# Lecture 9: Turing Machines

Ryan Bernstein

## 1 Introductory Remarks

### 1.1 Exam Statistics

Before I hand back the exams, let's talk about grades and statistics.

Way back at the beginning of the term, I told you that I had no intention of grading this class with the traditional 90-80-70 grade boundaries. This is because I think that numeric feedback is a good way of determining which things you need to work on. I'm also of the opinion that if you can perfectly recall 90-100% of the course material, you're either doing exceptionally well or you're not being taught as much as you could or should be.

If you're like me, this may not have been *that* reassuring. You're still seeing the numbers come in, and you've got no idea where in that spectrum the lines are drawn. After the midterm and three assignments, though, I feel like we have enough data to establish grading expectations. These are subject to change over the course of the term, but hopefully they do give you an idea of how well you're doing in the class.

| Grade | Overall Percentage |
|---|---|
| A | 80% |
| B | 70% |
| C | 60% |
| D | 50% |

For current grades, this skews us toward the top of the spectrum, with eight As and eight Bs. And while these cutoffs are, again, subject to change, the point here is this: the numbers may look harsh, but that's because the material is difficult and I don't want to teach you less of it. I'm not going to give everybody Cs.

As for the exam itself, we had:

- A high score of 95
- A mean of 66
- A median of 70

### 1.2 Assignment 2 Solutions

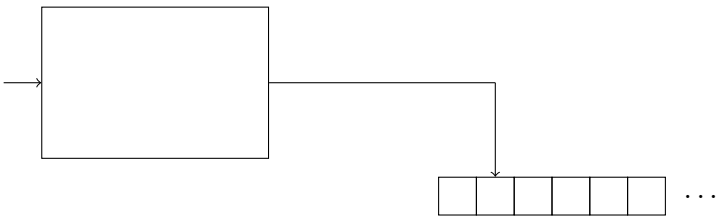
## 2 Turing Universal Machines

Assignment 1 contained a question in which we were wanted to prove that the regular languages were closed under intersection. Assuming the existence of DFAs $M_A = (Q_A, \Sigma, \delta_A, q_{start_A}, F_A)$ and $M_B = (Q_B, \Sigma, \delta_B, q_{start_B}, F_B)$, we had to create a "product" machine by taking the cross product of $Q_A$ and $Q_B$ and simulating the running of both machines in parallel.

One answer that I saw was simpler, and more closely mirrors the way that we as humans would test a string $s$ for membership in $A \cap B$ given these same two machines: just run $s$ through $M_A$ and then run it through $M_B$. If both machines accept, we know that $s \in A \cap B$.

Unfortunately, this wasn't possible. We're limited by a critical limitation in the development of finite (and pushdown) automata: that we read input one character at a time, moving strictly from left to right. We conceived of state machines as black boxes with a "tape head" that moved from the start of $s$ to the end. Running $s$ through both machines in this way would require us to "reset" this tape head to the beginning of the input.

This is the first new property of the next type of machine we'll be looking at. At the "black box" level, *Turing machines* look quite a bit like finite automata. They have a tape head that reads the input from some infinitely-long tape.



The Turing machine has two abilities that we haven't seen before:

1. The tape head is able to move left or right across the tape
2. The tape head is able to write to the tape

Every Turing machine starts with the input string $s$ written directly to the tape. Since this tape is infinitely long, we say that every cell *after* the input is initially populated by a blank character $\sqcup$. Similar to the pushdown automaton, we have an opportunity to both read and write a character from the current cell at every state transition. We also choose whether to move the tape head left or right.

The requirement that the tape head move strictly from left to right gave us a way to tell when our automata were finished processing the input string. Now that we have the ability to backtrack, it's much less clear when to declare ourselves "done" and check to see if we're in an accepting or rejecting state.

To address this, a Turing machine has exactly one accept state $q_{accept}$ and exactly one reject state $q_{reject}$. Until we enter one of these two states, we assume that the string is still being processed. Accordingly, entering either of these states will immediately terminate computation. If the machine terminated its computation by entering $q_{accept}$, we say that it accepted the string; if it ends by entering $q_{reject}$, we immediately say that it rejected the string.

## 2.1 Formal Description of a Turing Machine

A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{reject})$, where:

- $Q$ is a set of states
- $\Sigma$ is the input alphabet
- $\Gamma$ is the tape alphabet, which includes any characters that may be written to the tape. Since we write directly to the input tape, $\Sigma \subset \Gamma$
- $\delta$ is a transition function responsible for describing:
  1. What state we are currently in
  2. What character we read off of the tape next
  3. The state to which we transition
  4. What character we write to our current cell in the tape
  5. Whether we move left or right

  Therefore, $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- $q_{start}$ is the start state
- $q_{accept}$ is the sole accept state
- $q_{reject}$ is the sole reject state

## 2.2 Drawing Turing Machines

Like pushdown automata, state diagrams of Turing machines look a lot like a DFA or NFA, but have different labels on the transitions to represent the different form of the $\delta$ function. We said that $\delta : Q \times \Gamma_\epsilon \to Q \times \Gamma \times \{L, R\}$. Since this is a state diagram, both elements of $Q$ are represented by the state from which the transition originates and the state in which it ends. We label the transition arrow with:

1. The character read from the tape
2. A right-arrow
3. The character written to the current cell in the tape
4. The direction in which the tape head will move (from $\{L, R\}$)

A transition label therefore looks something like $0 \to x, L$.

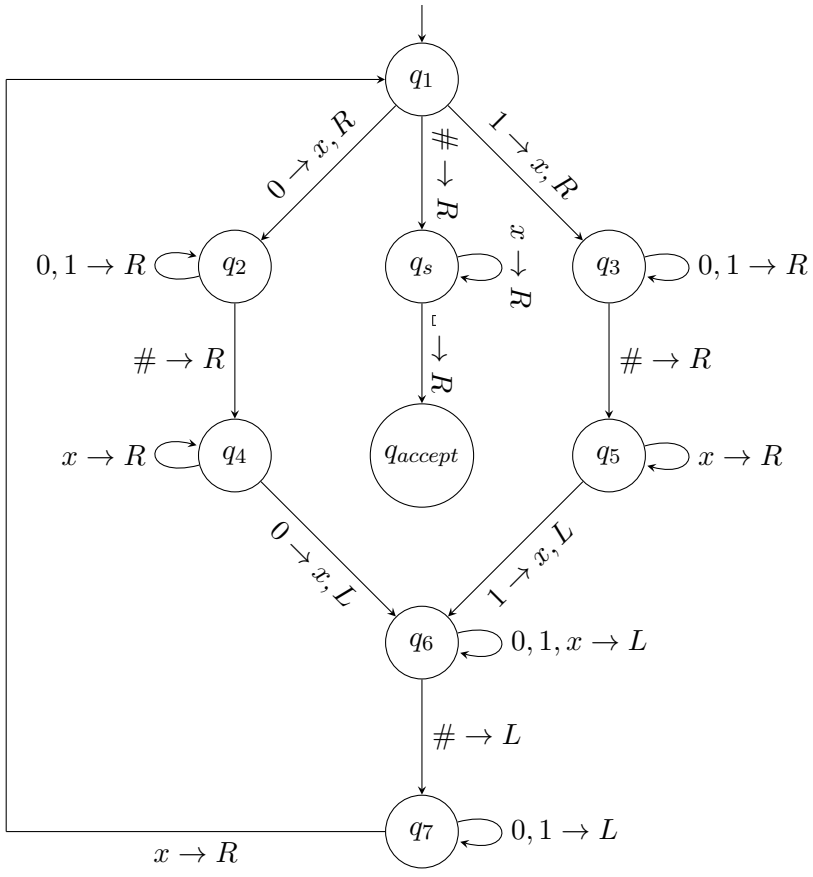
**Example** Draw a Turing machine that decides the language $L = \{w\#w \mid w \in \{0,1\}^*\}$.

Our strategy for this machine is pretty simple. Since we can write to the tape, we'll replace each character that we see on the left side of the # with an $x$, then scan to the right and make sure that the first character

on the right side of the # matches it. We'll then replace that character with an $x$ as well. For the string 01#01, our tape would then go through the following states:

| | |
|---|---|
| 01#01 | Mark the first 0 |
| $x$1#01 | Mark the first non-$x$ character on the right |
| $x$1#$x$1 | Look for the first unmarked character on the left |
| $xx$#$x$1 | Mark the first unmarked character on the right |
| $xx$#$xx$ | *ACCEPT* |

As an aside, this idea of "marking" states will be something we use a lot when constructing Turing machines. Now that we have the ability to write arbitrarily to the tape, we can easily keep track of things that we've already seen by expanding $\Gamma$. If we're moving the tape head between sections, we'll often need some way of remembering the cell from which we started. If we don't need to retain the value of that character, as we don't here, we can simply replace it with an $x$ or similar. If we do, we can just double $\Gamma$, replacing ones with $\mathring{1}$s and zeros with $\mathring{0}$s.




We've omitted explicit transitions to $q_{reject}$ just to simplify the picture.

I told you before that because of how quickly they grew in complexity, I wouldn't require you to draw a pushdown automaton on your homework or on an exam. Since Turing machines get even *more* complex, I'm never going to ask you to draw one of these, either. In fact, I intend to forget everything I just showed you as soon as I walk out the door today, so if you draw a Turing machine on your homework, I probably won't even know how to grade it.

## 2.3 Informal Construction of Turing Machines

Unlike regular and context-free languages, we won't be looking at any syntax (such as a regular expression or a context-free grammar) for describing Turing-decidable languages. To show that a language is decidable, we must construct a Turing machine for it, but we want to avoid drawing them whenever possible. Instead, we'll create Turing machines using informal descriptions of their behavior. As long as each step is algorithmically possible, the machine we describe should be possible as well.

**Example 2** Provide an informal description of the Turing machine that decides $\{w\#w \mid w \in \{0,1\}^*\}$

$M =$ "On input $w$:

1. If the string is not of the form $(0 \cup 1)^* \circ \# \circ (0 \cup 1)^*$, $REJECT$
2. While unmarked characters remain to the left of the #:
   (a) Mark the first unmarked character on the left side of the #
   (b) If the character marked was a zero, ensure that the first unmarked character on the right of the # is a zero. Mark this character.
   (c) If the character marked was a one, ensure that the first unmarked character on the right of the # is a one. Mark this character
3. If unmarked characters remain on the right side of the #, $REJECT$
4. $ACCEPT$"

As you can see, descriptions of these machines are a lot less formal than anything we've dealt with so far. Step 1 seems particularly powerful. How can we ensure that a string matches $(0 \cup 1)^* \circ \# \circ (0 \cup 1)^*$ without describing the movement of the tape head?

The answer lies in the fact that this is a regular expression, which means it represents a regular language. We can decide regular languages with a DFA that never modifies its input. This means that we can begin our Turing machine with a series of states that simulates that DFA. A "verification" scan of a string to see if it matches a regular expression is therefore always allowable.

**Example 3** Provide an informal description of a Turing machine that decides $\{1^{2^n} \mid n \geq 0\}$.

This machine is based on the idea that if we repeatedly divide a power of two in half will, we will see an even number each time until we eventually reach the value 1.

$M =$ "On input $w$:

1. If $w$ is not of the form $0^*$, $REJECT$
2. Sweep across the input from left to right, replacing off every other 1 with an $x$. If we traverse across an odd number of ones:
   - If this number was one, $ACCEPT$
   - $REJECT$
3. Return to the left end of the input tape

4. Go to step 1"

**Worksheet Exercise** Provide an informal description of a Turing machine that decides $\{0^n1^n2^n \mid n \geq 0\}$.

# 3 Decidability vs. Recognizability

Way back in Lecture 1, I mentioned that eventually, there would be a difference between the terms "decide" and "recognize". Until now, these concepts have been interchangeable. All of our models of computation operate on a string and definitively answer the question of whether or not it's an element of some language. Finite and pushdown automata accept or reject a string. A regular expression is or is not capable of matching a string. And a context-free grammar is or is not capable of generating a string.

Here, though, things become a bit more ambiguous. A Turing machine has a single accept state $q_{accept}$, which is something we've seen before in GNFAs. But it also has only one rejecting state, $q_{reject}$. When a Turing machine is given some string as input, it may accept it or reject it. But there's a third option as well: it may never enter either of these states at all.

We say that a Turing machine *decides* a language $L$ if it accepts or rejects any arbitrary string $s$ based on its membership in $L$. We say that a Turing machine *recognizes* a language $L$ if it accepts every string in the language. If a decider exists for some language, we say that the language is *Turing-decidable*; if a recognizer exists for it, we say that the language is *Turing-recognizable*.

What's the difference here? If a recognizer for $L$ will accept every string in $L$, can't we just assume that any string that it doesn't accept is not a member of $L$? This is where runtime comes into play. Since we no longer require the processing of our string to go strictly from start to end, we have no clear way of determining when the machine has "finished". If we see that the machine is not in $q_{accept}$ or $q_{reject}$, does this mean that $s$ is not a member of $L$? Or does the machine simply need more time to process?

We can think of a Turing machine as a predicate function that takes a string $s$ and returns a Boolean that indicates whether or not $s$ is in $L$. Entering $q_{accept}$ or $q_{reject}$ means that our function has returned `true` or `false`. Unless it does this, we don't know whether the function is still processing or if it is stuck in some infinite loop.

We'll be discussing more about what makes languages decidable or recognizable next week.

## 3.1 Turing Machines as Algorithms

You may have heard programming languages described as *Turing-complete* or *Turing-equivalent*. We've now seen enough to know what this means: a language is Turing equivalent if it can be used to simulate any arbitrary Turing machine. Why is this important?

The Church-Turing Thesis states that any computable function can be computed using a Turing machine like the ones we've been discussing. Turing-equivalence therefore means that a language is equivalent in power not only to a Turing machine itself, but also to every other Turing-equivalent language.

What does this mean for our Turing machines themselves? Since these are capable of computing anything that can be computed, what we're doing here is actually creating algorithms, albeit in a very restrictive

format. This also means that proof of Turing-decidability — which we're doing every time we create a Turing machine for some language — is also proof of computability in general.