Solutions for Week Nine

Context-Free Grammars

i. Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* | \text{w has no a's or has no b's}\}$. Write a CFG for $L$.

One option is

$$
S \rightarrow A \mid B \\
A \rightarrow aA \mid \varepsilon \\
B \rightarrow bB \mid \varepsilon
$$

Any string in $\Sigma^*$ that has no a's or no b's (for this choice of $\Sigma$) must consist purely of a's or purely of b's. This grammar works by having dedicated nonterminals for each case, then choosing which case to work with.

ii. Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* | \text{w has at least one a and at least one b}\}$. Write a CFG for $L$.

One option is

$$
S \rightarrow XaXbX \mid XbXaX \\
X \rightarrow aX \mid bX \mid \varepsilon
$$

Here, $X$ generates any possible string. This grammar works by, at the top level, placing down an a and a b and filling the gaps with arbitrary strings.

Another option is shown here:

$$
S \rightarrow aX \mid bY \\
X \rightarrow aX \mid bZ \\
Y \rightarrow aZ \mid bY \\
Z \rightarrow aZ \mid bZ \mid \varepsilon
$$

Here, each nonterminal stands for a particular combination of letters that have appeared so far: $S$ means "nothing has been written yet," $X$ means "only a's have been written," $Y$ means "only b's have been written," and $Z$ means "both a and b have been written." Each production rule then writes a single character and switches which nonterminal is written down. Only when we get to nonterminal $Z$ – when both a and b have been written – are we allowed to expand the nonterminal out into $\varepsilon$ and stop adding characters.
iii. Let $\Sigma = \{a, b\}$ and let $L = \{a^n ba^n \mid n \in \mathbb{N}\}$. Write a CFG for $L$.

This one is all about finding a build ordering. We build both groups of $a$'s simultaneously, working from the outside inward. Here's one grammar that does this:

$$S \rightarrow aSa \mid b$$

Notice the similarity between this grammar and the grammar for $\{a^n b^n \mid n \in \mathbb{N}\}$

iv. Let $\Sigma = \{a, b\}$ and let $L = \{a^n b^{2n} \mid n \in \mathbb{N}\}$. Write a CFG for $L$.

This grammar generalizes the grammar for $\{a^n b^n \mid n \in \mathbb{N}\}$ and works by building the $a$'s and $b$'s at the same time, working from the outside inward:

$$S \rightarrow aSbb \mid \varepsilon$$

You might have noticed that the trick of building from the outside inward is pretty common in context-free grammars.

v. Let $\Sigma = \{a, b\}$ and let $L = \{a^n b^m \mid n, m \in \mathbb{N} \text{ and } n \leq m \leq 5n\}$. Write a CFG for $L$.

There are many ways we can do this. One option is shown below. It works by using the general pattern from before, except that whenever we place down an $a$, we place down between one and five $b$'s on the other side:

$$S \rightarrow aSb \mid aSbb \mid aSbbbb \mid aSbbbb | \varepsilon$$

This works, but is a bit lengthy. Here's another option that works by saying that whenever we put an $a$ down on one side of the string, we have to put a $b$ down on the other, followed by four optional $b$'s:

$$S \rightarrow aSbXXX \mid \varepsilon$$

$$X \rightarrow b \mid \varepsilon$$

Here, the nonterminal $X$ means “either $b$ or nothing at all.”
vi. Let $\Sigma = \{a, b\}$ and let $L = \{a^nb^m \mid n, m \in \mathbb{N} \text{ and } n \neq m\}$. Write a CFG for $L$.

The main insight here is that any string of this form consists of a string of the form $a^nb^n$ either with some extra $a$'s tacked onto the front or some extra $b$'s tacked onto the back. We can use these insights to build this grammar:

$$
S \rightarrow AE \mid EB \\
E \rightarrow aEb \mid \varepsilon \\
A \rightarrow aA \mid a \\
B \rightarrow bB \mid b
$$

Here, $E$ generates all strings of the form $a^nb^n$, and $A$ and $B$ generate $a^+$ and $b^+$, respectively.

vii. Let $\Sigma = \{a, b, c\}$ and let $\mathcal{L} = \{a^nb^mc^p \mid n, m, p \in \mathbb{N} \text{ and } n = m \text{ or } n = p\}$. Write a CFG for $\mathcal{L}$.

I love this problem because it combines everything together. There are two options here: if $n = p$, then we need to build matching $a$'s and $c$'s on the outside and, when we're done, we fill in the middle with $b$'s. If $n = m$, then we build matching $a$'s and $b$'s and append any number of $c$'s. The main challenge is getting everything structured appropriately. Here's one option:

$$
S \rightarrow XC \mid Y \\
X \rightarrow aXb \mid \varepsilon \\
C \rightarrow cC \mid \varepsilon \\
Y \rightarrow aYc \mid B \\
B \rightarrow bB \mid \varepsilon
$$

Here, $X$ generates strings of the form $a^nb^n$. $Y$ generates strings of the form $a^nb^mc^n$ by generating the matching $a$'s and $c$'s on the outside and filling the middle with any number of $c$'s.

viii. Let $\Sigma = \{a, b\}$ and let $L = \{a^nb^n \mid n \in \mathbb{N}\}$. Write a CFG for $L^*$, the Kleene closure of $L$.

The grammar given below works by starting with a grammar for $\{a^nb^n \mid n \in \mathbb{N}\}$ and then letting us place down zero or more copies of strings from that language:

$$
S \rightarrow ES \mid \varepsilon \\
E \rightarrow aEb \mid \varepsilon
$$

Notice that the start symbol $S$ expands out into some string of $E$'s, and each $E$ expands to a string of the form $a^nb^n$. 
ix. Let $\Sigma = \{a, b\}$ and let $L = \{a^nb^m \mid n, m \in \mathbb{N} \text{ and either } n=2m \text{ or } m=2n \}$. Write a CFG for $L$.

The key insight here is that there are two independent options to pick from: we either have $n = 2m$ or we have $m = 2n$, and each option is something we can build a CFG for:

$$
S \rightarrow X \mid Y \\
X \rightarrow aXbb \mid \epsilon \\
Y \rightarrow aaYb \mid \epsilon
$$

x. Let $\Sigma = \{a, b\}$ and let $L = \{a^nb^n \mid n \in \mathbb{N}\}$. Write a CFG for $L$, the complement of $L$.

The main insight here is that any string not in $L$ must either (1) not consist of a series of $a$'s followed by a series of $b$'s or (2) consist of a series of $a$'s followed by a series of $b$'s, but not have the same quantity of each. Strings in case (1), as you saw on Problem Set Seven, are just the strings containing $ba$ as a substring. Strings in case (2) are generated by the CFG from part (vi) of this problem. Putting those insights together gives us the following:

$$
S \rightarrow XbaX \mid AE \mid EB \\
X \rightarrow aX \mid bX \mid \epsilon \\
E \rightarrow aEb \mid \epsilon \\
A \rightarrow aA \mid a \\
B \rightarrow bB \mid b
$$

Why we asked this question: CFGs are a powerful and expressive framework for defining languages, but they’re much trickier to design and work with than regular expressions. This problem was designed to give you a ton of practice writing grammars and understanding the key techniques involved (look for a build order, store information in nonterminals, think recursively, etc.). Although these languages all seem to more or less be variation on a theme, as you can see, the shapes of the grammars necessary to generate those languages look quite different!
Turing Machines

i. Let $\Sigma = \{0, 1\}$ and let $L = \{ w \in \Sigma^* | w$ is a palindrome $\}$. Draw a state-transition diagram of a TM for $L$.

Here is one possible option:

This TM is based on the following recursive observations: the strings 0, 1, and $\varepsilon$ 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 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 $\text{REG} \neq \text{R}$.

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

ii. Show that $\text{R} \subseteq \text{RE}$.

A decider for a language $L$ is a TM $M$ where $\mathcal{L}(M) = L$ and $M$ halts on all inputs. A recognizer for a language $L$ is a TM $M$ where $\mathcal{L}(M) = L$. 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 \( \mathbb{R} \)

i. Let \( L_1 \) and \( L_2 \) be decidable languages over the same alphabet \( \Sigma \). Prove that \( L_1 \cup L_2 \) is also decidable.

To do so, suppose that you have methods \( \text{inL1} \) and \( \text{inL2} \) matching the above conditions, then show how to write a method \( \text{inL1uL2} \) with the appropriate properties. Then, briefly justify why your construction is correct.

Consider this method:

```java
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 \( \text{inL1}(w) \) and \( \text{inL2}(w) \) will return true. Since \( \text{inL1} \) and \( \text{inL2} \) 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 \( w \not\in L_1 \cup L_2 \), then we know that \( w \not\in L_1 \) and \( w \not\in L_2 \). Therefore, \( \text{inL1}(w) \) and \( \text{inL2}(w) \) 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 \( \mathbb{R} \) languages are closed under concatenation.

Consider this method:

```java
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 \( L_1 \cdot L_2 \) is decidable.

**Proof:** Consider the above piece of code. If it’s given a string \( w \in L_1 \cdot L_2 \), 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 \( \text{inL1} \) and \( \text{inL2} \) 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 \( \text{inL1} \) and \( \text{inL2} \) will return true, so we return true overall. Similarly, if it’s given a string \( w \not\in L_1 \cdot L_2 \), 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 \( L_1 \cdot L_2 \) and returns false otherwise, so it’s (essentially) a decider for \( L_1 \cdot L_2 \). Thus \( L_1 \cdot L_2 \) 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.