

Problem Set 9

What problems are beyond our capacity to solve? Why are they so hard? And why is anything that we've discussed this quarter at all practically relevant? In this problem set – the last one of the quarter! – you'll explore the absolute limits of computing power.

Before attempting any of the problems on this problem set, we strongly recommend reading over the *Guide to Self-Reference* and *Guide to the Lava Diagram* that are available on the course website, which provide a ton of extra background that you might find useful here.

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

Good luck, and have fun!

Due Friday, March 15th at 2:30PM

Because this problem set is due on the last day of class, no late days may be used and no late submissions will be accepted. Sorry about that! On the plus side, we'll release solutions as soon as the problem set comes due, so you can study them for Monday's final exam.

Problem One: 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 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 . 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();

    /* Do the opposite of what's expected. */
    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. ■

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.

Go one sentence at a time and check that each claim is correct. Something is fishy here.

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 that there is some decider D for the language L , which we can represent in software as a method `willAccept`. Then we can build the following self-referential program, which we'll call P :

```

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

    /* See whether we'll accept, then do the oppo-
site. */
    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, we know that 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. ■

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

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

This one is subtle. Pull up the proof that A_{TM} is undecidable and compare this proof and that one side-by-side, going one sentence at a time if you need to.

Many of the examples we've seen of undecidable languages involve checking for properties of Turing machines or computer programs, which might give you the sense that *every* question you might want to ask about TMs or programs is undecidable. That isn't the case, though, and this question explores why.

Consider the following language L :

$$L = \{ \langle P \rangle \mid P \text{ is a syntactically valid C++ program} \}$$

Below is a purported proof that L is undecidable:

Theorem: The language L is undecidable.

Proof: Suppose for the sake of contradiction that L is decidable. That means that there's some decider D for L , which we can represent in software as a function `isSyntacticallyValid` that takes as input a program and then returns whether that program has correct syntax. Given this function, consider the following program P :

```

int main() {
    string me = mySource();

    /* Execute a line based on whether our syntax
is right. */
    if (isSyntacticallyValid(me)) {
        oops, this line of code isn't valid C++!
    } else {
        int num = 137; // Perfectly valid syntax!
    }
}

```

Now, either this program P is syntactically valid or it is not. If P has valid syntax, then when P is run on any input, it will get its own source code, determine that it is syntactically valid, then execute a syntactically invalid line of code – a contradiction! Otherwise, if P is not syntactically valid, then when P is run on any input, it will get its own source code, determine that it is not syntactically valid, at which point it executes a perfectly valid line of C++ code – a contradiction!

In either case we reach a contradiction, so our assumption must have been incorrect. Therefore, L is undecidable. ■

This proof, unfortunately, is incorrect.

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

Problem Two: Password Checking

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

When you log onto a website with a password, you have the presumption that your password is the only possible password that will log you in. There shouldn't be a “master key” password that can unlock any account, since that would be a huge security vulnerability. But how could you tell? If you had the source code to the password checking system, could you figure out whether your password was the only password that would grant you access to the system?

Let's frame this question in terms of Turing machines. If we wanted to build a TM password checker, “entering your password” would correspond to starting up the TM on some string, and “gaining access” would mean that the TM accepts your string. Let's suppose your password is the string `iheartquokkas`. We'll say that a *password checker* is a TM M where

$$\mathcal{A}(M) = \{\text{iheartquokkas}\};$$

that is, the TM accepts your password `iheartquokkas`, and it doesn't accept anything else. Given a TM, is there some way you could tell whether the TM was a password checker?

Consider the following language L :

$$L = \{ \langle M \rangle \mid M \text{ is a TM and } M \text{ is a password checker} \}.$$

Your task in this problem is to prove that L is undecidable (that is, $L \notin \mathbf{R}$). This means that there's no algorithm that can mechanically check whether a TM is suitable as a password checker. Rather than dropping you headfirst into this problem, we've split this problem apart into a few smaller pieces.

Let's suppose for the sake of contradiction that $L \in \mathbf{R}$. That means that there is some function

bool `isPasswordChecker(string program)`

with the following properties:

- If `program` is the source of a program that accepts just the string `iheartquokkas`, then calling `isPasswordChecker(program)` will return `true`.
- If `program` is not the source of a program that accepts just the string `iheartquokkas`, then calling `isPasswordChecker(program)` will return `false`.

We can try to build a self-referential program that uses the `isPasswordChecker` function to obtain a contradiction. Here's a first try:

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

    if (isPasswordChecker(me)) {
        reject();
    } else {
        accept();
    }
}
```

This code is, essentially, a (minimally) modified version of the self-referential program we used to get a contradiction for the language A_{TM} .

- Prove that the above program P is not a password checker.

What is the definition of a password checker? Based on that, what do you need to prove to show that P is not a password checker?

(Continued on the next page.)

- ii. Suppose that this program is *not* a password checker. Briefly explain why no contradiction arises in this case – no formal justification is necessary.

A good question to think about in the course of answering part (ii) of this problem: this program is very close to the one from the proof that A_{TM} is not decidable. Why do you get a contradiction in the original proof that A_{TM} is undecidable? Why doesn't that same contradiction work here?

Ultimately, the goal of building a self-referential program here is to have the program cause a contradiction regardless of whether or not it's a password checker. As you've seen in part (ii), this particular program does not cause a contradiction if it isn't a password checker. Consequently, if we want to prove that $L \notin \mathbf{R}$, we need to modify it so that it leads to a contradiction in the case where it is not a password checker.

- iii. Modify the above code so that it causes a contradiction regardless of whether it's a password checker. Then, briefly explain why your modified program is correct. No formal proof is necessary.

Follow the advice from the Guide to Self-Reference. Write out a specification of what your self-referential program is trying to do. Based on that, craft code for each of the two cases.

Problem Three: L_D , Cantor's Theorem, and Diagonalization

Here's another perspective of the proof that $L_D \notin \mathbf{RE}$. Suppose we let TM be the set of all encodings of Turing machines. That is,

$$TM = \{ \langle M \rangle \mid M \text{ is a TM} \}$$

We can then define a function $\hat{L} : TM \rightarrow \wp(TM)$ as follows:

$$\hat{L}(\langle M \rangle) = \mathcal{L}(M) \cap TM$$

This question explores some properties of this function.

- i. Briefly describe, in plain English, what $\hat{L}(\langle M \rangle)$ represents.

You shouldn't need more than a sentence.

- ii. Trace through the proof of Cantor's theorem from the Guide to Cantor's Theorem, assuming that the choice of the function f in the proof is the function \hat{L} . What is the set D that is produced in the course of the proof?

Problem Four: Double Verification

This problem explores the following beautiful and fundamental theorem about the relationship between the **R** and **RE** languages:

If L is a language, then $L \in \mathbf{R}$ if and only if $L \in \mathbf{RE}$ and $\bar{L} \in \mathbf{RE}$

This theorem has a beautiful intuition: it says that a language L is decidable ($L \in \mathbf{R}$) precisely if for every string in the language, it's possible to prove it's in the language ($L \in \mathbf{RE}$) and, simultaneously, for every string not in the language, it's possible to prove that the string is not in the language ($\bar{L} \in \mathbf{RE}$). In this problem, we're going to ask you to prove one of the two directions of this theorem.

Let L be a language where $L \in \mathbf{RE}$ and $\bar{L} \in \mathbf{RE}$. This means that there's a verifier V_{yes} for L and a verifier V_{no} for \bar{L} . In software, you could imagine that V_{yes} and V_{no} correspond to methods with these signatures:

```
bool checkIsInL(string w, string c)
```

```
bool checkIsNotInL(string w, string c)
```

Prove that $L \in \mathbf{R}$ by writing pseudocode for a function

```
bool isInL(string w)
```

that accepts as input a string w , then returns true if $w \in L$ and returns false if $w \notin L$. Then, write a brief proof explaining why your pseudocode meets these requirements. You don't need to write much code here. If you find yourself writing ten or more lines of pseudocode, you're probably missing something.

The theorem you proved in this problem is extremely useful for building an intuition for what languages are decidable. You'll see this in the next problem.

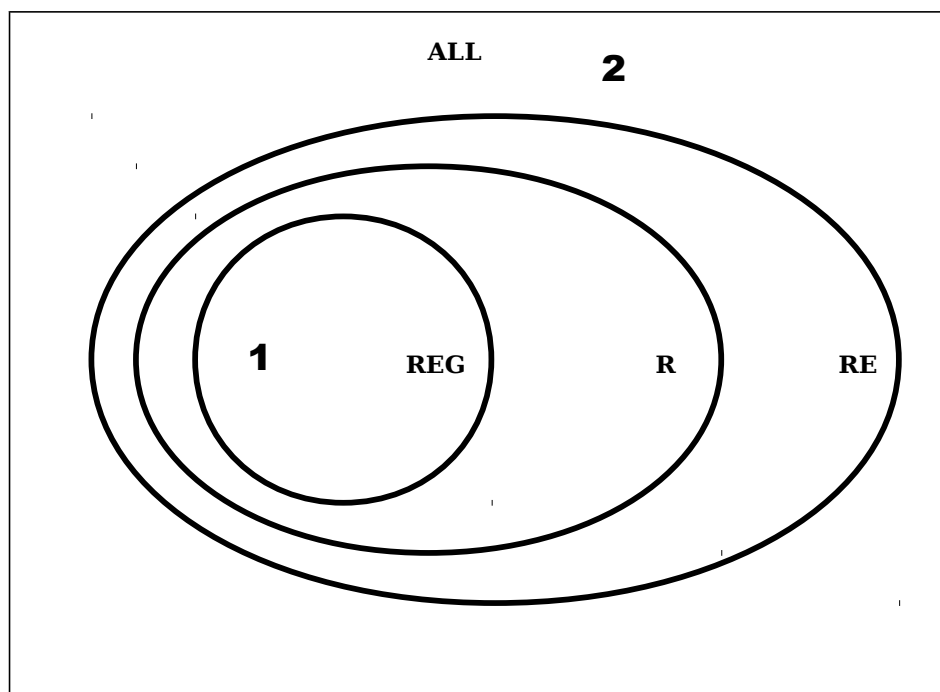
What other constructions have we done on verifiers? How did they work?

Problem Five: The Lava Diagram

Below is a Venn diagram showing the overlap of different classes of languages we've studied so far. We have also provided you a list of twelve numbered languages. For each of those languages, draw where in the Venn diagram that language belongs. As an example, we've indicated where Language 1 and Language 2 should go. No proofs or justifications are necessary – the purpose of this problem is to help you build a better intuition for what makes a language regular, **R**, **RE**, or none of these.

We *strongly* recommend reading over the Guide to the Lava Diagram before starting this problem.

To submit your answers, edit the file `LavaDiagram.h` in the `src/` directory of the starter files for this problem set.



1. Σ^*
2. L_D
3. $\{ a^n \mid n \in \mathbb{N} \}$
4. $\{ a^n \mid n \in \mathbb{N} \text{ and } n \text{ is a multiple of } 137 \}$
5. $\{ 1^n + 1^{m \cdot 2} 1^{n+m} \mid m, n \in \mathbb{N} \}$
6. $\{ \langle M \rangle \mid M \text{ is a Turing machine and } \mathcal{L}(M) \neq \emptyset \}$
7. $\{ \langle M \rangle \mid M \text{ is a Turing machine and } \mathcal{L}(M) = \emptyset \}$
8. $\{ \langle M \rangle \mid M \text{ is a Turing machine and } \mathcal{L}(M) = L_D \}$
9. $\{ \langle M, n \rangle \mid M \text{ is a TM, } n \in \mathbb{N}, \text{ and } M \text{ *accepts* all strings in its input alphabet of length at most } n \}$
10. $\{ \langle M, n \rangle \mid M \text{ is a TM, } n \in \mathbb{N}, \text{ and } M \text{ *rejects* all strings in its input alphabet of length at most } n \}$
11. $\{ \langle M, n \rangle \mid M \text{ is a TM, } n \in \mathbb{N}, \text{ and } M \text{ *loops* on all strings in its input alphabet of length at most } n \}$
12. $\{ \langle M_1, M_2, M_3, w \rangle \mid M_1, M_2, \text{ and } M_3 \text{ are TMs, } w \text{ is a string, and at least two of } M_1, M_2, \text{ and } M_3 \text{ accept } w. \}$

Problem Six: co-RE, Disjoint Unions, and Terrifyingly Difficult Problems

When we first saw that A_{TM} is undecidable, we could at least take consolation in the fact that A_{TM} is recognizable. Then we discovered that L_D is unrecognizable, and up to now we haven't had a comforting fact to fall back on. But there is something about L_D we can console ourselves with: the complement of L_D , the language \bar{L}_D , is an **RE** language.* In other words, while there's no general way to prove that some TM *will not* accept its own encoding, there *is* a general way to prove that some TM *will* accept its own encoding.

The fact that L_D isn't an **RE** language while \bar{L}_D is still in **RE** suggests that there might be a bit more to the computability landscape than just **R** and **RE**. And indeed there is: the same way that A_{TM} is the poster child of an **RE** language, the language L_D is the poster child of a **co-RE language**. Formally speaking, the class **co-RE** consists of all the languages whose *complement* is an **RE** language:

$$\text{co-RE} = \{ L \mid \bar{L} \in \text{RE} \}.$$

Intuitively speaking, the **co-RE** languages are languages where if you have a string that *isn't* in the language, there's some easy way to prove that it isn't in the language.

- i. Prove that $A_{TM} \notin \text{co-RE}$.

You haven't seen the term "co-RE" before, but you have seen something like this somewhere.

At this point, we have an **RE** language that's not in **co-RE** (A_{TM}) and a **co-RE** languages that's not in **RE** (L_D). As a coda to our treatment of unsolvable problems, let's see a language that's neither in **RE** nor **co-RE**. In other words, there's no general way to prove that strings in that language are indeed in that language, nor is there way to prove that strings not in that language are indeed not in that language. Yikes!

In what follows, let's assume all languages are over the alphabet $\Sigma = \{0, 1\}$.

Given two languages A and B over Σ , the **disjoint union** of A and B , denoted $A \uplus B$, is the language

$$A \uplus B = 0A \cup 1B.$$

For example, if $A = \{1, 10, 100, 1000\}$ and $B = \{\epsilon, 0, 1, 00, 01, 10, 11\}$, then $A \uplus B$ is the language

$$A \uplus B = \{01, 010, 0100, 01000, 1, 10, 11, 100, 101, 110, 111\}$$

Notice how each string in $A \uplus B$ is tagged with which language it originated in. Any string that starts with 0 came from A , and any string that starts with 1 came from B .

- ii. Let A and B be languages where $A \uplus B \in \text{RE}$. Prove that $A \in \text{RE}$. Structure your proof like what you did in Problem Four: assume you have code for a verifier or recognizer for $A \uplus B$, use it to write code for a verifier or recognizer for A , then explain why that code works as intended.
- iii. Let A and B be languages where $A \uplus B \in \text{co-RE}$. Prove that $A \in \text{co-RE}$.

If you have a language in co-RE, what can you say about that language? If you want to prove that a language is in co-RE, what do you need to prove about that language?

The results you proved in parts (ii) and (iii) of this problem can be extended to show that if $A \uplus B \in \text{RE}$, then $B \in \text{RE}$, and if $A \uplus B \in \text{co-RE}$, then $B \in \text{co-RE}$, though you don't need to prove this.

- iv. Let L be an undecidable language. Prove $L \uplus \bar{L} \notin \text{RE}$ and $L \uplus \bar{L} \notin \text{co-RE}$.

* While we're not going to ask you to formally prove that \bar{L}_D is an **RE** language, we would *definitely* recommend taking a few minutes to ponder how you'd build a recognizer or verifier for it.

Your result from part (iv) shows there are problems harder than A_{TM} and L_D ; one example is $L_D \cup \bar{L}_D$. But even this nightmarish problem sits within a class of problems that can still be touched by computing power, if only in a very weak sense. Specifically, $L_D \cup \bar{L}_D$ is what's called a Δ_2^0 language, where Δ_2^0 is a class of languages that contains all of **RE** and **co-RE**, plus a bunch of other problems. And Δ_2^0 itself sits inside even larger classes of languages, stretching outward to infinity. Want to learn more? Take Phil 152!

Congratulations on finishing the last problem set of the quarter!

We're extremely impressed with how much progress you've made since the start of the quarter.
Best of luck on the final exam!

Grand Challenge Problem: $P \stackrel{?}{=} NP$ (Worth an A+, \$1,000,000, and a Ph.D)

Prove or disprove: $P = NP$.

Take fifteen minutes and try this. Seriously. And if you can't crack this problem, feel free to submit your best effort, or the silliest answer you can think of.