a useful component for creating more complex algorithms. So you had normally see this machine used as part of a larger machine, and the larger machine would do the accepting or rejecting.

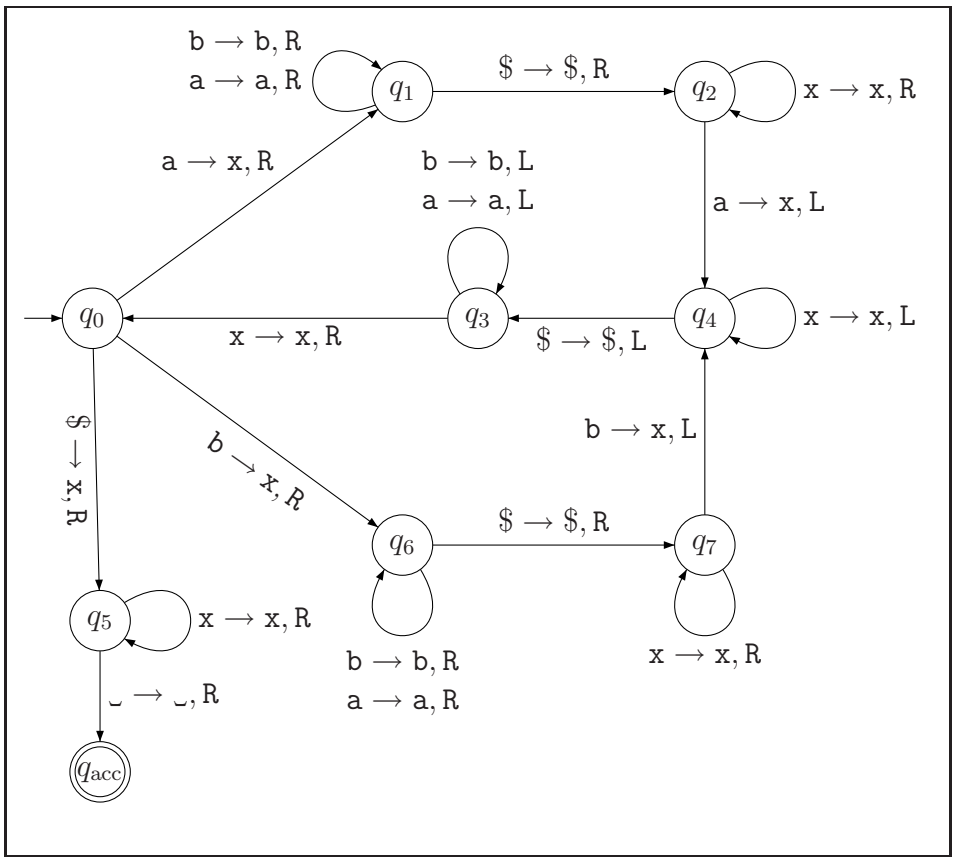


Figure 2: A TM for the language $w\$w$.

2.4 Formal definition of a Turing machine

A ***Turing machine*** is a 7-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) ,$$

where

- Q: finite set of states.
- Σ: finite input alphabet.
- Γ: finite tape alphabet.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}\}$: Transition function.

  As a concrete example, if $\delta(q, c) = (q', c', \mathtt{L})$ means that, that if the TM is at state q, and the head on the tape reads the character c, then it should move to state q', replace c on the tape by c', and move the head on the tape to the left.
- $q_0 \in Q$ is the initial state.
- $q_{\text{acc}} \in Q$ is the ***accepting/final*** state.

- $q_{\text{rej}} \in Q$ is the ***rejecting*** state.

This definition assumes that we've already defined a special blank character. In Sipser, the blank is written $\sqcup$ or $\textvisiblespace$. A popular alternative is B. (If you use any other symbol for blank, you should write a note explaining what it is.)

The special blank character (i.e., $\textvisiblespace$) is in the tape alphabet but it is not in the input alphabet.

2.4.1 Example

For the TM of Figure 2, we have the following $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, where

(i) $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_{\text{acc}}, q_{\text{rej}}\}$.

(ii) $\Sigma = \{\mathtt{a}, \mathtt{b}, \$\}$.

(iii) $\Gamma = \{\mathtt{a}, \mathtt{b}, \$, \textvisiblespace, \mathtt{x}\}$.

(iv) $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}\}$.

	a	b	$	␣	x
q_0	$(q_1, \mathtt{x}, \mathtt{R})$	$(q_6, \mathtt{x}, \mathtt{R})$	$(q_5, \mathtt{x}, \mathtt{R})$	reject	reject
q_1	$(q_1, \mathtt{a}, \mathtt{R})$	$(q_1, \mathtt{b}, \mathtt{R})$	$(q_2, \$, \mathtt{R})$	reject	reject
q_2	$(q_4, \mathtt{x}, \mathtt{L})$	reject	reject	reject	$(q_2, \mathtt{x}, \mathtt{R})$
q_3	$(q_3, \mathtt{a}, \mathtt{L})$	$(q_3, \mathtt{b}, \mathtt{L})$	reject	reject	$(q_0, \mathtt{x}, \mathtt{R})$
q_4	reject	reject	$(q_3, \$, \mathtt{L})$	reject	$(q_4, x, \mathtt{L})$
q_5	reject	reject	reject	$(q_{\text{acc}}, \textvisiblespace, \mathtt{R})$	$(q_5, \mathtt{x}, \mathtt{R})$
q_6	$(q_6, \mathtt{a}, \mathtt{R})$	$(q_6, \mathtt{b}, \mathtt{R})$	$(q_7, \$, \mathtt{R})$	reject	reject
q_7	reject	$(q_4, \mathtt{x}, \mathtt{L})$	reject	reject	$(q_7, \mathtt{x}, \mathtt{R})$
q_{acc}	No need to define				
q_{rej}	No need to define				

Here, reject stands for $(q_{\text{rej}}, \mathtt{x}, \mathtt{R})$.

(Filling this table was fun, fun, fun!)