Problem Set 8

In this problem set, you'll transition away from the regular languages to the context-free languages and to the realm of Turing machines. This will be your first foray beyond the limits of what computers can over hope to accomplish, and we hope that you find this as exciting as we do!

As always, please feel free to drop by office hours or ask on Ed if you have any questions. We'd be happy to help out.

Good luck, and have fun!

Due Friday, November 13th at 12:00PM noon Pacific.

Problem One: Designing CFGs

For each of the following languages, design a CFG for that language. To do so, download the starter files for Problem Set Eight and extract them somewhere convenient. Write and save your answers in the file `res/Grammars.cfgs`. Use the provided program to test and explore your CFG, and submit on Gradescope once you're finished.

As a note – tests for CFGs take longer to run than tests for regular expressions, DFAs, etc. If the test driver is running slowly on your system, set the project to build in *Release* rather than *Debug* mode. To do this, go to the left side of the Qt Creator window and click the picture of a monitor with "Debug" written below it. Then, choose the "Release" option. This disables debugging and turns on optimization, which markedly speeds up the tests.

i. Given Σ = {`a`, `b`, `c`}, write a CFG for the language { $w \in \Sigma^*$ | w contains `aa` as a substring }. For example, the strings `aa`, `baac`, and `ccaabb` are all in the language, but `aba` is not.

ii. In our lecture on regular expressions, we wrote the following regular expression that matched email addresses:

$$\texttt{a}^+\texttt{(.a}^+\texttt{)*@a}^+\texttt{(.a}^+\texttt{)}^+$$

Given Σ = {`@`, `.`, `a`}, write a CFG whose language is the same as the language of this regular expression.

iii. Given Σ = {`a`, `b`}, write a CFG for the language L = { $w \in \Sigma^*$ | w is ***not*** a palindrome }, the language of strings that are not the same when read forwards and backwards. For example, `aab` $\in L$ and `baabab` $\in L$, but `aba` $\notin L$, `bb` $\notin L$, and $\varepsilon \notin L$.

Don't try solving this one by starting with the CFG for palindromes and making modifications to it. In general, there's no way to mechanically turn a CFG for a language L into a CFG for the language $\overline{L}$, since the context-free languages aren't closed under complementation. However, the idea of looking at the first and last characters of a given string might still be a good idea.

iv. Let Σ be an alphabet containing these symbols:

$$\varnothing \quad \mathbb{N} \quad \{ \quad \} \quad , \quad \cup$$

We can form strings from these symbols which represent sets. Here's some examples:

$\varnothing$	$\{\varnothing, \mathbb{N}\} \cup \mathbb{N} \cup \varnothing$	$\{\varnothing\} \cup \mathbb{N} \cup \{\mathbb{N}\}$	$\{\varnothing, \varnothing, \varnothing\}$
$\{\{\mathbb{N}, \varnothing\} \cup \{\varnothing\}\}$	$\mathbb{N} \cup \{\mathbb{N}, \varnothing\}$	$\{\}$	$\{\mathbb{N}\}$
$\{\varnothing, \{\varnothing, \{\varnothing\}\}\}$	$\{\{\{\{\mathbb{N}\}\}\}\}$	$\mathbb{N}$	$\{\varnothing, \{\}\}$

Notice that some of these sets, like $\{\varnothing, \varnothing\}$ are syntactically valid but redundant, and others like $\{\}$ are syntactically valid but not the cleanest way of writing things. Here's some examples of strings that don't represent sets or aren't syntactically valid:

ε	$\}\varnothing\{$	$\varnothing\{\mathbb{N}\}$	$\{\{\}$
$\mathbb{N}, \varnothing, \{\varnothing\}$	$\{, \mathbb{N}\}$	$\{ \mathbb{N} \varnothing \},$	$\{,\}$
$\{\varnothing$	$\}\} \mathbb{N}$	$\{ \varnothing, \varnothing, \varnothing, \}$	$\{\mathbb{N}, , , \varnothing\}$

Write a CFG for the language { $w \in \Sigma^*$ | w is a syntactically valid string representing a set }. ***Please use the letters `n`, `u`, and `o` in place of $\mathbb{N}$, $\cup$, and $\varnothing$, respectively***.

Fun fact: the starter files for Problem Set One contain a parser that's designed to take as input a string representing a set and to reconstruct what set that is. The logic we wrote to do that parsing was based on a CFG we wrote for sets and set theory. Take CS143 if you're curious how to go from a grammar to a parser!

As a hint, as is often the case when writing CFGs, we recommend that you use different nonterminals to represent different components of the string. For example, structure of a comma-separated list of sets is different than the structure of an expression representing a single set.

Problem Two: The Complexity of Addition

This problem explores the following question:

How hard is it to add two numbers?

Suppose that we want to check whether $x + y = z$, where x, y, and z are all natural numbers. If we want to phrase this as a problem as a question of strings and languages, we will need to find some way to standardize our notation. In this problem, we will be using the ***unary number system***, a number system in which the number n is represented by writing out n 1's. For example, the number 5 would be written as `11111`, the number 7 as `1111111`, and the number 12 as `111111111111`.

Given the alphabet $\Sigma = \{$`1`, `+`, `=`$\}$, we can consider strings encoding $x + y = z$ by writing out x, y, and z in unary. For example:

4 + 3 = 7 would be encoded as `1111+111=1111111`

7 + 1 = 8 would be encoded as `1111111+1=11111111`

0 + 1 = 1 would be encoded as `+1=1`

Consider the alphabet $\Sigma = \{$`1`, `+`, `=`$\}$ and the following language, which we'll call *ADD*:

$$\{\ 1^m+1^n=1^{m+n} \mid m, n \in \mathbb{N}\ \}$$

For example, the strings `111+1=1111` and `+1=1` are in the language, but `1+11=11` is not, nor is the string `1+1+1=111`.

i. Prove or disprove: the language *ADD* defined above is regular.

ii. Write a context-free grammar for *ADD* in `res/Grammars.cfgs`, showing that *ADD* is context-free.

You may find it easier to solve this problem if you first build a CFG for this language where you're allowed to have as many numbers added together as you'd like. Once you have that working, think about how you'd modify it so that you have exactly two numbers added together on the left-hand side of the equation.

Problem Three: The Complexity of Pet Ownership

This problem explores the following question:

How hard is it to walk your dog without a leash?

Let's imagine that you're going for a walk with your dog, but this time don't have a leash. As in Problem Set Six and Problem Set Seven, let $\Sigma = \{$`y`, `d`$\}$, where `y` means that you take a step forward and `d` means that your dog takes a step forward. A string in Σ^* can be thought of as a series of events in which either you or your dog moves forward one unit. For example, the string `yydd` means that you take two steps forward, then your dog takes two steps forward.

Let *DOGWALK* = $\{$ $w \in \Sigma^*$ | w describes a series of steps where you and your dog arrive at the same point $\}$. For example, the strings `yyyddd`, `ydyd`, and `yyyddddddyyy` are all in *DOGWALK*.

i. Prove or disprove: the language *DOGWALK* defined above is regular.

ii. Write a context-free grammar for *DOGWALK* in `res/Grammars.cfgs`, showing that *DOGWALK* is context-free.

Check the lecture slides on CFGs for examples of grammars that don't work, and make sure you can articulate why those grammars are incorrect.

Problem Four: Equivalence Classes and Regular Languages

The Myhill-Nerode theorem is based on the distinguishability relation $\not\equiv_L$. A closely related binary relation is the ***indistinguishability*** relation for L, denoted $\equiv_L$. It's also a binary relation over Σ^*, and its definition is the negation of the one for distinguishability:

$$x \equiv_L y \quad \text{if} \quad \forall w \in \Sigma^*. (xw \in L \leftrightarrow yw \in L).$$

Amazingly, this is always an equivalence relation, regardless of what L is!

i. Prove that if L is a language over Σ, then $\equiv_L$ is an equivalence relation over Σ^*.

This proof will look a lot like the ones from Problem Set 3, except with more strings. So proceed slowly and methodically, don't use first-order logic in your proofs, etc.

Let's make this more concrete. Let $\Sigma = \{$a, b$\}$ and consider the language $M = \{ w \in \Sigma^* \mid$ the number of b's in w is congruent to 1 modulo 5 or to 3 modulo 5 $\}$. For example, a**b**a $\in M$, **b**aaa**b**aaa**b** $\in M$, and **bbbbbb** $\in M$, but aa $\notin M$ and a**bb**a $\notin M$.

ii. Fill in the blanks below to list all the equivalence classes of $\equiv_M$. We've given you exactly the number of blanks that you'll need to do this. No justification is required.

$[\_\_\_\_\_\_]_{\equiv M} = \{ w \in \Sigma^* \mid \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \}$

$[\_\_\_\_\_\_]_{\equiv M} = \{ w \in \Sigma^* \mid \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \}$

$[\_\_\_\_\_\_]_{\equiv M} = \{ w \in \Sigma^* \mid \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \}$

$[\_\_\_\_\_\_]_{\equiv M} = \{ w \in \Sigma^* \mid \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \}$

$[\_\_\_\_\_\_]_{\equiv M} = \{ w \in \Sigma^* \mid \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \}$

You might have noticed that each equivalence class of $\equiv_M$ either consists of a bunch of strings not in M or of a bunch of strings that are in M. That's not a coincidence!

iii. Let L be a language over some alphabet Σ and let $x \in \Sigma^*$ be some string. Prove that either *every string* in $[x]_{\equiv L}$ is in L or that *no strings* in $[x]_{\equiv L}$ are.

The ***index*** of an equivalence relation R, denoted $\boldsymbol{I(R)}$, is the number of equivalence classes of R. This quantity might be finite, or it might be an infinite cardinality like $\aleph_0$, or even one of the infinities bigger than that. Armed with the idea of an index, we can state a powerful theorem about finite automata:

Theorem: Let L be a language over Σ. Then if $I(\equiv_L)$ is infinite, L is not regular, and if $I(\equiv_L)$ is finite, then every DFA for L has at least $I(\equiv_L)$ states.

In other words, there's a connection between the number of equivalence classes of a particular binary relation and the minimum sizes of DFAs for that language!

iv. Prove the above theorem. Feel free to use the ***axiom of choice***, which says that every equivalence relation has at least one system of representatives.

Proving this theorem is mostly an exercise in connecting together ideas you've seen used in other places. Think about the relationship between indices and systems of representatives, between distinguishability and indistinguishability, and between what you're doing here and what you've done on the previous problem set.

There's a lovely intuition for this theorem. You can think of the indistinguishability relation for a language L as pinning down the idea "a DFA for L can't tell the difference between these two strings." If you think back to our intuition behind DFA design – build a DFA where each state keeps track of some different piece of information – then you can think of $I(\equiv_L)$ as capturing the number of different pieces of information you'd need to remember. The theorem then says that if you want to build a DFA for a language L, you'll need at least one state per piece of information.

Problem Five: What Does it Mean to Solve a Problem?

Let L be a language over Σ and M be a TM with input alphabet Σ. Here are three potential traits of M:

1. M halts on all inputs.
2. For any string $w \in \Sigma^*$, if M accepts w, then $w \in L$.
3. For any string $w \in \Sigma^*$, if M rejects w, then $w \notin L$.

At some level, for a TM to claim to solve a problem, it should have at least some of these properties. Interestingly, though, just having two of these properties doesn't say much.

i. Prove that if L is any language over Σ, then there is a TM M that satisfies properties (1) and (2).

You can describe your TM in one of many ways. First, you could draw a picture of the TM, along the lines of the ones we've done in class. Second, since we know that TMs are equivalent to computer programs, you could write some short pseudocode showing off how the TM works.

The whole point of this problem is to show that you have to be extremely careful about how you define "solving a problem," since if you define it incorrectly then you can "solve" a problem in a way that bears little resemblance to what we'd think of as solving a problem. Keep this in mind as you work through this one.

ii. Prove that if L is any language over Σ, then there is a TM M that satisfies properties (1) and (3).

iii. Prove that if L is any language over Σ, then there is a TM M that satisfies properties (2) and (3).

iv. Suppose L is a language over Σ for which there is a TM M that satisfies all of properties (1), (2), and (3). What can you say about L? Prove it.

Optional Fun Problem: TMs and Regular Languages

Let M be a TM with the following property: there exists a natural number k such that after M is run on any string w, M always halts after at most k steps. (One "step" corresponds to following a transition in the TM, which consists of writing a symbol, moving the tape head, and changing state.)

Prove that $\mathscr{L}(M)$ is regular.