Craig Silverstein
Gates 470
Phone: (72)3-1787
Office Hours: Wed 4:30–5:30, Fri 1–3, Gates 470
Section: Monday 2:15–3:15, 380-380W

Computer Science 154
School of Engineering
Stanford University

Section Handout #5

Those kick-butt Turing Machines!

Turing machines kick major hiney because of Church's thesis, which is basically: "Anything you can do my TM can do better." You can think of them as DFAs that can "back up" on the input string and also "take notes." Here's a comparison table:

Finite Automata	**Pushdown automata**	**Turing machines**
finite states	finite states	finite states
no stack	stack	no stack
read-once input tape	read-once input tape	read-write input tape

Let me expand on something that was said in class. Turing Machine tapes are *not* infinite; rather, they are arbitrarily long. At first, the tape has only $|w|$ squares. As you move right into the unknown, new squares (filled with the empty symbol $\sqcup$) are quickly stitched on to the end. This is the theoretical equivalent of "just in time" compilation. :-) Since each transition can create at most one new tape sqaure, if the TM halts in a finite number of steps, the length of the tape will be finite. This is good, since otherwise IDs could have infinite length (since they're supposed to encode the entire contents of the tape). That would be Bad.

Where does the tape-building happen? It happens in the definition of "moving right." Remember that moving right was defined to be going from the ID $(q, \alpha a, b\beta)$ to $(q', \alpha ab, \beta)$. But what if $\beta = \epsilon$? In this case, we just define moving right to be $(q, \alpha a, b) \to (q', \alpha ab, \sqcup)$. We've just created a new tape square. We can't have the third part of the ID ever be ϵ, because the tape head lives in the third part of the ID, and we can't have the tape head being in limbo.

Notice that the idea of falling-off-the-edge-of-the-tape is, similarly, in the definition of "moving left." Moving left means going from ID $(q, \alpha a, b\beta)$ to $(q', \alpha, ab\beta)$. But there is no such rule if we're at the very left of the tape (because then the second part of the ID is ϵ, which is not αa for any a) so there's no rule that applies. Thus you can't go to a new ID, and you're "stuck," just like if you were in an infinite loop.

What Turing Machines Do

Turing machines can do three things on an input w: accept (get to state q_a), reject (get to state q_r), or never halt (fall off the left end of the tape, explore the infinite nature of tape-hood, get into an infinite loop; in any case, never land in q_a or q_r). We give different names to the TM depending on which of these three things it does to different strings.

First off, every TM has a language associated with it. $L(M)$ is the set of all strings M **accepts** (that is, gets to q_a on). Obviously, this means M either rejects or never-halts on strings not in $L(M)$. If we know M will never never-halt, we say M **decides** $L(M)$ in addition to accepting it.

What's the difference? The point is that if M loop forever, you'll never know. You look at M, and it's chugging away, and you think: Is M in an infinite loop, or is it just slow? Everyone has had this experience with programs they've written. If your TM is a deciding TM, you *know* it's just slow: deciding TMs never get into an infinite loop. Otherwise, *shrug*.

ALL TMs are accepting TMs (just look at the definition). In particular, all deciding TMs are also accepting TMs.

What Languages Do

Accept and Decide — these are properties of TMs.

Recursive and Recursively Enumerable — these are properties of *languages*.

If there is some TM that *accepts* a language L (that is, $L = L(M)$ for some TM M), L is **recursively enumerable** or **R.E.** (not to be confused with Regular Expression!) If there is some TM that decides a language L, L is recursive. Note that the definition is *some* TM: if you find a TM that decides L, L is recursive. But if you find a TM that only accepts L without deciding it, L may still be recursive, because some other TM may work better. (L is RE in both cases.)

Just as the set of deciding TMs is a subset of the set of accepting TMs, the set of recursive languages is a subset of the set of RE language. But while all TMs are accepting, **not all languages are RE**. This is what the halting problem is all about (more on that later).

Here's the chart:

What M does	$L(M)$ is...	We call M...	We call $L(M)$...
Accept some strings, reject some strings, never-halt on some strings	strings M accepts	accepting	RE
Accept some strings, reject the rest of the strings	strings M accepts	deciding	recursive

Hints on Homework 4

(1) People ask me: what does rigorous proof mean? It means you show how to translate from one machine to another and prove the translation preserves acceptance. Suppose you think the answer is "the context-free languages." Then you would take the 7-tuple for the TM (where δ is of a restricted form) and give the conversion from that to the tuple for a PDA. You would then prove the PDA and the TM accept the same language (using an inductive if-and-only-if proof). Then you show how, given a PDA, to construct a TM accepting the same language, where the δ function of the TM has the restricted form. Again prove the langauges are the same.

The result is 4 proofs. But you can do the proof once and then point out the differences in the other versions. That is, to do an if-and-only-if proof you need to prove things in two directions, but if any step is basically the same in either direction (eg constructing a pair as opposed to destructing the pair), you don't need to talk about it in both directions.

(2) "Formally describe" means we want a table for the δ function. You can draw a diagram too to help with the intuition, but it's not enough. You do not need to *prove* your TM works, but an intuition of why it works — that is, what you're trying to get the TM to do — would be nice.

(3) R.E. here means "recursively enumerable" (not "regular expression"!). If L is RE, there's a TM that does what to words in L? What does it do to words not in L? If $\overline{L}$ is RE, there's a (different) TM that does what to words in L (not $\overline{L}$, but L)? What does it do to words not in L? Wouldn't it be nice if you had a machine that was the best of both worlds?

Your proof here can be more handwavy than in problem (1). A verbal description of what you would do at each transition is enough, and an explanation of why it works. eg "We copy the input to the second tape. Then we run M_1 on the first tape, but when M_1 would go to state q_a, we go to state q_0 of M_2 and run M_2 on the second tape. If M_1 goes to state q_r we reject. This accepts $L(M_1) \cap L(M_2)$ because if w is in both languages, then..."

(4) Compare the singleton-TM M' and the original TM M. M' has fewer states, but more tapes. Can a surfeit of one compensate for an deficit of the other? Hint: this problem is easiest if you extend the alphabet of M'.