Solutions for Week Nine

Regular vs. Context-Free

i. $\Sigma = \{a, b\}$ and $L = \{(ab)^n \mid n \in \mathbb{N}\}$.

This language is regular. It corresponds to the regular expression $(ab)^*$. If your solution used an automata, try using the state elimination algorithm to see if you get an equivalent regular expression!

ii. $\Sigma = \{a, b\}$ and $L = \{(ab)^n a^n \mid n \in \mathbb{N}\}$.

This language is NOT regular.

Proof: We will use the Myhill-Nerode Theorem to prove that L is not regular. Let $S = \{ (ab)^n \mid n \in \mathbb{N} \}$. Notice that S is infinite because it contains one string per natural number and there are infinitely many natural numbers. We also claim that any two distinct strings from S are distinguishable relative to L. To see this, consider any two strings $(ab)^n$, $(ab)^m \in S$ with $m \neq n$. Then $(ab)^n a^n \in L$ but $(ab)^m a^n \notin L$. Therefore, we see that $(ab)^n \not\equiv_L (ab)^m$. Consequently, by the Myhill-Nerode theorem, we know that L is not regular. ■

However, L is context-free, as shown by the following CFG.

$S \to abSa \mid \varepsilon$

iii. $\Sigma = \{a, b\}$ and $L = \{(ab)^n a^m \mid n, m \in \mathbb{N}$ and the total number of a's is even $\}$.

This language is regular. Note that since the total number of a's is even, either both n and m are even, or both are odd. Thus we can write L as the regular expression $((abab)^2)^*((aa)^2)^* \cup ab((abab)^2)^*a((aa)^2)^*$.

iv. $\Sigma = \{a, b\}$ and $L = \{ w \in \Sigma^* \mid$ every prefix of w has at least as many a's as b's $\}$.

This language is NOT regular.

Proof: We will use the Myhill-Nerode Theorem to prove that L is not regular. Let $S = \{ a^n \mid n \in \mathbb{N} \}$. Notice that S is infinite because it contains one string per natural number and there are infinitely many natural numbers. We also claim that any two distinct strings from S are distinguishable relative to L. To see this, consider any two strings a^n, $a^m \in S$ with $m \neq n$. Without loss of generality, assume $n > m$. Then $a^n b^n \in L$ but $a^m b^n \notin L$. Therefore, we see that $a^n \not\equiv_L a^m$. Consequently, by the Myhill-Nerode theorem, we know that L is not regular. ■

However, L is context-free, as shown by the following CFG. The key insight here is that every b must be preceded by some a, plus there can be some extra a's. You should convince yourself that both (a) every string generated by this grammar is in L, and also (b) every string in L can be generated by this grammar (not obvious).

$S \to aSbS \mid aS \mid \varepsilon$

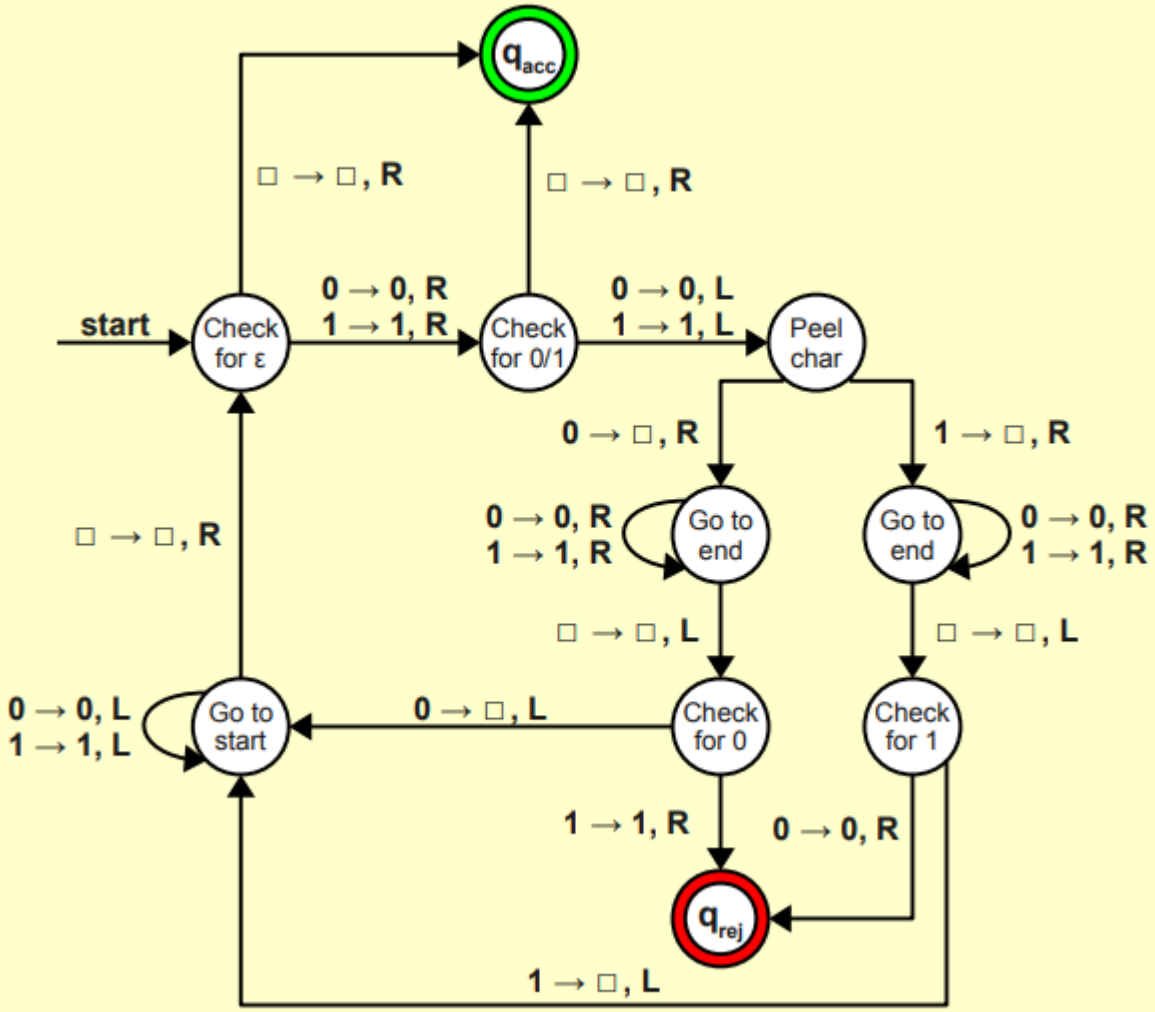
Here's another CFG that also works:

$S \to SS \mid aSb \mid a \mid \varepsilon$

Turing Machines

i. Let $\Sigma = \{0, 1\}$ and let $L = \{ w \in \Sigma^* \mid w \text{ is a palindrome} \}$. Draw a TM for L.

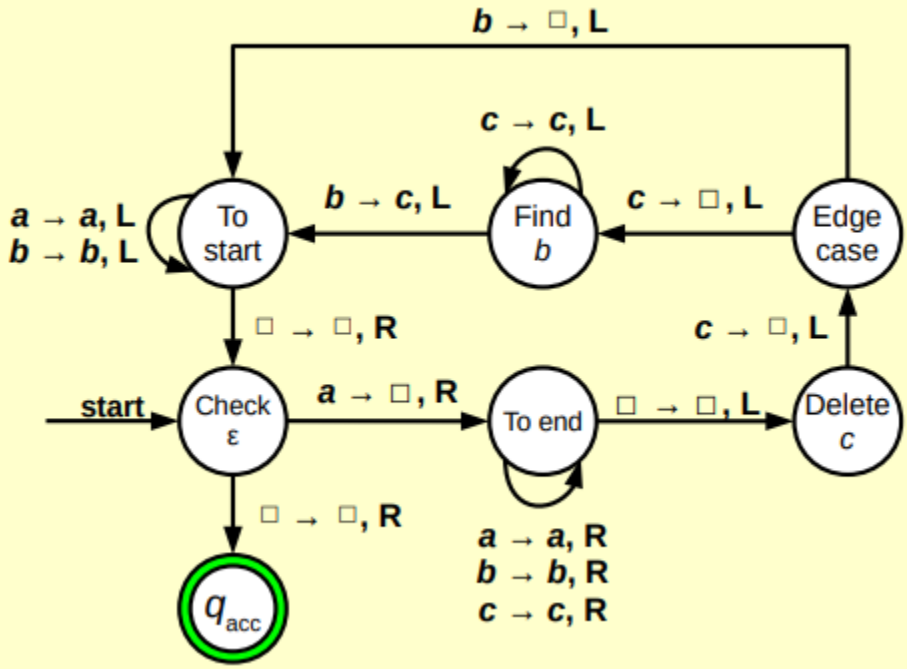
Here is one possible option:



This TM is based on the following recursive observations: the strings 0, 1, and ε are all palindromes. Otherwise, a string is a palindrome if and only if the first and last characters match and removing them leaves a palindrome. Notice how the TM uses constant storage to remember what the first character of the string is.

ii. Draw the state-transition diagram for a TM whose language is $\{ a^n b^n c^n \mid n \in \mathbb{N} \}$.

Here's one option:



This machine works by crossing off an *a* from the front of the string, then going to the back of the string and deleting a *c*. If there are no *c*'s left, it then tries to cross off a *b*. Otherwise, it deletes a second *c*, finds the first *b* it can, and replaces that *b* with a *c*. This has the effect of converting $a^n b^n c^n$ into $a^{n-1} b^{n-1} c^{n-1}$. Eventually, if the string becomes empty, we accept. Otherwise, at some point we run into a mismatched or misordered symbol and reject.

Why we asked this question: Part (i) of this problem was designed to get you thinking about how to use constant storage in the TM (specifically, to remember what character you'd need) and to think about how to recursively simplify a complex problem into a much simpler one. Part (ii) we liked because it has the same general structure as the TM for the language $\{a^n b^n \mid n \in \mathbb{N}\}$ (using recursion to keep peeling off matching symbols until we run out of characters), but is a bit more involved because the mechanism for crossing off characters is a bit trickier. Hopefully, if you felt comfortable with the high-level approach to the problem, you were able to come up with a TM that was at least on the right track.

The Story So Far

i. Show that **REG** ≠ **R**.

In lecture, we built a decider for $\{ 0^n1^n \mid n \in \mathbb{N} \}$, so $\{ 0^n1^n \mid n \in \mathbb{N} \} \in$ **R**. However, we also know that this language is not regular (it's the canonical nonregular language), so $\{ 0^n1^n \mid n \in \mathbb{N} \} \notin$ **REG**. Therefore, **R** ≠ **REG**.

ii. Show that **R** ⊆ **RE**.

A decider for a language L is a TM M where $\mathscr{L}(M) = L$ and M halts on all inputs. A recognizer for a language L is a TM M where $\mathscr{L}(M) = L$. Therefore, any decider for a language L is also a recognizer for the same language L, so there isn't even any conversion necessary.

Why we asked this question: We've breezed pretty quickly through the explanations of how all these classes of languages relate, but it's quite interesting to actually see the transformations involved here. We hoped that this problem would give you a better sense for how all these models of computation relate.

Closure Properties of R

i. Let L_1 and L_2 be decidable languages over the same alphabet Σ. Prove that $L_1 \cup L_2$ is also decidable. To do so, suppose that you have methods *inL1* and *inL2* matching the above conditions, then show how to write a method *inL1uL2* with the appropriate properties. Then, briefly justify why your construction is correct.

Consider this method:

```
bool inL1uL2(string w) {
    return inL1(w) || inL2(w);
}
```

Theorem: If L_1 and L_2 are decidable, then $L_1 \cup L_2$ is decidable.

Proof: Consider the above piece of code. If it's given a string $w \in L_1 \cup L_2$, then we know that either $w \in L_1$ or $w \in L_2$ (or both). Therefore, at least one of *inL1(w)* and *inL2(w)* will return true. Since *inL1* and *inL2* always return values, this means that the expression will always eventually evaluate the call to the method that returns true, so this method returns true. On the other hand, if it's given a string $w \notin L_1 \cup L_2$, then we know that $w \notin L_1$ and $w \notin L_2$. Therefore, *inL1(w)* and *inL2(w)* will return false, so the overall method returns false. Overall, we've seen that this method returns true if $w \in L_1 \cup L_2$ and false otherwise, so the method is (essentially) a decider for $L_1 \cup L_2$, so $L_1 \cup L_2$ is decidable, as required. ■

ii. Repeat problem (i), except proving that the **R** languages are closed under concatenation.

Consider this method:

```
bool inL1L2(string w) {
    for (int i = 0; i <= w.length(); i++) {
        if (inL1(w.substring(0, i)) && inL2(w.substring(i)) {
            return true;
        }
    }
    return false;
}
```

Theorem: If L_1 and L_2 are decidable, then $L_1 L_2$ is decidable.

Proof: Consider the above piece of code. If it's given a string $w \in L_1L_2$, then we know that there is some way to write $w = xy$ where $x \in L_1$ and $y \in L_2$. The above code will try all possible ways of splitting w into x and y and test each to see if they have the appropriate properties. If we test an incorrect split, then both calls to *inL1* and *inL2* will return but not both return true, so we skip and go to the next iteration. If we test a correct split, both calls to *inL1* and *inL2* will return true, so we return true overall. Similarly, if it's given a string $w \notin L_1L_2$, then we know that there is no possible way to split the string into two pieces x and y where $x \in L_1$ and $y \in L_2$. Therefore, each iteration of the loop will complete but fail to return true, so at the end we end up returning false. Overall, this means that the above method returns true if the input is in L_1L_2 and returns false otherwise, so it's (essentially) a decider for L_1L_2. Thus L_1L_2 is decidable. ■

Decidable Languages

Theorem: *a*b* is undecidable.

Proof: By contradiction; assume *a*b* is decidable. Let *D* be a decider for it. Consider what happens when we run *D* on a string of infinitely many *a*'s followed by a *b*, and on a string of infinitely many *a*'s. Let's call this first string *x* and the second string *y*. Since *D* is a decider, it halts on all inputs, and therefore cannot run for an infinitely long time. Therefore, *D* must halt before reading the last character of *x* and the last character of *y*. Because *x* and *y* are the same except for their last character, we see that *D* must have the same behavior when run on *x* and when run on *y*. If *D* accepts *x*, then *D* also accepts *y*, but *y* is not in the language *a*b*. Otherwise, *D* rejects *x*, but *x* is in the language *a*b*. Both cases contradict the fact that *D* is a decider for *a*b*. We have reached a contradiction, so our assumption must have been wrong. Thus *a*b* is undecidable. ■

What's wrong with this proof?

By definition, all strings must have finite length, so there's no such thing as an infinite-length string. Therefore, the strings *x* and *y* described here don't actually exist, so you can't run *D* on those two strings at all.

Why we asked this question: Every quarter, we usually get one or two people asking this question, either as a generalization of Myhill-Nerode ("why can't you use Myhill-Nerode for TMs?") or because they're convinced that this argument shows the regular languages aren't always decidable. Resolving this question relies on a somewhat technical point (strings can't have infinite length) that might seem like a bit of a cop-out, but actually hits at a deeper point: you can have languages with infinitely many different strings, but where each string in the language actually happens to be finite.