Extra Practice Problems 3

Here’s another (giant!) compilation of practice problems you can use to review just about everything from this quarter. Let us know if there are any other topics you’d like more practice with – we can expand this packet of problems with some additional exercises.

Problem One: Cartesian Products, Subsets, Logic, and Proofwriting

Consider the following claim:

Claim: if $A$, $B$, $C$, and $D$ are sets where $A \times B \subseteq C \times D$, then $A \subseteq C$ and $B \subseteq D$.

This question is all about this claim.

i. The claim is false. Disprove it.

Below is a purported proof that the claim is true:

(Incorrect!) Proof: Let $A$, $B$, $C$, and $D$ be any sets where $A \times B \subseteq C \times D$. We will prove that $A \subseteq C$ and that $B \subseteq D$. To do so, consider any $a \in A$ and $b \in B$. We will prove that $a \in C$ and that $b \in D$.

Since $a \in A$ and $b \in B$, we see that $(a, b) \in A \times B$. Because we know that $A \times B \subseteq C \times D$, we see that $(a, b) \in C \times D$. Therefore, we have $a \in C$ and $b \in D$, as required. ■

The error in this proof is fairly subtle and involves a mismatch between what needs to be proved and what was actually proved here.

You might think that the issue here has to do with the fact that the author says “consider any $a \in A$ and $b \in B$” without checking that there actually are elements of $a$ and $b$. Turns out, that’s not a problem! Look back over the proofs we’ve written this quarter where we prove universally-quantified statements about elements of sets. You’ll notice that we’ve never explicitly checked to make sure those sets aren’t empty. The reason for this is that there’s an implicit contract between the proof writer and the proof reader. The writer knows that if they’re trying to prove a universally-quantified statement about a set and the set is empty, then the claim is vacuously true. The reader can then think “that’s fine, I don’t care about that case, so please show me the case where the set isn’t empty.”

Instead, the issue here has to do with a mismatch between first-order logic statements.

ii. Look at the last sentence of the first paragraph of this proof, where the author articulates where they’re going to begin and where they’re going to end. Based on what’s stated there, write out, in first-order logic, what exactly it is that the author is going to prove about $a$ and $b$. Some things to think about: what quantifiers apply to the variables? What assumption is being made? What is going to be proved?

iii. Now, write out, in first-order logic, what statement should be proved about $a$ and $b$. 
iv. Explain why what you found in (ii) is not logically equivalent to what you found in (iii).

**Problem Two: Repeated Squaring**

In many applications in computer science, especially cryptography, it is important to compute exponents efficiently. For example, the RSA public-key encryption system, widely used in secure communication, relies on computing huge powers of large numbers. Fortunately, there is a fast algorithm called repeated squaring for computing $x^y$ in the special case where $y$ is a natural number.

The repeated squaring algorithm is based on the following function $RS$:

$$RS(x, y) = \begin{cases} 
1 & \text{if } y = 0 \\
RS(x, y/2)^2 & \text{if } y \text{ is even and } y > 0 \\
x \cdot RS(x, (y-1)/2)^2 & \text{if } y \text{ is odd and } y > 0
\end{cases}$$

For example, we could compute $2^{10}$ using $RS(2, 10)$ as follows:

In order to compute $RS(2, 10)$, we need to compute $RS(2, 5)^2$.

In order to compute $RS(2, 5)$, we need to compute $2 \cdot RS(2, 2)^2$.

In order to compute $RS(2, 2)$, we need to compute $RS(2, 1)^2$.

By definition, $RS(2, 0) = 1$

so $RS(2, 1) = 2 \cdot RS(2, 0)^2 = 2 \cdot 1^2 = 2$.

so $RS(2, 2) = RS(2, 1)^2 = 2^2 = 4$.

so $RS(2, 5) = 2 \cdot RS(2, 2)^2 = 2 \cdot 4^2 = 32$.

so $RS(2, 10) = RS(2, 5)^2 = 32^2 = 1024$.

The $RS$ function is interesting because it can be computed much faster than simply multiplying $x$ by itself $y$ times. Since $RS$ is defined recursively in terms of $RS$ with the $y$ term roughly cut in half, $RS$ can be evaluated using approximately $\log_2 y$ multiplications. (You don't need to prove this).

Prove that for any $x \in \mathbb{R}$ and any $y \in \mathbb{N}$, that $RS(x, y) = x^y$. *(Hint: use complete induction on $y$).*

**Problem Three: Diagonalization and Logic**

Given the predicates

- $\text{String}(w)$, which states that $w$ is a string over alphabet $\Sigma$;
- $\text{TM}(M)$, which states that $M$ is a TM with input alphabet $\Sigma$; and
- $\text{Accepts}(M, w)$, which states that $M$ accepts $w$,

along with the function $\langle O \rangle$, which represents the encoding of some object $O$, write a statement in first-order logic that says “$L_D \notin \text{RE}$.” (We’ll cover $L_D$ in Week Nine. Looking forward: the language $L_D$ is defined as $L_D = \{ \langle M \rangle \mid \langle M \rangle \notin \mathcal{L}(M) \}$.)

**Problem Four: Addition, Formally**

Let $A$, $B$, $C$, and $D$ be sets where $|A| = |C|$, $|B| = |D|$, $A \cap B = \emptyset$, and $C \cap D = \emptyset$. Using the formal definition of equal cardinality, prove that $|A \cup B| = |C \cup D|$.

Fun fact: this is how we can define what it means to add two cardinalities. For example, $\aleph_0 + \aleph_0$ is formally defined as “the cardinality of $|A \cup B|$, where $A$ and $B$ are any two disjoint sets of cardinality $\aleph_0$.”
Problem Five: Partial Sums
Suppose that you have a set $S$ of $n > 0$ natural numbers. Prove that there must be a nonempty subset of $S$ where the sum of the numbers in that subset is a multiple of $n$. (Hint: Number the elements of $S$ as $x_1, x_2, \ldots, x_n$. Then, look at $x_1, x_1 + x_2, x_1 + x_2 + x_3$, etc.)

Problem Six: Fun with DFAs and NFAs
Here's some true-or-false questions to ponder:

i. True or false: If $D$ is a DFA over alphabet $\Sigma$ and $D$ has no accepting states, then $\mathcal{L}(D) = \emptyset$.

ii. True or false: If $D$ is a DFA over alphabet $\Sigma$ and $D$ has no rejecting states, then $\mathcal{L}(D) = \Sigma^*$.

iii. True or false: If $N$ is an NFA over alphabet $\Sigma$ and $N$ has no accepting states, then $\mathcal{L}(N) = \emptyset$.

iv. True or false: If $N$ is an NFA over alphabet $\Sigma$ and $N$ has no rejecting states, then $\mathcal{L}(N) = \Sigma^*$.

Let $\Sigma = \{a, b, c, d, e\}$ and let $L$ be the following language:

$$L = \{w \in \Sigma^* | \text{every character from } \Sigma \text{ appears at least once in } w\}$$

Any DFA for $L$ must have at least 32 states (you don't need to prove this.)

v. Prove that any DFA for $\overline{L}$ must have at least 32 states.

vi. Design a reasonably-sized NFA for $\overline{L}$. This shows that even if you can't find a small NFA for a language, you might be able to find a small NFA for its complement.

Problem Seven: Antitautonyms
Let $\Sigma = \{a, b\}$ and consider the language $L = \{wx | w \in \Sigma^*, x \in \Sigma^*, |w| = |x|, \text{ and } w \neq x\}$. Prove that $L$ is not a regular language.

Problem Eight: Closure Properties of CFGs
This question explores closure properties of CFLs.

i. Show that the context-free languages are closed under union, concatenation, and Kleene star.

ii. Although we didn't prove this, the context-free languages are not closed under complementation. In lecture, you saw a CFG for the language $\{w \in \{a, b\}^* | w \text{ is a palindrome}\}$, and on Problem Set Eight you built a CFG for the complement of this language. Explain how this is possible even though the context-free languages aren't closed under complementation.

Problem Nine: Designing Turing Machines
Design a TM over the alphabet $\Sigma = \{a, b\}$ whose language is $\{w \in \Sigma^* | w \text{ does not contain } aa \text{ or } bb \text{ as substrings}\}$. 
**Problem Ten: Approximating \( L_D \)**

Prove that there is a language \( X \) where \( X \subseteq L_D \), where \( X \) contains infinitely many strings, and where \( X \) is an RE language.

**Problem Eleven: Narcissistic Turing Machines**

Let \( L = \{ \langle M \rangle \mid M \text{ is a TM and } A(M) = \{ \langle M \rangle \} \} \). In other words, \( L \) is the set of all TMs that accept themselves and only themselves. (We can think of them as narcissistic TMs.)

Prove that \( L \not\in R \).

**Problem Twelve: Powers, Multiples, and Induction**

Let \( k \geq 1 \) be any natural number. Prove, by induction, that \((k+1)^n – 1 \) is a multiple of \( k \) for all \( n \in \mathbb{N} \).

For a bit more of an algebraic exercise, try proving this without using induction. It’s really cute! 😃

**Problem Thirteen: Strongly Connected Graphs**

A directed graph is called **strongly connected** if for any pair of nodes \( u \) and \( v \) in the graph, there's a path from \( u \) to \( v \) and from \( v \) to \( u \). In a directed graph, the **indegree** of a node is the number of edges entering it, and its **outdegree** is the number of edges leaving it. Find a strongly-connected directed graph with 137 nodes where each node's **indegree** is equal to its **outdegree**.

**Problem Fourteen: Closure Properties and Logic**

Given the predicates

- \( TM(M) \), which states that \( M \) is a TM;
- \( String(w) \), which states that \( w \) is a string; and
- \( Accepts(M, w) \), which states that \( M \) accepts \( w \),

Write a statement in first-order logic that says “the RE languages are closed under union.”

**Problem Fifteen: Properties of Functions**

This question explores properties of special classes of functions.

i. Prove or disprove: if \( f : \mathbb{R} \rightarrow \mathbb{R} \) is a bijection, then \( f(r) \geq r \) for all \( r \in \mathbb{R} \).

ii. Prove or disprove: if \( f : \mathbb{N} \rightarrow \mathbb{N} \) is a bijection, then \( f(n) = n \) for all \( n \in \mathbb{N} \).

iii. Prove or disprove: if \( f : \mathbb{R} \rightarrow \mathbb{R} \) and \( g : \mathbb{R} \rightarrow \mathbb{R} \) are bijections, then the function \( h : \mathbb{R} \rightarrow \mathbb{R} \) defined as \( h(x) = f(x) + g(x) \) is also a bijection.

**Problem Sixteen: The Indistinguishability Relation**

Let \( L \) be a language over a set \( \Sigma \). Prove that if \( x \equiv_L y \), then for any string \( z \in \Sigma^* \) we have \( xz \equiv_L yz \). (Here, the relation \( \equiv_L \) is the indistinguishability relation you saw in Problem Set Seven and Problem Set Eight.)
Problem Seventeen: Spin Me An Entree
Suppose that \( n \) people are seated at a round table at a restaurant. Each of the \( n \) people orders a different entrée for dinner. The waiter brings all of the entrées out and places one dish in front of each person. Oddly enough, the waiter doesn't put anyone's dish in front of them.
Prove that there is some way to rotate the table so that at least two people have their entree in front of them.

Problem Eighteen: Regular Languages and Parity
Consider the following language over \( \Sigma = \{ O, E \} \):

\[
PARITY = \{ w \mid w \text{ has even length and has the form } E^n \text{ or } w \text{ has odd length and has the form } O^n \}
\]
For example, \( EE \in PARITY, \ 0000 \in PARITY, \ EEEE \in PARITY, \) and \( \varepsilon \in PARITY, \) but \( EEE \notin PARITY, \ E0 \notin PARITY, \) and \( 0000 \notin PARITY. \)
i. Write a regular expression for \( PARITY. \)
ii. Design a DFA that accepts \( PARITY. \)

Problem Nineteen: Giant Balanced Strings
Let \( \Sigma = \{ a, b \} \) and let \( L = \{ w \in \Sigma^* \mid w \text{ has the same number of } a\text{'s and } b\text{'s and } |w| \geq 10^{100} \}. \)
i. Prove or disprove: \( L \) is not a regular language.
ii. Prove or disprove: there is at least one infinite subset of \( L \) that is regular.

Problem Twenty: CFGs and Swedish Pop Music
Let \( \Sigma = \{ a, b \} \) and let \( L = \{ w \in \Sigma^* \mid w \text{ is a palindrome and } w \text{ contains } abba \text{ as a substring } \}. \) Write a context-free grammar for \( L. \)

Problem Twenty One: Checking for Equality
Let \( \Sigma = \{ a, b, = \}. \) Draw the state-transition diagram of a TM for the \( \{ w=w \mid w \in \{ a, b \}^* \}. \)
Problem Twenty Two: Closure Properties of RE

Earlier in this packet of problems you translated the statement “the RE languages are closed under union” into first-order logic. It turns out that this statement is true, but a bit trickier to prove than you might expect.

If we take a language $L \in \text{RE}$, we know that we can get a recognizer $M$ for it. A recognizer for $L$, in software, would be a function

```cpp
bool inL(string w)
```

that takes as input a string $w$. If $w \in L$, then $\text{inL}(w)$ returns true. If $w \notin L$, then $\text{inL}(w)$ may return false, or it may loop infinitely.

Let $L_1$ and $L_2$ be RE languages. Below is an incorrect construction that purportedly is a recognizer for $L_1 \cup L_2$:

```cpp
bool inL1uL2(string w) {
    return inL1(w) || inL2(w);
}
```

Here, $\text{inL1}$ and $\text{inL2}$ are recognizers for $L_1$ and $L_2$, respectively.

i. Give concrete examples of languages $L_1$ and $L_2$ and implementations of methods $\text{inL1}$ and $\text{inL2}$ such that the above piece of code is not a recognizer for $L_1 \cup L_2$. Justify your answer.

To show that the RE languages are closed under union, it's easiest to think about combining together two verifiers for the input languages to produce a verifier for their union.

ii. Using the verifier definition of RE, prove that the RE languages are closed under union.

Problem Twenty Three: Output Restrictions

Let $\Sigma = \{a, b\}$ and let $L = \{ \langle M \rangle \mid M$ is a TM and $\mathbb{L}(M) \subseteq a^* \}$. Prove that $L \notin \text{R}.$

Problem Twenty Four: Power Sets and Cartesian Products

Prove or disprove: there are sets $A$ and $B$ where $\varphi(A \times B) = \varphi(A) \times \varphi(B)$.

Problem Twenty Five: The Well-Ordering Principle

The well-ordering principle states that if $S \subseteq \mathbb{N}$ and $S \neq \emptyset$, then $S$ contains an element $n_0$ that is less than all other elements of $S$. There is a close connection between the well-ordering principle and the principle of mathematical induction.

Suppose that $P$ is some property such that

- $P(0)$
- $\forall k \in \mathbb{N}. (P(k) \rightarrow P(k+1))$

Using the well-ordering principle, but without using induction, prove that $P(n)$ holds for all $n \in \mathbb{N}$. This shows that if you believe the well-ordering principle is true, then you must also believe the principle of mathematical induction.
Problem Twenty Six: Restricting and Manipulating Logic
Consider the following formula in first-order logic:
\[
\forall x \in \mathbb{R}. \forall y \in \mathbb{R}. (x < y \rightarrow \exists p \in \mathbb{Z}. \exists q \in \mathbb{Z}. (q \neq 0 \land x < p/q \land p/q < y))
\]
This question explores this formula.

i. Translate this formula into plain English. As a hint, there's a very simple way of expressing the concept described above.

ii. Rewrite this formula so that it doesn't use any universal quantifiers.

iii. Rewrite this formula so that it doesn't use any existential quantifiers.

iv. Rewrite this formula so that it doesn't use any implications.

v. Negate this formula and push the negations as deep as possible.

Problem Twenty Seven: Restrictions of Relations
Let \( R \) be a binary relation over a set \( A \). For any set \( B \subseteq A \), we can define the restriction of \( R \) to \( B \), denoted \( R|_B \), to be a binary relation over the set \( B \) defined as follows:
\[
x R|_B y \text{ if } x R y.
\]
In other words, the relation \( R|_B \) behaves the same as \( R \), but only on the elements of \( B \).

i. Prove or disprove: if \( R \) is an equivalence relation over a set \( A \) and \( B \) is an arbitrary subset of \( A \), then \( R|_B \) is an equivalence relation over \( B \).

ii. Prove or disprove: if \( R \) is a strict order over a set \( A \) and \( B \) is an arbitrary subset of \( A \), then \( R|_B \) is a strict order over \( B \).

iii. Prove or disprove: there is a strict order \( R \) over a set \( A \) and a set \( B \subseteq A \) such that \( R|_B \) is an equivalence relation.

iv. Prove or disprove: there is an equivalence relation \( R \) over a set \( A \) and a set \( B \subseteq A \) such that \( R|_B \) is a strict order.
Problem Twenty Eight: Functions and Relations, Together!

(Midterm Exam, Spring 2015)

In this question, let \( A = \{1, 2, 3, 4, 5\} \). Let \( f : A \rightarrow A \) be an arbitrary function from \( A \) to \( A \) that we know is not a surjection. We can then define a new binary relation \(~\) as follows: for any \( a, b \in A \), we say \( a \sim f b \) if \( f(a) = b \). Notice that this relation depends on the particular non-surjective function \( f \) that we pick; if we choose \( f \) differently, we'll get back different relations. This question explores what we can say with certainty about \(~\) knowing only that its domain and codomain are \( A \) and that it is not a surjection.

Below are the six types of relations we explored over the course of this quarter. For each of the types, determine which of the following is true:

- The relation \(~\) is always a relation of the given type, regardless of which non-surjective function \( f : A \rightarrow A \) we pick.
- The relation \(~\) is never a relation of the given type, regardless of which non-surjective function \( f : A \rightarrow A \) we pick.
- The relation \(~\) is sometimes, but not always a relation of the given type, depending on which particular non-surjective function \( f : A \rightarrow A \) we pick.

Since these options are mutually exclusive, check only one box per row. (Hint: Draw a lot of pictures.)

<table>
<thead>
<tr>
<th>(~) is reflexive</th>
<th>□ Always</th>
<th>□ Sometimes, but not always</th>
<th>□ Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>(~) is irreflexive</td>
<td>□ Always</td>
<td>□ Sometimes, but not always</td>
<td>□ Never</td>
</tr>
<tr>
<td>(~) is symmetric</td>
<td>□ Always</td>
<td>□ Sometimes, but not always</td>
<td>□ Never</td>
</tr>
<tr>
<td>(~) is asymmetric</td>
<td>□ Always</td>
<td>□ Sometimes, but not always</td>
<td>□ Never</td>
</tr>
<tr>
<td>(~) is transitive</td>
<td>□ Always</td>
<td>□ Sometimes, but not always</td>
<td>□ Never</td>
</tr>
<tr>
<td>(~) is an equivalence relation</td>
<td>□ Always</td>
<td>□ Sometimes, but not always</td>
<td>□ Never</td>
</tr>
<tr>
<td>(~) is a strict order</td>
<td>□ Always</td>
<td>□ Sometimes, but not always</td>
<td>□ Never</td>
</tr>
</tbody>
</table>
Problem Twenty Nine: Permutation Parity*

Let \( n \) be an odd natural number and consider the set \( S = \{ 1, 2, 3, \ldots, n \} \). A permutation of \( S \) is a bijection \( \sigma : S \rightarrow S \). In other words, \( \sigma \) maps each element of \( S \) to some unique element of \( S \) and does so in a way such that no two elements of \( S \) map to the same element.

Let \( \sigma \) be an arbitrary permutation of \( S \). Prove that there is some \( r \in S \) such that \( r - \sigma(r) \) is even.

Problem Thirty: Reversing Regular Languages

If \( w \) is a string, then \( w^R \) represents the reversal of that string. For example, the reversal of “table” is “elbat.” If \( L \) is a language, then \( L^R \) is the language \( \{ w^R \mid w \in L \} \) consisting of all the reversals of the strings in \( L \).

It turns out that the regular languages are closed under reversal.

i. Give a construction that turns an NFA for a language \( L \) into an NFA for the language \( L^R \). No proof is necessary.

ii. Give a construction that turns a regular expression for a language \( L \) into a regular expression for the language \( L^R \). No proof is necessary.

Problem Thirty One: Centrist Languages

Prove that the language \( \{ w \in \{ a, b \}^* \mid |w| \equiv_3 0 \text{ and the middle third of the characters in } w \text{ contains at least one } a \} \) is not regular.

Problem Thirty Two: Parenthesis Parity

Let \( \Sigma = \{ (, ) \} \) and let \( L = \{ w \in \Sigma^* \mid w \text{ is a string of balanced parentheses and } w \text{ has an even number of open parentheses} \} \). Write a CFG for \( L \).

Problem Thirty Three: A Better Sorting Algorithm

In lecture, we designed a Turing machine that, given a string of 0s and 1s, puts them into ascending order and then halts. The sorting algorithm we used worked by finding a copy of the substring 10, reversing it, and repeating until no more copies of this substring exists. While this algorithm works, it's not very efficient.

Design a TM that sorts a string of 0s and 1s and does so more efficiently than the machine from class. By “more efficiently,” we mean that the TM you design should, on average, take many fewer steps to complete than our TM.

Problem Thirty Four: Computable Functions
A function \( f : \Sigma^* \to \Sigma^* \) is called a **computable function** if it is possible to write a function \( \text{compute}_f \) that takes in a string \( w \) and outputs \( f(w) \).

Given any computable function \( f \) and language \( L \), let’s define \( f[L] = \{ w \in \Sigma^* | \exists x \in L. f(x) = w \} \). In other words, \( f[L] \) is the set of strings formed by applying \( f \) to each string in \( L \).

Prove that if \( L \in \mathsf{RE} \) and \( f \) is a computable function, then \( f[L] \in \mathsf{RE} \).

Problem Thirty Five: Complementary TMs
Prove that \( L = \{ \langle M, N \rangle | M \text{ is a TM, } N \text{ is a TM, and } \mathcal{L}(M) = \overline{\mathcal{L}(N)} \} \) is not in \( \mathsf{R} \).

Problem Thirty Six: Nonregular Languages via a Different Path
The Myhill-Nerode theorem is a powerful tool for proving that languages aren’t regular, but it might not be the easiest way to prove that a given language isn’t regular. This problem explores a different route you can take to prove that various languages aren’t regular.

i. Prove that if \( L_1 \) is a language, \( L_2 \) is a regular language, and \( L_1 \cap L_2 \) is not regular, then \( L_1 \) is not regular.

ii. Using your result from part (i), but without using the Myhill-Nerode theorem, prove that the language \( L = \{ w \in \{a, b\}^* | w \text{ has the same number of } a \text{’s as } b \text{’s} \} \) is not regular.

Problem Thirty Seven: Just a Few More Grammars
Below is a list of alphabets and languages over those alphabets. Design a CFG for each language.

i. Let \( \Sigma = \{1, \geq \} \) and let \( L = \{ 1^n \geq 1^m | m, n \in \mathbb{N} \text{ and } m \geq n \} \). Write a CFG for \( L \).

ii. On Problem Set 8, you explored the language \( L_1 = \{ 1^m+1^n=1^m \cdot n | m, n \in \mathbb{N} \} \) over the alphabet \( \{1, +, =\} \) Consider the following generalization of this language, which we will call \( L_2 \), which consists of all strings describing unary encodings of two sums that equal one another. For example:

\[
1 + 3 = 4 \text{ would be encoded as } 1+111=1111
\]
\[
4 = 1 + 3 \text{ would be encoded as } 1111=1+111
\]
\[
2 + 2 = 1 + 3 \text{ would be encoded as } 11+11=1+111
\]
\[
2+0+2+0=0+4+0 \text{ would be encoded as } 11++11+=+1111+
\]
\[
0=0 \text{ would be encoded as } =
\]

Notice that there can be any number of summands on each side of the \( = \), but there should be exactly one \( = \) in the string; thus \( 1=1=1 \not\in L_2 \). Write a CFG for \( L_2 \).

iii. Let \( \Sigma = \{ (, ), [ , ] \} \) and let \( L = \{ w \in \Sigma^* | w \text{ is a string of balanced parentheses and brackets} \} \). This means that all parentheses and brackets must match one another, and collectively they must obey the appropriate nesting rules. For example, \( ([ ])[ ] \in L \), but \( [ ] \) \not\in L \). Write a CFG for \( L \).
Problem Thirty Eight: Formalizing the Lava Diagram

In the Guide to the Lava Diagram, we explored these two languages:

\[ L_1 = \{ \langle M \rangle \mid M \text{ is a TM and } |\mathcal{L}(M)| \geq 2 \} \]
\[ L_2 = \{ \langle M \rangle \mid M \text{ is a TM and } |\mathcal{L}(M)| = 2 \} \]

The Guide makes many claims about these languages, but never actually proves them.

i. Prove that \( L_1 \) is undecidable.

ii. Prove that \( L_2 \) is undecidable.

iii. Show that \( L_1 \) is recognizable by designing a verifier for it. Your verifier should be represented in pseudocode via a method with a signature like this one:

\[
\text{bool imConvincedIsInL1(TM } M, \text{ Arg1Type } \text{ arg}_1, \text{ Arg2Type } \text{ arg}_2, \ldots, \text{ ArgnType } \text{ arg}_n)\]

where the arguments beyond the first represent the certificate and can be of any type you’d like.

Problem Thirty Nine: Self-Reference and RE

Since \( A_{\text{TM}} \) is an RE language, there’s a recognizer for \( A_{\text{TM}} \), which we can represent in software as a method \( \text{willAccept} \). Consider the following program \( P \):

```c
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Prove that this program loops on all inputs.
Problem Forty: Threat Detection

There have been a ton of news articles about computer systems being attacked by independent actors and nation-states. You might wonder why our computers are so vulnerable – couldn't someone just write a program that analyzes a program's source code and determine whether it has any security problems?

Let's consider a simplified scenario. Imagine that there's a special method

```java
void sendSecretDataTo(String emailAddress)
```

that, if called, sends an email containing a bunch of secret information to the specified email address. For example, you might call `sendSecretDataTo("john.roberts@supremecourt.gov")` to send all the secret data to Chief Justice John Roberts, or call `sendSecretDataTo("bad.actor@hackers.com")` to send all the secret data to evil hackers.

You are interested in whether it's possible to write a method

```java
bool canLeakDataTo(String program, String emailAddress)
```

that takes as input the source code of a program and an email address, then returns true if there is some execution of `program` that causes the secret data to the specified email address and returns false otherwise. This program would let you check whether a particular program might ever leak data to a specified email address, which would make it easier to check whether the program is secure.

Is it possible to implement this method? If so, write code for the method, then prove that your code works as intended. If not, prove that it's not possible to implement this method.

(As a note, if you try implementing this method, you should do so in a way that doesn’t call `sendSecretDataTo`, and if you try proving this method can’t be written, you can assume that no correct implementation will ever call `sendSecretDataTo`.)

Problem Forty One: Rock, Paper, Scissors

(Final Exam, Winter 2012)

The number of characters in a regular expression is defined to be the total number of symbols used to write out the regular expression. For example, `(a ∪ b)*` is a six-character regular expression, and `ab` is a two-character regular expression.

Let Σ = {a, b}. Find examples of all of the following:

- A regular language over Σ with a one-state NFA but no one-state DFA.
- A regular language over Σ with a one-state DFA but no one-character regular expression.
- A regular language over Σ with a one-character regular expression but no one-state NFA.

Prove that all of your examples have the required properties.