Problem Set 6 Solutions

Problem One: The Complexity of Addition (16 Points)

Consider the language $ADD = \{\mathtt{1}^m\mathtt{+}\mathtt{1}^n\overset{?}{=}\mathtt{1}^{m+n} \mid m, n \in \mathbb{N}\}$. That is, ADD consists of strings encoding two unary numbers and their sum.

i. Prove that ADD is not a regular language.

Proof: Let $S = \{\ \mathtt{1}^n \mid n \in \mathbb{N}\ \}$. S is an infinite set of strings, since it contains one string for each natural number. Now, consider any two distinct strings $\mathtt{1}^n$, $\mathtt{1}^m \in S$. Then $1^n\mathtt{+}\overset{?}{=}\mathtt{1}^n \in ADD$, but $\mathtt{1}^m\mathtt{+}\overset{?}{=}\mathtt{1}^n \notin ADD$. Therefore, there is a string w (namely, $\mathtt{+}\overset{?}{=}\mathtt{1}^n$) such that the string $\mathtt{1}^n w \in ADD$ and the string $\mathtt{1}^m w \notin ADD$, so any two distinct strings in S are distinguishable relative to ADD. Therefore, S is an infinite set of strings distinguishable relative to ADD, so by the Myhill-Nerode theorem ADD is not regular. ■

ii. Write a context-free grammar for ADD. Show a derivation of $\mathtt{1+1}\overset{?}{=}\mathtt{11}$ and $\mathtt{+111}\overset{?}{=}\mathtt{111}$ using your grammar. This proves that ADD is context-free.

One possible grammar is

$$S \to \mathtt{1}S\mathtt{1} \mid \mathtt{+}B$$
$$B \to \mathtt{1}B\mathtt{1} \mid \mathtt{=}$$

The idea behind this grammar is that S begins by adding an equal number of `1`s to the start and end of the string. We then lay down a **+** in-between these ones, then start expanding B, which places some other number of `1`s in the string, half at the end and half after the **+** we just laid down. This process terminates by having B expand out to the **=** sign. Here are the requested derivations:

S ⇒ 1S1 ⇒ 1+B1 ⇒ 1+1B11 ⇒ 1+1=11	S ⇒ +B ⇒ +1B1 ⇒ +11B11 ⇒ +111B111 ⇒ +111=111

Why we asked this question: We wanted part (i) of this question to be a gentle introduction to the Myhill-Nerode theorem. We hoped that the question would be similar enough to the examples from lecture that you would be able to notice the similarity and focus on writing the proof rather than searching for the right set. Part (ii) of this question was designed to help you gain experience designing context-free grammars. We chose it because your intuition about how the strings ought to be built probably doesn't match the actual way the build order works.

Problem Two: Minimum-State DFAs (28 Points)

i. Suppose that S is a finite set of n strings distinguishable relative to L. Prove that any DFA for L must have at least n states. *(Hint: Start with the proof of Myhill-Nerode from lecture and make appropriate modifications.)*

Proof: Let L be a language and S a set of n strings distinguishable relative to L. We will prove by contradiction that any DFA for L must have at least n states. Assume for the sake of contradiction that there is some DFA D for L that has fewer than n states. By the pigeonhole principle, since there are n strings in S and at most $n - 1$ states in D, there must be two distinct strings $x, y \in S$ such that when D is run on x and y, they end up in the same state.

Since all strings in S are distinguishable relative to L, there must be some string w such that $xw \in L$ and $yw \notin L$. Because D is deterministic and D ends in the same state when run on x and y, we know that D must end in the same state when run on xw and yw. If that state is accepting, then D accepts yw, but $yw \notin L$. Otherwise, the state is rejecting, so D rejects xw, but $xw \in L$. In both cases, we contradict that D is a DFA for L.

We have reached a contradiction, so our assumption must have been wrong. Thus if D is a DFA for L, we see D must have at least n states. ■

ii. In Problem Set 5, you build an NFA for the language $L = \{ w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}^* \mid$ the last character of w appears nowhere else in the string, and $|w| \geq 1 \}$. Prove that any DFA for this language must have at least 32 states.

Proof: There are 32 possible subsets of $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$. For each of those subsets, choose one string whose only characters are the characters in that set. Then, let S be a set of all of the strings formed this way.

We claim that any two strings in S are distinguishable relative to L. Let x and y be any two distinct strings in L. Since $x \neq y$, one of these two strings must contain a character the other does not; assume without loss of generality that x contains character c but y does not. Then $xc \notin L$, since c appears somewhere in x, but $yc \in L$ because $|yc| \geq 1$ and c appears nowhere in y.

Since S is a set of 32 strings distinguishable relative to L, by our result from (i) any DFA for L must have at least 32 states. ■

iii. In Problem Set 5, you built an NFA for the language $\{ w \in \{\texttt{0}, \texttt{1}\}^* \mid w$ contains at least two `1`'s with exactly five characters between them. $\}$ Prove that any DFA for this language must have at least 64 states.

Proof: Let $S = \{ w \in \{\texttt{0}, \texttt{1}\}^* \mid |w| = 6 \}$ (that is, the set of all length-six strings made from `0`s and `1`s). There are 64 such strings. We further claim that any pair of strings in S are distinguishable relative to L. Let $x, y \in S$ be two distinct strings. Note that neither x nor y belongs to L, since with only six characters each it is not possible for either string to contain two `1`'s separated by exactly five characters.

Since $x \neq y$, there must be some index i such that x and y have different characters at that index. Without loss of generality, assume the ith character of x is a `0` but the ith character of y is `1`. Now, let w be a string of length five that is all zeros except at position i. Then $xw \notin L$, since no two `1`s in xw are separated by exactly 5 characters, but $yw \in L$ because the `1`s at positions i and $i + 6$ are separated by exactly 5 characters.

Since S is a set of 64 strings distinguishable relative to L, by our result from (i) we know that any DFA for L must have at least 64 states. ■

Why we asked this question: This question was designed to make sure that you understood the intuition behind the Myhill-Nerode theorem. Part (i) specifically requests that you modify the proof of the Myhill-Nerode theorem from lecture to show lower bounds on numbers of states rather than to show that a language was not regular. Parts (ii) and (iii) were then designed to get you thinking about how to find distinguishable strings and to reinforce how powerful a tool nondeterminism is.

Problem Three: Primes are Nonregular (20 Points)

i. Let p be a prime number and let k be some positive natural number. Prove that there is a natural number n for which $p + kn$ is composite.

Proof: Let p be prime and let k be a positive natural number. Then $p + pk = p(1 + k)$. Since k is positive, $1 + k \geq 2$, and therefore $p + pk$ is not prime. ■

ii. Let p_1 and p_2 be prime numbers where $p_1 < p_2$. Prove that there is some value of n for which the number $p_1 + n(p_2 - p_1)$ is prime but $p_2 + n(p_2 - p_1)$ is composite. *(Hint: Use your result from part (i). You might also want to use this fact: if there are any positive natural numbers with some property, there is a smallest positive natural number with some property.)*

Proof: Let n be the smallest natural number such that $p_2 + n(p_2 - p_1)$ is composite. This means, in particular, that $p_2 + (n - 1)(p_2 - p_1)$ is prime. Note that $p_2 + (n - 1)(p_2 - p_1) = p_1 + n(p_2 - p_1)$. Therefore, there is some n such that $p_2 + n(p_2 - p_1)$ is composite and $p_1 + n(p_2 - p_1)$ is prime. ■

iii. Prove that L is not a regular language. *(Hint: There are infinitely many prime numbers.)*

Proof: Consider the set L, which contains infinitely many strings. Now consider two strings distinct strings $\mathbf{a}^{p_1}$, $\mathbf{a}^{p_2} \in L$. By our result from part (ii), there exists some n such that $p_1 + n(p_2 - p_1)$ is prime and $p_2 + n(p_2 - p_1)$ is composite. Therefore, $\mathbf{a}^{p_1}\mathbf{a}^{n(p_2 - p_1)} \in L$ but $\mathbf{a}^{p_2}\mathbf{a}^{n(p_2 - p_1)} \notin L$, so $\mathbf{a}^{p_1}$ and $\mathbf{a}^{p_2}$ are distinguishable relative to L. Therefore, L is an infinite set of strings distinguishable relative to L, so L is not regular. ■

Why we asked this question: We chose this question for a few reasons. First, parts (i) and (ii) of this problem are primarily an exercise in discrete math, which we wanted to make sure you were continuing to work with throughout the quarter. The proof in part (ii), in particular, was designed to revisit proofs by extremal case, which many people were still somewhat confused about after Problem Set Three. Finally, we asked part (iii) to make sure that you were comfortable working with the Myhill-Nerode theorem, as the set of strings to choose is more complex than in other questions.

Problem Four: Designing CFGs (24 Points)

Below are a list of alphabets and languages over those alphabets. For each language, design a context-free grammar that generates that language, then show derivations for the indicated strings.

i. For the alphabet $\Sigma = \{ \mathbf{a}, \mathbf{b}, \mathbf{c} \}$, write a CFG for the language $L = \{ w \in \Sigma^* \mid w \text{ contains } \mathbf{aa} \text{ as a substring} \}$. Show a derivation of the strings **aa**, **abaabc**, and **ccaabb** using your grammar.

One possible CFG is as follows:

$$S \rightarrow A\mathbf{aa}A$$

$$A \rightarrow \mathbf{a}A \mid \mathbf{b}A \mid \mathbf{c}A \mid \varepsilon$$

Here, the nonterminal S begins by placing a **aa** somewhere in the string, surrounded by two copies of A. A then expands out to any string of **a**s, **b**s, and **c**s.

Derivations of the given strings are shown here:

S ⇒ A**aa**A ⇒ **aa**A ⇒ **aa**	S ⇒ A**aa**A ⇒ **a**A**aa**A ⇒ **ab**A**aa**A ⇒ **abaa**A ⇒ **abaab**A ⇒ **abaabc**A ⇒ **abaabc**	S ⇒ A**aa**A ⇒ **c**A**aa**A ⇒ **cc**A**aa**A ⇒ **ccaa**A ⇒ **ccaab**A ⇒ **ccaabb**A ⇒ **ccaabb**

ii. For the alphabet $\Sigma = \{\mathbf{a}, \mathbf{b}\}$, write a CFG for the language $L = \{\ w \in \Sigma^* \mid w \text{ is } \textit{\textbf{not}} \text{ a palindrome}\ \}$. That is, w is not the same when read forwards and backwards, so **aab** $\in L$ and **baabab** $\in L$, but **aba** $\notin L$ and **bb** $\notin L$. Show derivations of **aab** and **abbaba**.

One CFG is given here:

$$S \rightarrow \mathbf{a}S\mathbf{a} \mid \mathbf{b}S\mathbf{b} \mid \mathbf{a}X\mathbf{b} \mid \mathbf{b}X\mathbf{a}$$

$$X \rightarrow \varepsilon \mid \mathbf{a}X \mid \mathbf{b}X$$

This is based on the following observation: a string is not a palindrome if after finding some number of matching symbols on both sides of the string, a mismatched pair is found. S will generate the matching symbols, then the mismatched pair, and then let X generate the middle of the string arbitrarily.

Derivations of the given strings are shown here:

S ⇒ **a**X**b** ⇒ **aa**X**b** ⇒ **aab**	S ⇒ **a**S**a** ⇒ **ab**S**ba** ⇒ **abb**X**aba** ⇒ **abbaba**

iii. For the alphabet $\Sigma = \{\mathbf{a}, \mathbf{b}\}$, write a CFG for the language $L = \{\ w \in \Sigma^* \mid |w| \equiv_4 0$, and the first quarter of the characters in w contains at least one **b** $\}$. For example, $\underline{\mathbf{b}}\mathbf{aaa} \in L$, $\underline{\mathbf{b}}\mathbf{bbb} \in L$, $\underline{\mathbf{ab}}\mathbf{bbbbba} \in L$, $\underline{\mathbf{bbb}}\mathbf{aaabbbaaa} \in L$, $\underline{\mathbf{aba}}\mathbf{bbbbbbbbb} \in L$, but $\underline{\mathbf{a}}\mathbf{bbb} \notin L$, $\varepsilon \notin L$, $\mathbf{b} \notin L$, $\underline{\mathbf{aa}}\mathbf{bbbbaa} \notin L$, and $\underline{\mathbf{aaa}}\mathbf{bbbbbbbbb} \notin L$. (For simplicity, I've underlined the first quarter of the characters in each string). Show a derivation of **baaa**, **abaaaaaa**, and **baaaaaaa**.

One possible grammar is

$$S \rightarrow \mathbf{a}SXXX \mid \mathbf{b}TXXX$$
$$T \rightarrow TXXXX \mid \varepsilon$$
$$X \rightarrow \mathbf{a} \mid \mathbf{b}$$

The idea behind this grammar is that the nonterminal S will repeatedly put an **a** in the first quarter of the string and then three arbitrary symbols at the end of the string. Once we place the **b** in the first quarter of the string, we can then continuously lay down four arbitrary symbols in the middle of the string.

Derivations of the given strings are shown here. For simplicity, I've compressed the steps of expanding out the X's.

S ⇒ **b**TXXX ⇒ **b**XXX $\Rightarrow^*$ **baaa**	S ⇒ **a**SXXX ⇒ **ab**TXXXXXX ⇒ **ab**XXXXXX $\Rightarrow^*$ **abaaaaaa**	S ⇒ **b**TXXX ⇒ **b**TXXXXXXX ⇒ **b**XXXXXXX $\Rightarrow^*$ **baaaaaaa**

iv. Suppose that you live in a one-dimensional world with your house at position 0. One day, you decide to go for a walk by taking steps of size ±1 forward and backward. You begin at your house and, after completing your walk, end up back at your house. Consider the alphabet Σ = {**l**, **r**}. A string in Σ* describes a possible walk where at each step you either move a step left (**l**) or a step right (**r**). For example, the string "**lrrrll**" means that you take a step left, then three steps right, then two steps left. Let *L* = { *w* ∈ Σ* | *w* describes a series of steps in which you arrive at the same place at which you started }. Write a CFG that generates *L*. Show derivations of **llrr**, **rlrl**, and **llrrrrll** using your grammar.

One grammar is as follows:

S → **l**S**r**S | **r**S**l**S | ε

The intuition behind this grammar is that the first symbol of the string (if any) must either by **l** or **r**. If it's **l**, then somewhere else in the string we need to put a **r**, and similarly if the string starts with a **r** there must be a **l** somewhere else.

Derivations of the requested strings are shown here:

S	S	S
⇒ **l**S**r**S	⇒ **r**S**l**S	⇒ **l**S**r**S
⇒ **ll**S**r**S**r**S	⇒ **r**S**lr**S**l**S	⇒ **ll**S**r**S**r**S
⇒ **llr**S**r**S	⇒ **rlr**S**l**S	⇒ **llr**S**r**S
⇒ **llrr**S	⇒ **rlrl**S	⇒ **llrr**S
⇒ **llrr**	⇒ **rlrl**	⇒ **llrrr**S**l**S
		⇒ **llrrr**S**l**
		⇒ **llrrrr**S**l**S**l**
		⇒ **llrrrrl**S**l**
		⇒ **llrrrrll**

Problem Five: Testing Emptiness (16 Points)

Suppose that you have a magic machine that takes as input two CFGs over the same alphabet Σ and returns whether every string generated by the first CFG is also generated by the second CFG. In other words, given two grammars G_1 and G_2 over the same alphabet Σ, the magic machine determines whether $\mathscr{L}(G_1) \subseteq \mathscr{L}(G_2)$. What else could you do with this machine?

i. Suppose you want to test whether G never generates any strings (that is, whether $\mathscr{L}(G) = \emptyset$). Describe how you could use the oracle to decide whether G generates no strings.

Here is one possible procedure. Let E be the CFG $S \to S$. Next, ask the oracle whether $\mathscr{L}(G) \subseteq \mathscr{L}(E)$ and report whatever the oracle says as the answer.

ii. Formally prove that your procedure from (i) is correct by proving the following: your procedure reports that $\mathscr{L}(G) = \emptyset$ iff it's actually true that $\mathscr{L}(G) = \emptyset$.

Proof: First, we prove that $\mathscr{L}(E) = \emptyset$. A simple inductive argument shows that after n steps of any derivation done in E, the string generated will be S. This means that E never generates any strings of terminals, so E never generates any strings at all. Accordingly, $\mathscr{L}(E) = \emptyset$.

Next, we claim that for any set A, that $A \subseteq \emptyset$ iff $A = \emptyset$. For the "only if" step, note that if $A = \emptyset$, then $A \subseteq \emptyset$ because the empty set is a subset of any set. For the "if" step, we proceed by contradiction. Suppose that $A \subseteq \emptyset$ but that $A \neq \emptyset$. Since $A \neq \emptyset$, there must be some x where $x \in A$. But then since $A \subseteq \emptyset$, this means that $x \in \emptyset$, which is impossible. We've reached a contradiction, so our assumption must have been wrong and therefore if $A \subseteq \emptyset$, then $A = \emptyset$.

Now we prove the overall theorem. Note that our procedure reports that $\mathscr{L}(G) = \emptyset$ iff it's the case that $\mathscr{L}(G) \subseteq \mathscr{L}(E)$. Next, note that $\mathscr{L}(G) \subseteq \mathscr{L}(E)$ iff $\mathscr{L}(G) \subseteq \emptyset$. Finally, note that $\mathscr{L}(G) \subseteq \emptyset$ iff $\mathscr{L}(G) = \emptyset$. This means that our procedure reports that $\mathscr{L}(G) = \emptyset$ iff $\mathscr{L}(G)$ indeed is the empty set, as required. ■

Why we asked this question: As with Problem Five on the last problem set, this question is an example of a reduction between problems. By the time you're reading this, we'll probably have started talking about reductions in lecture. Our hope is that this question will help you better understand reductions by giving you concrete examples of working with them in a variety of contexts!

Problem Six: Ambiguous Grammars (16 Points)

In Friday's lecture, you saw the following unambiguous grammar for arithmetic expressions involving addition, subtraction, multiplication, and division:

$$E \rightarrow T \mid T \mathbin{\texttt{+}} E \mid T \mathbin{\texttt{-}} E$$

$$T \rightarrow \texttt{int} \mid \texttt{int} \mathbin{\texttt{*}} T \mid \texttt{int} \mathbin{\texttt{/}} T$$

This question asks you to expand the grammar to incorporate additional operations.

i. Suppose that you want to add exponentiation to the grammar so that you can write expressions like **`int * int ^ int + int * int + int`**. The exponentiation operator **`^`** should bind more tightly than any other operation. Additionally, it should be *right-associative*, so the string **`int ^ int ^ int`** should be interpreted as **`int ^ (int ^ int)`**. Modify the above grammar to add exponentiation that is right-associative. Make sure that your grammar is still unambiguous, and show a parse tree for **`int ^ int ^ int * int + int`**.

Here's one possible new grammar:

$$E \rightarrow T \mid T \mathbin{\texttt{+}} E \mid T \mathbin{\texttt{-}} E$$

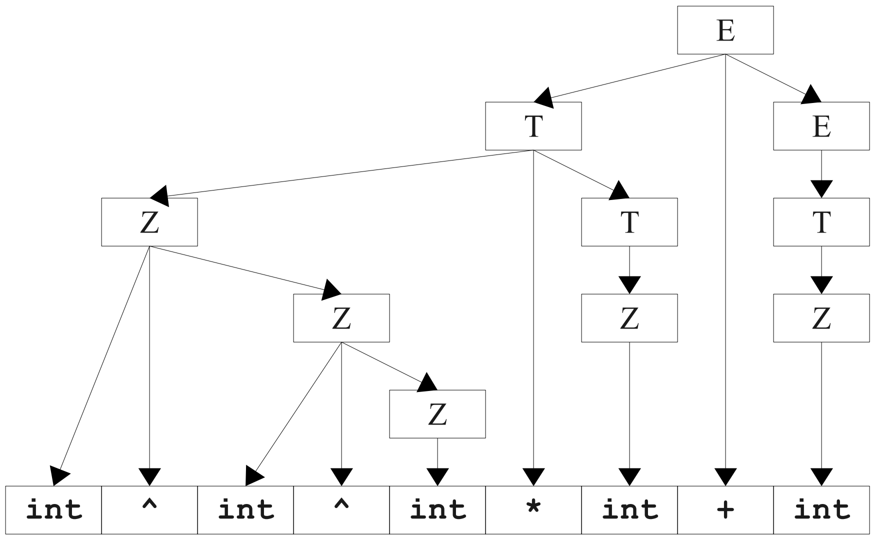
$$T \rightarrow Z \mid Z \mathbin{\texttt{*}} T \mid Z \mathbin{\texttt{/}} T$$

$$Z \rightarrow \texttt{int} \mid \texttt{int} \mathbin{\texttt{^}} Z$$

When we came up with the original grammar, the idea was to assemble the string as a collection of terms added together or subtracted from one another, where each term consisted of products or quotients. The grammar we built reflected this by having "layers," where the top layer handles addition/subtraction and the second layer handles multiplication/division. We're continuing this trend with this grammar by having a third layer below multiplication/division that handles exponentiation.

To enforce that exponentiation is right-associative, we ensure that the Z is on the right side of the recursive rule involving Z. That ensures that the parse tree will always have the operators bind more tightly on the right than on the left.

Here is the parse tree for the given string:



Notice how the exponentiation is right-associative due to the Z term expanding out right-recursively.

ii. Now, suppose you want to allow for explicit parenthesization of arithmetic expressions to indicate operator precedence. For example, **`(int + int) * int`** should be parsed to mean "add the first two **`int`**s, then multiply the result by an integer." Modify the grammar you developed in part (i) to support parenthesized expressions. Make sure that your grammar is still ambiguous, and give a parse tree for **`int ^ (int + int * int) ^ int`**.

Here is one possible grammar:

$$E \to T \mid T \textbf{ + } E \mid T \textbf{ - } E$$

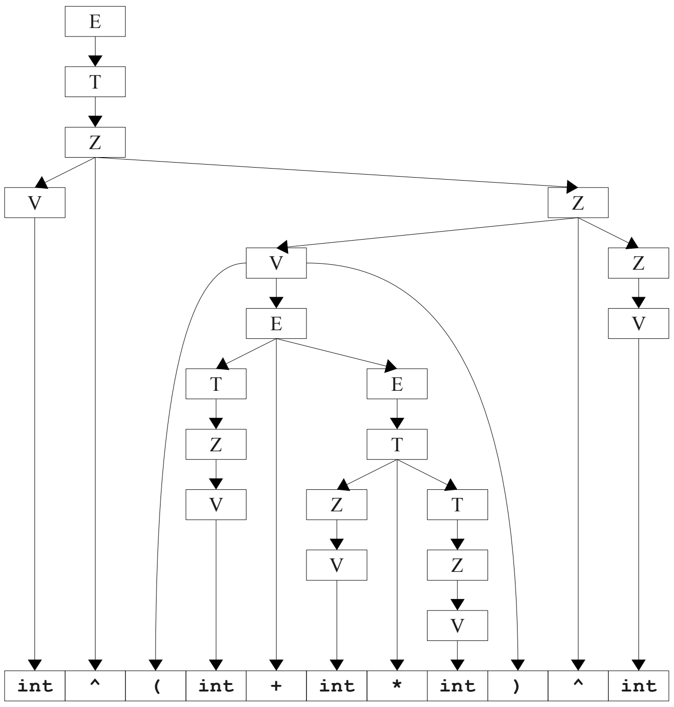
$$T \to Z \mid Z \textbf{ * } T \mid Z \textbf{ / } T$$

$$Z \to V \mid V \textbf{ ^ } Z$$

$$V \to \texttt{int} \mid \texttt{(}E\texttt{)}$$

This again continues our tradition of using layering to enforce precedence. Here, V expands out to "something atomic with respect to the other operators," which here is either an integer or a parenthesized expression.

Here is a parse tree for the given string:



Why we asked this question: Designing unambiguous CFGs is challenging and requires some nuanced constructions. We asked this question to make sure you understood exactly why the unambiguous grammar from lecture worked correctly and to help you see how to extend the construction to design more complex grammars. Finally, we wanted you to draw the parse trees to make sure you had a strong visual intuition for why these grammars were unambiguous.