Problem Set 8

In this problem set, you'll transition away from the regular languages to the context-free languages, the **R** and **RE** languages, and beyond! 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, June 2nd at the start of class.

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 $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$, 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 $\Sigma = \{\texttt{a}, \texttt{b}\}$, write a CFG for the language $\{ w \in \Sigma^* \mid w$ is a palindrome and $|w| \geq 10 \}$. (Recall that a palindrome is a string that's the same when read forwards and backwards.) For example, we have $\texttt{abaaaaaaba} \in L$ and $\texttt{abbbababbba} \in L$, but $\texttt{abba} \notin L$ and $\texttt{bbbbbbbbba} \notin L$.

iii. Given $\Sigma = \{\texttt{a}, \texttt{b}\}$, 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, $\texttt{aab} \in L$ and $\texttt{baabab} \in L$, but $\texttt{aba} \notin L$ and $\texttt{bb} \notin L$.

iv. Given $\Sigma = \{\texttt{1}, \texttt{+}, \texttt{=}\}$, write a context-free grammar for the language $\{ \texttt{1}^m\texttt{+}\texttt{1}^n\texttt{=}\texttt{1}^{m+n} \mid m, n \in \mathbb{N} \}$. 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`.

v. 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 $\Sigma = \{\texttt{y}, \texttt{d}\}$, 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 $L = \{ w \in \Sigma^* \mid$ w describes a series of steps where you and your dog arrive at the same point $\}$. For example, the strings `yyyddd`, `ydyd`, and `yyyddddddyyy` are all in L. Write a CFG for L.

vi. 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:

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

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

Problem Two: 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 construct an NFA *N* such that $\mathscr{L}(G) = \mathscr{L}(N)$. You don't need to formally prove that your construction is correct, but you should give a good intuitive explanation as to why the grammar has the same language as the generated NFA. Additionally, to illustrate your construction, show the NFA you'd construct from the following right-linear grammar:

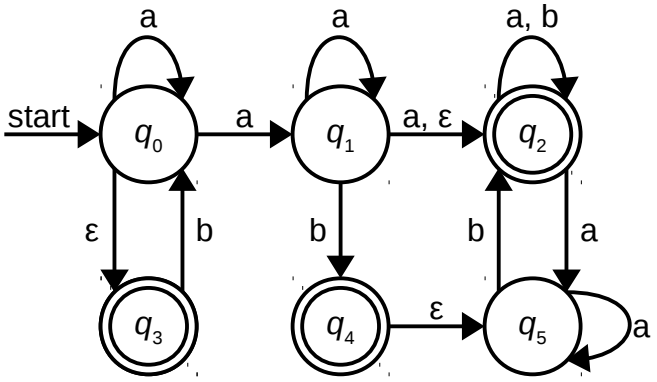
A → aB | bC

B → aB | ε

C → aD | A | bC

D → aD | bD | ε

ii. Let *N* be an NFA. Describe how to construct a right-linear grammar *G* such that $\mathscr{L}(G) = \mathscr{L}(N)$. You don't need to formally prove that your construction is correct, but you should give a good intuitive explanation as to why the generated grammar has the same language as the NFA. Additionally, to illustrate your construction, show the grammar that you'd construct from the following NFA:



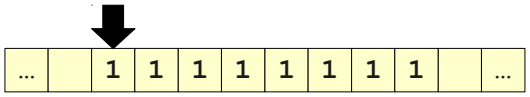
Problem Three: The Collatz Conjecture

In Wednesday's lecture, we discussed the *Collatz conjecture*, which claims that the following procedure (called the *hailstone sequence*) terminates for all positive natural numbers n:

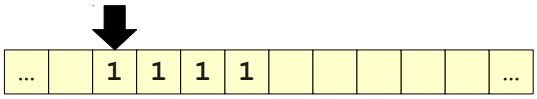
- If $n = 1$, stop.
- If n is even, set $n = n / 2$.
- If n is odd, set $n = 3n + 1$.
- Repeat.

In lecture, we claimed that it was possible to build a TM for the language $L = \{ 1^n \mid n \geq 1$ and the hailstone sequence terminates for $n \}$ over the alphabet $\Sigma = \{1\}$. In this problem, you will do exactly that. The first two parts to this question ask you to design key subroutines for the TM, and the final piece asks you to put everything together to assemble the final machine.

i. Design a TM subroutine that, for every $n \in \mathbb{N}$, given a tape holding 1^{2n} 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, and your TM should end with the tape head reading the first 1 of the result. For example, given this initial configuration:

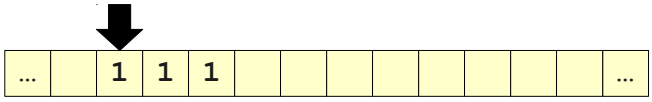


The TM would end with this configuration:

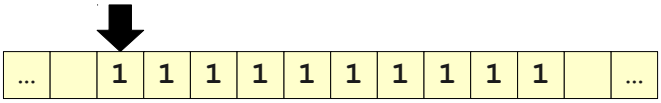


You can assume that there are an even number of 1s on the tape at startup and can have your TM behave however you'd like if this isn't the case. Please use our provided TM editor to design, develop, test, and submit your answer to this question. Since our TM tool doesn't directly support subroutines,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

ii. Design a TM subroutine that, for every $n \in \mathbb{N}$, given a tape holding 1^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:



The TM would end with this configuration:



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. Using your TMs from parts (i) and (ii) as subroutines, 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 $\texttt{1}^{2n}$ into $\texttt{1}^{n}$, have your machine transition into the state named `half`. When that subroutine finishes, the TM will automagically transition 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 $\texttt{1}^{n}$ into $\texttt{1}^{3n+1}$, have your machine transition into the state named `trip`. When that subroutine finishes, the TM will automagically transition into the state labeled `trip_`. You do not need to – and should not – define any transitions into `trip_` or out of `trip`.

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 Four: Jumbled Jargon

Over the past week 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.

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. 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.

Problem Six: R and RE Languages

We have covered a lot of terminology and concepts in the past few days pertaining to Turing machines and **R** and **RE** languages. These problems are designed to explore some of the nuances of how Turing machines, languages, decidability, and recognizability all relate to one another. Please don't hesitate to ask if you're having trouble answering these questions – we hope that by working through them, you'll get a much better understanding of key computability concepts.

i. Give a high-level description of 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). 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.

Problem Seven: Isn't Everything Undecidable?

(We recommend reading the Guide to Self-Reference on the course website before attempting this problem.)

In lecture, we proved that A_{TM} and the halting problem are undecidable – that, in some sense, they're beyond the reach of algorithmic problem-solving. The proofs we used involved the nuanced technique of self-reference, which can seem pretty jarring and weird the first time you run into it. The good news is that with practice, you'll get the hang of the technique pretty quickly!

One of the most common questions we get about self-reference proofs is why you can't just use a self-reference argument to prove that *every* language is undecidable. As is often the case in Theoryland, the best way to answer this question is to try looking at some of the ways you might try to use self-reference to prove that every language is undecidable, then see where those arguments break down.

To begin with, consider this proof:

> ***Theorem:*** All languages are undecidable.
>
> ***Proof:*** Suppose for the sake of contradiction that there is a decidable language *L*. Then we can build the following self-referential program, which we'll call *P*:
>
> ```
> int main() {
>     string me = mySource();
>     string input = getInput();
>
>     if (willAccept(me, input) {
>         reject();
>     } else {
>         accept();
>     }
> }
> ```
>
> Now, given any input *w*, program *P* either accepts *w* or it does not accept *w*. If *P* accepts *w*, then the call to `willAccept(me, input)` will return true, at which point *P* rejects *w*, a contradiction! Otherwise, *P* does not accept *w*, so the call to `willAccept(me, input)` will return false, at which point *P* accepts *w*, a contradiction!
>
> In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, no languages are decidable. ■

This proof has to be wrong because we know of many decidable languages.

i. What's wrong with this proof? Be as specific as possible.

Here's another incorrect proof that all languages are undecidable:

> ***Theorem:*** All languages are undecidable.
>
> ***Proof:*** Suppose for the sake of contradiction that there is a decidable language L. This means there's a decider for L; call it `inL`.
>
> Now, consider the following program, which we'll call P:
>
> ```
> int main() {
>     string input = getInput();
>
>     if (inL(input)) {
>         reject();
>     } else {
>         accept();
>     }
> }
> ```
>
> Now, given any input w, either $w \in L$ or $w \notin L$. If $w \in L$, then the call to `inL(input)` will return true, at which point P rejects w, a contradiction! Otherwise, if $w \notin L$, then the call to `inL(input)` will return false, at which point P accepts w, a contradiction!
>
> In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, no languages are decidable. ■

It's a nice read, but this proof isn't correct.

ii. What's wrong with this proof? Be as specific as possible.

Extra Credit Problem: Quine Relays (1 Point Extra Credit)

In either C, C++, Python, or Java, write four different programs with the following properties:

- Running the first program prints the complete source code of the second program.
- Running the second program prints the complete source code of the third program.
- Running the third program prints the complete source code of the fourth program.
- Running the fourth program prints the complete source code of the first program.
- None of the programs perform any kind of file reading.

In other words, we'd like a collection of four different programs, each of which prints the complete source of the next one in the sequence, wrapping back around at the end.

Please submit your programs by emailing the staff list (cs103-spr1617-staff@lists.stanford.edu) with the subject "PS8 EC" and attaching your source files as a .zip archive. (We ask that you submit this way because we need to be able to independently verify that your programs work as expected.)