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 Piazza if you have any questions. We'd be happy to help out.

Good luck, and have fun!

Due Friday, March 9th at 2:30PM.

Problem One: Designing CFGs

For each of the following languages, design a CFG for that language. ***Please use our online tool to design, test, and submit the CFGs in this problem***. To use it, visit the CS103 website and click the "CFG Editor" link under the "Resources" header. You should only have one member from each team submit your grammars; tell us who this person is when you submit the rest of the problems through GradeScope.

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

ii. Given $\Sigma = \{$a, b$\}$, write a CFG for the language $\{ w \in \Sigma^* \mid w$ 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 be a good idea.

iii. Let Σ be an alphabet containing these symbols:

Ø ℕ { } , ∪

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

Ø	{Ø, ℕ} ∪ ℕ ∪ Ø	{Ø} ∪ ℕ ∪ {ℕ}	{Ø, Ø, Ø}
{{ℕ, Ø} ∪ {Ø}}	ℕ ∪ {ℕ, Ø}	{}	{ℕ}
{Ø, {Ø, {Ø}}}	{{{{ℕ}}}}	ℕ	{Ø, {}}

Notice that some of these sets, like {Ø, Ø} 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	}Ø{	Ø{ℕ}	{{}
ℕ, Ø, {Ø}	{, ℕ}	{ ℕ Ø },	{,}
{Ø	}} ℕ	{ Ø, Ø, Ø, }	{ℕ, , , Ø}

Write a CFG for the language $\{ w \in \Sigma^* \mid w$ is a syntactically valid string representing a set $\}$. When using the CFG tool, please use the letters n, u, and o in place of ℕ, ∪, and Ø, 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!

Test your CFG thoroughly! In Fall 2017, a quarter of the submissions we received weren't able to derive the string {Ø, Ø, Ø}.

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*:

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

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*, showing that *ADD* is context-free. (Please submit your CFG online.)

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

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

ii. Write a context-free grammar for *DOGWALK*, showing that *DOGWALK* is context-free. (Please submit your CFG online.)

Be careful, and test your CFG! As you saw in lecture, a lot of ideas that seem plausible here don't work.

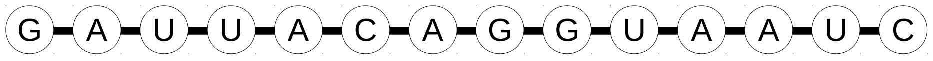
Problem Four: The Complexity of RNA Hairpins

RNA strands consist of strings of *nucleotides*, molecules which encode genetic information. Computational biologists typically represent each RNA strand as a string made from four different letters, `A`, `C`, `G`, and `U`, each of which represents one of the four possible nucleotides.

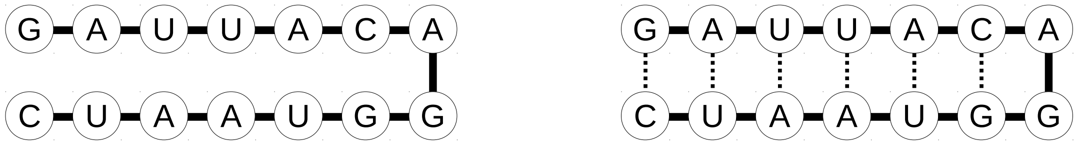
Each of the the four nucleotides has an affinity for a specific other nucleotide. Specifically:

`A` has an affinity for `U` (and vice-versa) `C` has an affinity for `G` (and vice-versa)

This can cause RNA strands to fold over and bind with themselves. Consider this RNA strand:



If you perfectly fold this RNA strand in half, you get the following:



Notice that each pair of nucleotides – except for the `A` and the `G` on the far right – are attracted to the corresponding nucleotide on the other side of the RNA strand. Because of the natural affinities of the nucleotides in the RNA strand, the RNA strand will be held in this shape. This is an example of an *RNA hairpin*, a structure with important biological roles.

For the purposes of this problem, we'll say that an RNA strand forms a hairpin if

- it has even length (so that it can be cleanly folded in half);
- it has length at least ten (there are at least four pairs holding the hairpin shut); and
- all of its nucleotides, except for the middle two, have an affinity for its corresponding nucleotide when folded over. (The middle two nucleotides in a hairpin might coincidentally have an affinity for one another, but it's not required. For example, `CCCC`$\underline{\texttt{AU}}$`GGGG` forms a hairpin.)

This problem explores the question

How hard is it to determine whether an RNA strand forms a hairpin?

Let Σ = {`A`, `C`, `G`, `U`} and let L_{RNA} = { $w \in \Sigma^*$ | w represents an RNA strand that forms a hairpin }. For example, the strings `UGACCCGUCA`, `GUACAAGUAC`, `UUUUUUUUUAAAAAAAAA`, and `CCAACCUUGG` are all in L_{RNA}, but the strings `AU`, `AAAAACUUUUU`, `GGGC`, and `GUUUUAAAAG` are all not in L_{RNA}.

i. Prove that L_{RNA} is not regular. Since this language imposes a lot of requirements on the strings it contains, if in the course of your proof you want to claim that a particular string is or is not in L_{RNA}, please articulate clearly why the string does or does not meet each of the requirements of strings in L_{RNA}.

There's a good amount of trial and error required here. Test your proof carefully – pick concrete examples of strings from your set and make sure that your argument really does work for them. The fact that the two middle characters don't have to match makes this a little bit trickier than you might initially suspect.

ii. Design a CFG for L_{RNA}, which proves that the language is context-free. Please submit your grammar online.

Problem Five: Right-Linear Grammars

A context-free grammar is called a ***right-linear grammar*** if every production in the grammar has one of the following three forms:

- A → ε
- A → B, where B is a nonterminal.
- A → aB, where a is a terminal and B is a nonterminal.

For example, the following is a right-linear grammar:

A → aB | bB | ε

B → aC | bA | C

C → bA | aA | ε

The right-linear grammars are all context-free grammars, so their languages are all context-free. However, it turns out that this class of grammars precisely describe the regular languages. That is, a language L is regular if and only if there is a right-linear grammar G such that $L = \mathscr{L}(G)$.

i. Let G be a right-linear grammar. Describe how to start with G and construct an NFA N such that $\mathscr{L}(G) = \mathscr{L}(N)$. Briefly justify why N accepts precisely the strings that G can generate. To illustrate your construction, show the NFA you'd build from the following right-linear grammar:

   A → aB | bC

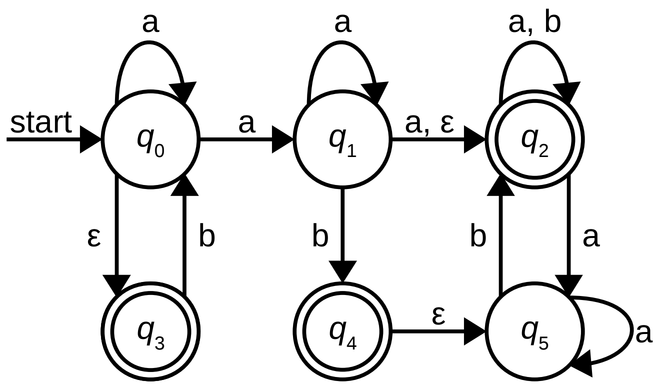
   B → aB | ε

   C → aD | A | bC

   D → aD | bD | ε

   Please submit your NFA for this question through our DFA/NFA editor.

ii. Let N be an NFA. Describe how to start with N and construct a right-linear grammar G such that $\mathscr{L}(G) = \mathscr{L}(N)$. Briefly justify why N accepts precisely the strings that G can generate. To illustrate your construction, show the grammar that you'd build from the following NFA:



   Please submit your right-linear grammar through our online CFG tool.

We're expecting you to actually apply your construction to this NFA, not to eyeball the NFA and find a grammar that happens to have the same language. The point of doing this is to make sure that you've thought through all of the necessary cases.

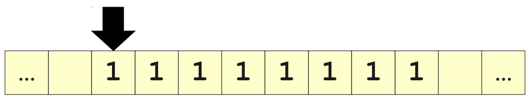
Problem Six: The Collatz Conjecture

The *Collatz conjecture* is a famous conjecture (an unproved claim) that says the following procedure (called the *hailstone sequence*) terminates for all positive natural numbers n:

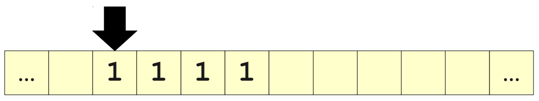
- If $n = 1$, stop.
- If n is even, set $n = n / 2$ and repeat from the top.
- If n is odd, set $n = 3n + 1$ and repeat from the top.

Let $L = \{ \texttt{1}^n \mid n \geq 1$ and the hailstone sequence terminates for $n \}$ be a language over the singleton alphabet $\Sigma = \{\texttt{1}\}$. It turns out that it's possible to build a TM for this language, which means that $L \in \textbf{RE}$, and in this problem you'll do just that. Parts (i) and (ii) will ask you to design two useful subroutines, and you'll assemble the overall machine in part (iii).

i. Design a TM subroutine that, given a tape holding a string of the form $\texttt{1}^{2n}$ (where $n \in \mathbb{N}$) surrounded by infinitely many blanks, ends with $\texttt{1}^n$ written on the tape, surrounded by infinitely many blanks. You can assume the tape head begins reading the first `1` (or points to an arbitrary blank cell in the case where $n = 0$), and your TM should end with the tape head reading the first `1` of the result (or any blank cell if $n = 0$). For example, given the initial configuration



your TM subroutine would end with this configuration:



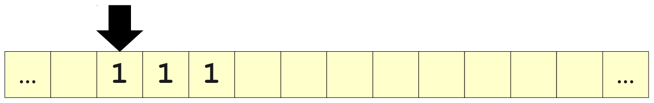
You can assume that there are an even number of `1`s on the tape at startup and can have your TM behave however you'd like if this isn't the case. Please use our provided TM editor to design, develop, test, and submit your answer to this question. Since our TM tool doesn't directly support subroutines, just have your machine accept when it's done.

For reference, our solution has fewer than 10 states. If you have significantly more than this and are struggling to get your TM working, you might want to change your approach. It's totally fine if you have a bunch of states, provided that your solution works.

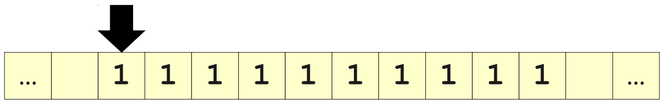
There are a lot of different solutions here. Some use very little extra tape. Some use a lot of extra tape. Some don't need any other tape symbols. Some do. Be creative, try things out, and don't be afraid to back up and try something else if your approach doesn't seem to be panning out.

(Continued on the next page)

ii. Design a TM subroutine that, given a tape holding a string of the form $\mathtt{1}^n$ (for some $n \in \mathbb{N}$), surrounded by infinitely many blanks, ends with $\mathtt{1}^{3n+1}$ written on the tape, surrounded by infinitely many blanks. You can assume that the tape head begins reading the first `1`, and your TM should end with the tape head reading the first `1` of the result. For example, given this configuration



your TM subroutine would end with this configuration:



You can assume the number of `1`s on the tape at startup is odd and can have your TM behave however you'd like if this isn't the case. Please use our provided TM editor to design, develop, test, and submit your answer to this question. Since our TM tool doesn't directly support subroutines, just have your machine accept when it's done. *(For reference, our solution has fewer than 10 states. If you have significantly more than this, you might want to change your approach.)*

iii. Draw the state transition diagram for a Turing machine M that recognizes L. Our TM tool is configured for this problem so that you can use our reference solutions for parts (i) and (ii) as subroutines in your solution. To do so, follow these directions:

1. Create states named `half`, `half_`, `trip`, and `trip_`.
2. To execute the subroutine that converts $\mathtt{1}^{2n}$ into $\mathtt{1}^n$, transition into the state named `half`. When that subroutine finishes, the TM will automagically jump into the state labeled `half_`. You do not need to – and should not – define any transitions into `half_` or out of `half`.
3. To execute the subroutine that converts $\mathtt{1}^n$ into $\mathtt{1}^{3n+1}$, transition into the state named `trip`. When that subroutine finishes, the TM will automagically jump into the state labeled `trip_`. You do not need to – and should not – define any transitions into `trip_` or out of `trip`.

Calling ahead to Monday's lecture: a TM M recognizes a language L if M accepts all of the strings in L and either rejects or loops on all strings that are not in L. In other words, your TM should accept every string in L, and for any string not in L it can either loop infinitely or reject the string.

Please use our provided TM editor to design, develop, test, and submit your answer to this question. *(For reference, our solution has fewer than 15 states. If you have significantly more than this, you might want to change your approach.)*

Problem Seven: TMs, Formally

Just as it's possible to formally define a DFA using a 5-tuple, it's possible to formally define a TM as an 8-tuple $(Q, \Sigma, \Gamma, B, q_0, Y, N, \delta)$ where

- Q is a finite set of states, which can be anything;
- Σ is a finite, nonempty set called the ***input alphabet***;
- Γ is a finite, nonempty set called the ***tape alphabet***, where $\Sigma \subseteq \Gamma$;
- $B \in \Gamma - \Sigma$ is the ***blank symbol***;
- q_0 is the start state, where $q_0 \in Q$;
- $Y \subseteq Q$ is the set of ***accepting states***;
- $N \subseteq Q$ is the set of ***rejecting states***, where $Y \cap N = \emptyset$; and
- δ is the ***transition function***, described below.

This question explores some aspects of the definition.

i. Is it possible to have a TM with no states? Justify your answer.

ii. Is it possible to have a TM with no *accepting* states? Justify your answer.

iii. Is it possible to have a TM with no *rejecting* states? Justify your answer.

iv. Why is the restriction $Y \cap N = \emptyset$ there? Justify your answer.

v. Is it possible to have a TM where $\Sigma = \Gamma$? Justify your answer.

Now, let's talk about the transition function. As with DFAs, the transition function of a Turing machine is what formally defines the transitions. If q is a state in a TM that isn't an accepting state or a rejecting state and a is a symbol that can appear on the TM's tape, then

$$\delta(q, a) = (r, b, D)$$

where r is the new state to transition into, b is the symbol to write back to the tape, and D is either L for "move left" or R for "move right." Because TMs immediately stop running after entering an accepting or rejecting state, the δ function should not be defined for any state q that's either accepting or rejecting. Aside from this, δ should be defined for every combination of a (non-accepting, non-rejecting) state q and any symbol a that can appear on the tape.

vi. Based on the above description of δ, what should the domain of δ be? What should it codomain be? Answer this question by filling in the following blanks, and briefly justify your answer.

$$\delta : \underline{\hspace{6em}} \to \underline{\hspace{6em}}$$

Problem Eight: Regular and Decidable Languages

In class, we alluded to the fact that **REG** (the class of all regular languages) is a subset of **R** (the class of all decidable languages), but we never actually justified this claim.

Describe a construction that, given a DFA D, produces a decider D' where $\mathscr{L}(D) = \mathscr{L}(D')$. Briefly justify why the TM D' you construct is a decider and why it accepts precisely the strings that D accepts. Illustrate your example by applying it to a small DFA D of your choice.

Although you have a formal 5-tuple definition of a DFA and a formal 8-tuple definition of a TM at your disposal, we're not expecting you to write your solution at that level of detail.

Remember that DFAs and TMs work completely differently with regards to accepting and rejecting states and that the transitions in TMs have a very different structure than the transitions in DFAs!

Problem Nine: Jumbled Jargon

(We will cover the material necessary to solve this problem on Monday.)

We've introduced a number of terms and definitions pertaining to Turing machines, languages, and what it means to solve a problem. Some of the terms we've described are adjectives that can only describe TMs, while others are adjectives that can only describe languages. Using them incorrectly leads to statements that aren't mathematically meaningful.

To reason by analogy, consider the statement "the set $\mathbb{N}$ is even." This statement isn't meaningful, because "even" can only be applied to individual natural numbers, and $\mathbb{N}$ isn't a natural number. Similarly, the statement $1 \in 5$ isn't meaningful, since 5 isn't a set. The statement $\mathbb{Z} \subseteq \mathbb{N}$ is meaningful but not true – it's the mathematical equivalent of a grammatically correct statement that just happens to be false.

Below is a series of statements. For each statement, decide whether that statement is mathematically meaningful or not. If it's not mathematically meaningful, explain why not. If it is mathematically meaningful, determine whether it's true or false and briefly justify your answer.

i. If M is a Turing machine, w is a string, and M accepts w, then A_{TM} accepts $\langle M, w \rangle$.

ii. If M is a Turing machine, w is a string, and M loops on w, then $\langle M, w \rangle \notin \mathscr{L}(U_{TM})$.

iii. U_{TM} is decidable.

iv. $\langle U_{TM} \rangle$ is decidable.

v. $\{\langle U_{TM} \rangle\}$ is decidable.

You have a ton of experience type-checking things by this point in the quarter. Use that intuition here.

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

Let L be a language over Σ and M be a TM with input alphabet Σ. Below are three properties that may hold for 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).

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

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.

Problem Eleven: R and RE Languages

The following problems are designed to explore some of the nuances of how Turing machines, languages, decidability, and recognizability all relate to one another. We hope that by working through them, you'll get a much better understanding of key computability concepts.

i. Give a TM M such that $\mathscr{L}(M) \in \mathbf{R}$, but M is not a decider (you can draw a concrete example of TM, or give pseudocode for a program along the lines of what we've done in class). Briefly justify your answer. This shows that just because a TM's language is decidable, it's not necessarily the case that the TM itself must be a decider.

ii. Only *languages* can be decidable or recognizable; there's no such thing as an "undecidable string" or "unrecognizable string." Prove that for every string w, there's an **R** language containing w and an **RE** language containing w.

iii. Here's a weird one. Let Σ be an alphabet containing all characters that can appear in a person's name. Prove that the following language L_{2020} is decidable, subject to the assumption that there is a single US presidential election in 2020 and that it ends with a single winner:

$$L_{2020} = \{\ w \in \Sigma^* \mid w \text{ is the name of the winner of the 2020 presidential election }\}$$

Then, explain how it's possible to build a decider for the language L_{2020} given that no one has any idea who is going to win the 2020 election!

Optional Fun Problem: TMs and Regular Languages (1 Point Extra Credit)

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