

## 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 ever 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 8<sup>th</sup> 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  $\{w \in \Sigma^* \mid w \text{ contains } aa \text{ as a substring}\}$ . 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  $\{w \in \Sigma^* \mid w \text{ is not a palindrome}\}$ , 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  $\bar{L}$ , 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.*

- iii. Let  $\Sigma$  be an alphabet containing these symbols:

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

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

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

Notice that some of these sets, like  $\{\emptyset, \emptyset\}$  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:

$\varepsilon$	$\}\emptyset\{$	$\emptyset\{\mathbb{N}\}$	$\{\{\}$
$\mathbb{N}, \emptyset, \{\emptyset\}$	$\{\mathbb{N}\}$	$\{\mathbb{N} \emptyset\}$ ,	$\{\}$
$\{\emptyset$	$\}\}\mathbb{N}$	$\{\emptyset, \emptyset, \emptyset, \}$	$\{\mathbb{N}, , , \emptyset\}$

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

*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 is very different from the structure of an expression combining multiple sets together.*

## 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*, 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  $\Sigma^*$  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  $DOGWALK = \{ w \in \Sigma^* \mid w \text{ describes a series of steps where you and your dog arrive at the same point} \}$ . 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*, showing that *DOGWALK* is context-free. (Please submit your CFG online.)

*Be careful, 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

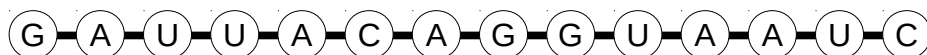
RNA strands consist of strings of *nucleotides*, molecules which encode genetic information. Computational biologists typically represent each RNA strand as a string made from four different letters, A, C, G, and U, each of which represents one of the four possible nucleotides.

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, CCCCAUGGGG forms a hairpin.)

This problem explores the question

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

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

Design a CFG for  $L_{RNA}$ , which proves that the language is context-free. Please submit your grammar online. (This language turns out to not be regular, though the proof of that result using the Myhill-Nerode theorem is heavy on details and light on the intuition, so we won't ask you to do that here.)

## Problem Five: Equivalence Classes and Regular Languages, Part Two

On Problem Set Seven, you explored the *indistinguishability* relation for  $L$ , denoted  $\equiv_L$ , defined as

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

You specifically proved that for any language  $L$ , the relation  $\equiv_L$  is an equivalence relation and that any DFA for  $L$  must have at least  $I(\equiv_L)$  states. In this problem, you're going to prove an amazing result:

**Theorem:** If  $L$  is a language where  $I(\equiv_L)$  is finite, then  $L$  is regular.

In other words, if you know absolutely nothing about a language other than there are finitely many equivalence classes of the  $\equiv_L$  relation, then somewhere out there, there must be a DFA for  $L$ !

Let  $L$  be an arbitrary language over some alphabet  $\Sigma$  where  $I(\equiv_L)$  is finite. We are going to prove that  $L$  is regular by defining a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$  for this language  $L$ . The key insight behind this proof is how to choose  $Q$ . Specifically, we will choose  $Q$  to be the set of equivalence classes of  $\equiv_L$ :

$$Q = \{ [w]_{\equiv_L} \mid w \in \Sigma^* \}.$$

It might seem strange to have the states of a DFA be sets, but then again, you've seen something like this before. In Problem Set Six, when working through the subset construction, you created a DFA whose states literally were sets of states of some particular NFA.

- i. Explain why  $Q$  is finite. This should take you at most a sentence or two.

We now need to figure out how to pick a start state and wire up our transitions. Our goal will be to define  $q_0$  and  $\delta$  so that our DFA has the following property: if you run  $w$  through this DFA, the state you end up in corresponds to  $[w]_{\equiv_L}$ . It turns out that choosing  $q_0$  and  $\delta$  as follows makes this work:

$$q_0 = [\varepsilon]_{\equiv_L} \qquad \delta([x]_{\equiv_L}, a) = [xa]_{\equiv_L}.$$

Of course, you shouldn't take our word for it. You should prove that these choices make everything work!

- ii. Prove that for any string  $w \in \Sigma^*$ , we have  $\delta^*(w) = [w]_{\equiv_L}$ .

*Need a refresher on the definition of  $\delta^*$ ? Check Problem Set Seven.*

To seal the deal, we need to choose our set of accepting states. We'll define  $F$  as follows:

$$F = \{ [w]_{\equiv_L} \mid \exists x \in [w]_{\equiv_L}. x \in L \}.$$

In other words,  $F$  is the set of all equivalence classes containing at least one string in  $L$ .

- iii. On Problem Set Seven, you saw that we can formally define  $\mathcal{L}(D) = \{ w \in \Sigma^* \mid \delta^*(w) \in F \}$ . Prove that with this choice of  $F$ , we have  $\mathcal{L}(D) = L$ .

*There is a ton of formal notation here, but at the end of the day, this question is just asking you to prove that two sets are equal. Think way back to Problem Set One. What's the easiest way to do this?*

*Your proof should use the formal definitions provided here rather than higher-level concepts like "the DFA accepts  $w$ " or "run the DFA on  $w$ ." Also, perhaps a result from Problem Set Seven would be useful here?*

By combining the two theorems you've explored about indistinguishability – the one you proved last time, and the one from above – we get this fundamental result:

**Theorem (Myhill-Nerode):** A language  $L$  is regular if and only if  $I(\equiv_L)$  is finite.

Furthermore, if  $I(\equiv_L)$  is finite, the smallest possible DFA for  $L$  has exactly  $I(\equiv_L)$  states.

This result formalizes the intuition we've had about regular languages corresponding to problems you

can solve with only finite memory. The “memory” you need corresponds to remembering which equivalence class the string you’ve seen so far happens to fall into.

If you talk to CS theory folk and mention “the Myhill-Nerode theorem,” they’ll assume you’re talking about the above theorem! The version we saw in lecture is just a special case of this more general one.

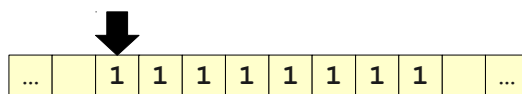
## Problem Six: The Collatz Conjecture

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  $n$ :

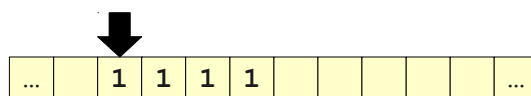
- If  $n = 1$ , stop.
- If  $n$  is even, set  $n = n / 2$  and repeat from the top.
- If  $n$  is odd, set  $n = 3n + 1$  and repeat from the top.

Let  $L = \{ 1^n \mid n \geq 1 \text{ and the hailstone sequence terminates for } n \}$  be a language over the singleton alphabet  $\Sigma = \{1\}$ . It turns out that it's possible to build a TM for this language, which means that  $L \in \mathbf{RE}$ , 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).

- Design a TM subroutine that, given a tape holding a string of the form  $1^{2n}$  (where  $n \in \mathbb{N}$ ) surrounded by infinitely many blanks, ends with  $1^n$  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:



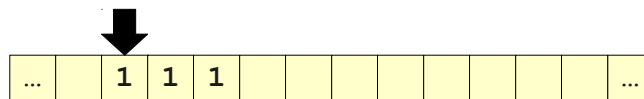
You can assume that there are an even number of 1s on the tape at startup and can have your TM behave however you'd like if this isn't the case. Please use our provided TM editor to design, develop, test, and submit your answer to this question. Since our TM tool doesn't directly support subroutines,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

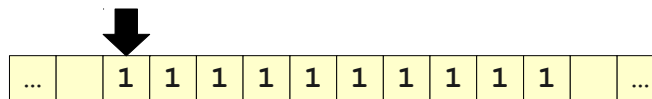
*There are a lot of different solutions here. Some use very little extra tape. Some use a lot of extra tape. Some don't need any other tape symbols. Some do. Be creative, try things out, and don't be afraid to back up and try something else if your approach doesn't seem to be panning out.*

*(Continued on the next page...)*

- ii. Design a TM subroutine that, given a tape holding a string of the form  $1^n$  (for some  $n \in \mathbb{N}$ ), surrounded by infinitely many blanks, ends with  $1^{3n+1}$  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 1s 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 and are struggling to get things working, 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  $1^{2n}$  into  $1^n$ , transition into the state named `half`. When that subroutine finishes, the TM will automatically 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  $1^n$  into  $1^{3n+1}$ , transition into the state named `trip`. When that subroutine finishes, the TM will automatically jump into the state labeled `trip_`. You do not need to – and should not – define any transitions into `trip_` or out of `trip`.

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 and are struggling to get things working, you might want to change your approach. For those of you who did the Hailstone Sequence problem in CS106A – you're now solving the same problem using pure mathematics! Did you expect you'd ever get to do something like that?*

*Because TMs can go into infinite loops, our provided TM simulator will only simulate TMs for some fixed, large number of steps. Some inputs to your TM might take a very long time to complete purely because the Hailstone Sequence takes a long time to complete in those cases. When that happens, you'll get timeouts reported. It's probably nothing to worry about if you're seeing timeouts for very large or long strings, but you shouldn't be seeing timeouts for, say, strings whose lengths are between 5 and 10.*

## 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;
- $\Sigma$  is a finite, nonempty set called the **input alphabet**;
- $\Gamma$  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
- $\delta$  is the **transition function**, described below.

Remember that the definition is the official arbiter of what is legal and what isn't, so if the definition doesn't preclude something, it's legal regardless of how counterintuitive or weird it might be. 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  $\delta$  function should not be defined for any state  $q$  that's either accepting or rejecting. Aside from this,  $\delta$  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  $\delta$ , what should the domain of  $\delta$  be? What should its codomain be? Answer this question by filling in the following blanks, and briefly justify your answer.

$$\delta : \underline{\hspace{2cm}} \rightarrow \underline{\hspace{2cm}}$$

*In class, we said that any missing transitions in a TM implicitly reject. By that, we didn't mean that the TM's transition function can be undefined on certain inputs. Instead, it means "if we don't draw in a transition in a picture representing the TM, it means that the transition does exist and goes to a rejecting state, but we were just too lazy to draw it in." So you should assume that every transition not drawn in a picture of a TM really is there and really goes to some rejecting state.*

*Also, take a moment to appreciate the fact that you can read the notation in this question and understand what it means! Could you imagine doing that at the start of the quarter?*

## 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  $\mathcal{A}(D) = \mathcal{A}(D')$ . 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: What Does it Mean to Solve a Problem?

Let  $L$  be a language over  $\Sigma$  and  $M$  be a TM with input alphabet  $\Sigma$ . 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  $\Sigma$ , then there is a TM  $M$  that satisfies properties (1) and (2).
- ii. Prove that if  $L$  is any language over  $\Sigma$ , then there is a TM  $M$  that satisfies properties (1) and (3).
- iii. Prove that if  $L$  is any language over  $\Sigma$ , then there is a TM  $M$  that satisfies properties (2) and (3).
- iv. Suppose that  $L$  is a language over  $\Sigma$  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.*

## Optional Fun Problem Two: TMs and Regular Languages (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  $\mathcal{A}(M)$  is regular.