CHAPTER 14

Computability

14.1 WHY YOU SHOULD CARE

In this chapter we will study what Turing machines are capable of from a higher level and see why we believe this model of computation to be the one that captures everything we believe a computer capable of doing. We will then show that there is a problem that is uncomputable. Since we established Turing machines as our model of computation, we will see that no mechanical computer that currently exists or ever will exist can solve this problem. We will then show that a number of other problems are also uncomputable by bootstrapping from this one problem using a technique called a reduction, another important concept in computer science.

14.2 VARIATIONS OF TURING MACHINES

While the standard model of the Turing machine from the last chapter is as powerful as we will need, we will see in this section that it is equivalent in power to seemingly more advanced models of computation, such as Turing machines that have multiple tapes.

We will first see how to construct a Turing machine with multiple tracks using a standard Turing machine. A two-track Turing machine has each cell on the tape split into a top and bottom part so that each cell can hold two symbols. (See Figure 14.1.) This model of the Turing machine can then make transitions based on the state and the symbols on both of the tracks.

We can simulate any such two-track Turing machine by treating the tape alphabet as pairs of symbols from the original tape alphabet. For example, if a cell had a 1 and a B in it, we would represent this with a single symbol $\frac{1}{B}$ and make a standard Turing machine that did whatever the two-track Turing machine did by designing the

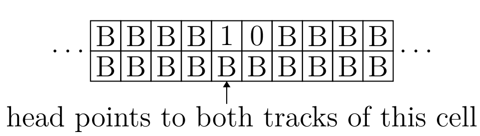

Figure 14.1: Multi-track TM

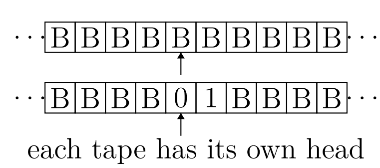

Figure 14.2: Multi-tape TM

appropriate rules. Similarly, we can see that a multi-track Turing machine, for some number of tracks k, would be no more powerful than a standard Turing machine because we can replace the tape alphabet with k-tuples that represent any combination of k symbols from the original tape alphabet.

We can also design multi-tape Turing machines in which there are multiple independent tapes, for which each tape has a separate head. (See Figure 14.2.) Now, each head can be at a different cell of the tape. A k-tape Turing machine can be simulated by a $2k$-track Turing machine by using odd numbered tracks to represent the tapes and the even numbered tracks to maintain the head position of the track before it. We omit the details of how to design such a Turing machine as they would be excessively complex. (You can take a Theory of Computation course to learn more.) Since we know that we can simulate a $2k$-track Turing machine with a standard one, we can now simulate a Turing machine with any (finite) number of independent tapes with a standard Turing machine from the last chapter.

The point of demonstrating these variations is to show you that there are more complex models possible that are no more powerful than the Turing machine. These variations are also getting closer to models of computation you are used to in which you can keep track of multiple values at the same time.

14.3 THE CHURCH-TURING THESIS

Now that we have seen that Turing machines are capable of maintaining multiple tapes (variables in memory) and can perform many of the operations we deem necessary for computation (arithmetic, loops, conditionals) in examples and exercises in the last chapter, we are ready to make an assertion about the power of Turing machines. The following is one version of what is called the Church-Turing thesis, named after two pioneers in the theory of computation, Alonzo Church and Alan Turing.

Definition 14.1

The **Church-Turing thesis** is that all mechanical computation can be performed by a Turing machine.

The above statement is called a thesis because it is a hypothesis. Since the term "mechanical computation" is not well defined, the above thesis cannot be mathematically proved. It hypothesizes that any computation that we think is feasible can be done by a Turing machine. All modern computers are no more or less powerful than a Turing machine in that they can solve the same problems. Given enough time and memory, a Turing machine can perform any computation that we can do on a computer. Conversely, modern computers can perform any computation that a Turing machine is capable of given enough time and computer memory.

Now, this is not to say that we will never be able to build a computer that is more powerful than a Turing machine. It just means that we don't currently believe some more powerful computer is possible and all empirical evidence seems to support that belief.

14.4 UNIVERSAL TURING MACHINES

An important property of the Turing machine is that it is powerful enough to simulate itself. That is, given the encoding of a Turing machine and an input string, we can design a Turing machine (called a **Universal Turing Machine**) that can simulate the given Turing machine on the given input.

You might be wondering how you can encode a Turing machine as a string. Since the description of a Turing machine is always finite, we can encode the description into a single string. For example, we can encode the name of the states, the input and tape alphabet, the starting and accepting states, and the blank symbol into a string. We can then encode the δ transition rules as sequences—for example, $\delta(q_0, 0) = (q_1, B, R)$ could be encoded by a string such as $q0\#0\#q1\#B\#R$, where # is a special separator symbol. Thus, any Turing machine can be represented with a string encoding.

Example 14.1

The Turing machine for adding two unary numbers in Example 13.14 can be given by the following string:

```
3#q0#q1#q2#2#0#1#3#0#1#B#q0#B#1#q2#q0#0#q0#1#R#q0#1#q1#1#R#
q1#1#q1#1#R#q1#B#q2#B#R
```

Notice that all parts of the encoding use the # symbol as a separator. The initial 3 indicates that there are 3 states, which are then named right after. Next, the 2 indicates the size of Σ and these symbols then follow. The 3 indicates the size of Γ and these symbols follow. Next comes the start state q_0 followed by the blank symbol B. The size of F and its contents (just q_2) are given next. Finally, each of the δ transitions is given in the format explained above. For example, the first one `q0#0#q0#1#R` represents $\delta(q_0, 0) = (q_0, 1, R)$.

Given such an encoding of a Turing machine, it is possible (with a lot of effort!) to design a Turing machine that could take such an encoding and use it to process the behavior of the Turing machine on some input string. We could do this by, for example, storing the input string on one tape, the current state on another, the head position on a third, and the Turing machine encoding on a fourth. For each step of the Turing machine simulation, we would look up what character is at the current head position and what state we are currently at and then search the Turing machine encoding tape for the correct move to execute next.

The idea of the Universal Turing Machine is important for a number of reasons. For one, it tells us that Turing machines are capable of running other Turing machines. In fact, this is exactly what happens when you run a program using a compiler or interpreter—these are examples of Universal Turing Machine programs in that they can process arbitrary (correct) code.

Another important reason for Turing machines being able to simulate other Turing machines is that we can have a Turing machine call another Turing machine in a subroutine, allowing us to chain together increasingly complex computations.

14.5 RECURSIVE AND RECURSIVELY ENUMERABLE LANGUAGES

We are now ready to define what languages are computable.

Definition 14.2

A language is called **recursive** (or **computable** or **decidable**) if there exists a Turing machine that accepts the language and halts on all inputs.

Note that we specify that the Turing machine for the language must halt on all inputs. This is because Turing machines are allowed to reject a string by running forever. Since computations that run forever are not very useful, we specify that the Turing machine must halt and accept or halt and reject on every possible input.

There is also a broader collection of languages that defines everything a Turing machine is capable of computing called the recursively enumerable languages.

Definition 14.3

A language is called **recursively enumerable** if there exists a Turing machine that accepts the language.

Notice that this collection of languages contains the recursive ones because recursive languages all have Turing machines (that also must halt on all inputs). This classification isn't as useful from a practical standpoint because it allows Turing machines to run forever on some inputs, but it does help us with some proofs later in this chapter. Next, we prove a few results based on these definitions.

Theorem 14.1

Any language L is recursive if and only if $\overline{L}$ is recursive.

Proof

Let L be a recursive language. Then by the definition of a recursive language, L must have a Turing machine M that accepts it and halts on all inputs. We will design a new Turing machine M' that accepts $\overline{L}$ and halts on all inputs, thereby showing that $\overline{L}$ is recursive as well.

Since M halts on all inputs, it must reject by reaching a state / symbol combination undefined for it. We thus define $M's$ δ function to be identical to that for M except for any missing transitions it should go to a new accepting state. Thus, all strings rejected by M would be accepted by M'. Secondly, the accepting state(s) for M are all made non-accepting for M', so that any computation that reaches them now halts and rejects any strings that were accepted by M. This newly constructed Turing machine M' now halts on all inputs and accepts precisely the language $\overline{L}$. It follows that $\overline{L}$ is recursive.

The other direction follows from replacing L with $\overline{L}$ in the above proof. Thus, L is recursive if and only if $\overline{L}$ is recursive.

Theorem 14.2

The language L is recursive if and only if L and $\overline{L}$ are recursively enumerable.

Proof

Let L be any recursive language. Then, by definition, L has a Turing machine for it that halts on all inputs. Since L has a Turing machine for it, L is also recursively enumerable. By Theorem 14.1 we know that $\overline{L}$ is also recursive and thus also recursively enumerable. We have thus shown that if L is recursive, then L and $\overline{L}$ are both recursively enumerable.

Now we will assume that L and $\overline{L}$ are both recursively enumerable. Then, by definition, L has a Turing machine that accepts it, call it M, and $\overline{L}$ has a Turing machine that accepts it, call it $\overline{M}$. We can design a Turing machine M' for L that halts on all inputs as follows. On any input, M' will simulate M on the input and in parallel simulate $\overline{M}$ on another copy of the input (perhaps on a second tape), alternating steps for these two simulations. We know that we can have a Turing machine simulate another Turing machine based on the existence of the Universal Turing Machine. If the simulation of M ever halts and accepts, then M' will halt and accept. If the simulation of $\overline{M}$ ever halts and accepts, then M' will reject. Notice that M' accepts exactly the language L since M is a Turing machine for L and $\overline{M}$ is a Turing machine for $\overline{L}$. Moreover, exactly one of M and $\overline{M}$ must halt and accept because the input is either in L (in which case M will accept) or it is not (in which case $\overline{M}$ will accept). Since M' is a Turing machine that accepts L and halts on all inputs, this means that L must be recursive.

We have thus proved that L is recursive if and only if L and $\overline{L}$ are recursively enumerable.

Notice in the above proof that we had to run the simulations for M and $\overline{M}$ in parallel because it could be possible for either M or $\overline{M}$ to run forever on rejection and so if we had run them sequentially the computation may never end.

14.6 A NON-COMPUTABLE PROBLEM

To show that there are non-recursive problems, we first need to understand that all strings and Turing machines can be enumerated (listed in a numbered list). To list off all the strings in a fixed order, we can use shortlex ordering. For a review of shortlex ordering of strings, take a look at Section 1.10. To list all Turing machines, we can write an encoding for each one (see Section 14.4) and then also use a shortlex ordering on these. Let us call the Turing machines generated in this manner M_i for each string i.

We are now ready to define the Diagonal Problem (D). The Diagonal Problem can be intuitively thought of as the set of strings that when interpreted as a Turing machine accepts itself as an input.

Definition 14.4

$D = \{i : M_i \text{ accepts input } i\}$

To understand the above definition, we have to interpret the string i in two distinct ways. In one interpretation, we treat the string i as the encoding for a Turing machine M_i as we saw in Section 14.4. If the string i does not correspond to the encoding of any Turing machine, then we simply define M_i to be a simple Turing machine that rejects all inputs by immediately halting and rejecting. The second interpretation of i is simply that of a string—an input string to be accepted or rejected.

Next, we define the complement of the language D, called $\overline{D}$.

Definition 14.5

$\overline{D} = \{i : M_i \text{ rejects input } i\}$

We will now show that $\overline{D}$ does not have a Turing machine that accepts it. Before formally proving this, let us build some intuition about why this is the case. We use a technique called **diagonalization**.

Imagine that we could construct an infinitely large table that has all the Turing machines on the rows and the strings on the columns, as illustrated in Figure 14.3. For simplicity, we will assume that the strings are over the alphabet $\{0, 1\}$. Each entry in the table will indicate whether the Turing machine on the row accepts (A) the string on the column or rejects (R) it. (The entries in Figure 14.3 are not real and just shown for illustrative purposes.) We will focus on the diagonal entries of the table (marked with boxes) as the diagonal language and its complement are about these values. We can show that M_λ is not a Turing machine for $\overline{D}$ because (in this example table) λ is accepted by M_λ, putting λ in D (by the definition of D) but then a Turing machine for $\overline{D}$ should reject λ. A similar argument shows that M_0 is also not a valid Turing machine for $\overline{D}$. On the other hand, the Turing machine M_1 rejects the string 1, which means that 1 is in $\overline{D}$, but now M_1 cannot be a Turing machine for $\overline{D}$ as it should have accepted 1. Continuing this argument for each diagonal entry (whether it is A or R), shows that no Turing machine in our list of all possible Turing machines can accept $\overline{D}$. This shows that $\overline{D}$ is not recursively enumerable.

	λ	0	1	00	01	...
M_λ	$\boxed{\text{A}}$	R	R	R	R	...
M_0	A	$\boxed{\text{A}}$	A	A	A	...
M_1	R	R	$\boxed{\text{R}}$	R	R	...
M_{00}	R	A	A	$\boxed{\text{R}}$	A	...
M_{01}	A	A	R	R	$\boxed{\text{A}}$	...
...	...	...	...	...	...	...

Figure 14.3: Table of whether each Turing machine accepts/rejects each string

Theorem 14.3

The language $\overline{D}$ is not recursively enumerable.

Proof

We will prove that there cannot exist any Turing machine that accepts $\overline{D}$ via contradiction. Assume that $\overline{D}$ is accepted by a Turing machine with encoding i. We have two possible cases:
$\underline{M_i \text{ accepts } i}$: If it is the case that M_i accepts i, then by the definition of D it must be that $i \in D$. Then M_i cannot be a Turing machine for $\overline{D}$ since it should reject strings in D such as i.
$\underline{M_i \text{ rejects } i}$: If it is the case that M_i rejects i, then by the definition of $\overline{D}$ it must be that $i \in \overline{D}$. Then M_i cannot be a Turing machine for $\overline{D}$ since it should accept strings in $\overline{D}$ such as i.
We get a contradiction in both cases. Thus, a Turing machine for $\overline{D}$ cannot exist and so $\overline{D}$ is not recursively enumerable.

We can conclude from this the following result for D.

Theorem 14.4

The language D is not recursive.

Proof

We showed earlier in Theorem 14.2 that a language is recursive if and only if the language and its complement are recursively enumerable. Since we know that the complement of D is not recursively enumerable, it must be that D is not recursive.

We have shown that $\overline{D}$ and D are not recursive or computable, but they are not very natural languages. We will use them to next show that other, more easy to understand problems, are not computable as well.

14.7 REDUCTIONS

We will now show how to prove the non-computability of another language by bootstrapping from the previous two languages using a technique called a reduction. Consider the following language:

Definition 14.6

$A = \{(i, x) : M_i \text{ accepts } x\}$.

The language A consists of all pairs of Turing machine encodings and strings such that the Turing machine accepts the string. We encode pairings such as this by introducing new symbols `(`, `)`, and `,`. If we could compute this language, we could then tell whether any arbitrary program accepts an arbitrary input. Unfortunately, no Turing machine (that halts on all inputs) exists for it as we see next.

Theorem 14.5

The language A is not recursive.

Proof

We will assume that A is recursive for a contradiction. Then there exists a Turing machine M for A that halts on all inputs. We will use M to design a Turing machine for D that halts on all inputs.

We design a Turing machine M' that on input i will run M on the input (i, i) and then do what it does (accept if M accepts or reject if M rejects). We know that it is possible to design a Turing machine that runs another machine based on what we learned about the Universal Turing Machine.

We now show that M' is a Turing machine for D that halts on all inputs. This has three parts:

The Turing machine M' must halt on all inputs as the machine M halts on all inputs.

If $i \in D$, then M_i accepts i (by the definition of D), so M accepts (i, i) and thus M' accepts.

If $i \notin D$, then M_i rejects i (by the definition of D), so M rejects (i, i) and thus M' rejects.

The last two parts together give us that $i \in D$ if and only if M' accepts i, which means that M' correctly accepts all strings in D and rejects all others, making it a Turing machine for D that halts on all inputs. However, we already proved that D is not recursive, which gives us a contradiction. Thus, A is not recursive.

Intuitively, the above reduction uses the fact that determining whether M_i accepts an arbitrary string x should be even harder than determining whether M_i accepts the specific string i.

Since we believe Turing machines to be able to perform any mechanical computation possible on a computer, we can rephrase the implications of the above theorem thus:

> It is impossible to design a program that can predict the output of an arbitrary program on arbitrary input.

14.8 PROGRAM COMPARISON

We'll next see a reduction that is a little more involved. The notation $L(M_a)$ denotes the language of the Turing machine M_a.

Theorem 14.6

The language

$$SAME = \{(a, b) : L(M_a) = L(M_b)\}$$

is not recursive.

The language $SAME$ is conceptually the pairs of Turing machines that behave identically on all inputs. Translated from Turing machines to more practical usage, the uncomputability of $SAME$ implies the following:

> It is impossible to design a program that can tell if two programs behave identically on all inputs.

By showing that this problem is non-computable, this means that it is impossible for your CS instructor to design a program that takes as input two programs (for example, a working solution and a student's submission) and determine whether they behave identically on all inputs.

Proof

We will assume that $SAME$ is recursive for a contradiction. Then there exists a Turing machine M for $SAME$ that halts on all inputs. We will use M to design a Turing machine M' for D that halts on all inputs.

We design a Turing machine M' that on input i will create two Turing machines M_a and M_b as follows. The Turing machine M_a will ignore its input and instead just simulate M_i on input i doing whatever it does (accept if it accepts, reject if it rejects, or run forever if it runs forever). We know that it is possible to design a Turing machine that runs another machine based on what we learned about the Universal Turing Machine. The Turing machine M_b accepts all strings by making the start state accepting. Finally, the Turing machine M' runs the Turing machine M (for $SAME$) on input (a, b) and does whatever it does (accept or reject).

We now show that M' is a Turing machine for D that halts on all inputs. We prove the following three facts:

We know that M' halts on all inputs as the machine M halts on all inputs.

If $i \in D$, then M_i accepts i (by the definition of D) which means that M_a will always accept and hence $L(M_a) = \Sigma^*$. Moreover, $L(M_b) = \Sigma^*$ by design, so $L(M_a) = L(M_b)$ and thus M will accept, causing M' to accept.

If $i \notin D$, then M_i rejects i (by the definition of D) which means that M_a will always reject and hence $L(M_a) = \emptyset \neq \Sigma^* = L(M_b)$. Thus, M will reject, causing M' to reject.

Putting this together, gives us that $i \in D$ if and only if M' accepts, making M' a Turing machine for D that halts on all inputs. However, we already proved that D is not recursive, which gives us a contradiction. Thus, $SAME$ is not recursive.

At the heart of this reduction is the design of the Turing machine M_a that ignores its input and simulates M_i on input i. It is designed specifically so that for any $i \in D$ the Turing machine M_a will always accept giving $L(M_a) = \Sigma^*$ and for all $i \notin D$ the Turing machine M_a always rejects giving $L(M_a) = \emptyset$. This stark dichotomy allows us to use M_a in the reduction to distinguish whether a given string is in D or not by using a Turing machine for $SAME$.

More specifically, we have set up the Turing machines M_a and M_b that are given as input to the Turing machine for $SAME$ as follows:

$i \in D$?	M_i accepts i?	$L(M_a)$	$L(M_a) = L(M_b)$?	M accepts	M' accepts
Yes	Yes	Σ^*	Yes	Yes	Yes
No	No	$\emptyset$	No	No	No

The proof then takes the following form:

1. Assume for a contradiction that some Turing Machine M accepts $SAME$ and halts on all inputs.

2. Build a new machine M' that on input i halts and accepts if $i \in D$ and halts and rejects if $i \notin D$. It does so by running M (the Turing machine for $SAME$) on the Turing machines M_a and M_b described above. (This is the part of the proof that you will have to adapt, using a table similar to the one above, to solve most of the exercises.) Since M halts on all inputs, so must M'.

3. Since we showed in the last step that M' is a Turing machine for D that halts on all inputs, this shows that D is recursive. This is a contradiction, so $SAME$ can not be recursive.

14.9 THE HALTING PROBLEM

Another classic problem that is known to be uncomputable is the **Halting Problem**. Simply put, the Halting Problem asks whether a given computation will ever halt or if it will run forever. By showing that this problem is not recursive we know, for example, that it is impossible to design a compiler that can always detect infinite loop bugs in arbitrary code.

Next, we define the Halting Problem and show that it is undecidable as well.

Definition 14.7

$H = \{(i, x) : M_i \text{ halts on input } x\}$.

Notice that this definition is about whether a Turing machine halts or not on a given input, making it about the behavior of the Turing machine beyond whether it accepts certain strings or not. This makes the reduction a little more complicated.

Theorem 14.7

The language H is not recursive.

Proof

We will assume that H is recursive for a contradiction. Then there must exist a Turing machine M for H that halts on all inputs. We will use M to design a Turing machine for D that halts on all inputs.

We design a Turing machine M' that on input i will compute the encoding i' of a Turing machine $M_{i'}$ that will behave the same as M_i, except if M_i is about to halt and reject $M_{i'}$ will go into an infinite loop. Next, M' will run the Turing machine M on the input (i', i). We can imagine designing a Turing machine that automatically modifies another Turing machine to behave in the above manner since it is easy to add a state for going into an infinite loop by, for example, moving right on the tape forever. We also know that it is possible to design a Turing machine that runs another machine based on what we learned about the Universal Turing Machine.

We now show that M' is a Turing machine for D that halts on all inputs. We show the following three parts:

The Turing machine M' must halt on all inputs as the machine M halts on all inputs.

If $i \in D$, then M_i accepts i (by the definition of D), so $M_{i'}$ halts on input i, and thus it must be the case that (i', i) is accepted by M, so M' accepts.

If $i \notin D$, then M_i rejects i (by the definition of D), so by design $M_{i'}$ will reject i by running forever, and thus (i', i) is rejected by M, hence M' rejects.

The last two parts together give us that $i \in D$ if and only if M' accepts, making M' a Turing machine for D that halts on all inputs. However, we already proved that D is not recursive, which gives us a contradiction.

Thus, H is not recursive.

In practical terms:

> It is impossible to design a program that can tell if another program will ever terminate.

The Halting Problem is one of the classic problems in computer science that is uncomputable.

14.10 CLASSES OF LANGUAGES

When studying the difficulty of solving a problem, computer scientists use the notion of a class of problems.

Definition 14.8

A **class** of problem is a set of languages that share a common characteristic in terms of how hard it is to compute them.

Example 14.2

Some examples of classes of problems that we have studied in this book include the regular languages (Chapter 2), the context-free languages (Chapter 12), the recursive and recursively enumerable languages (Chapter 14).

In conclusion, here is a diagram with the containment properties of these classes.

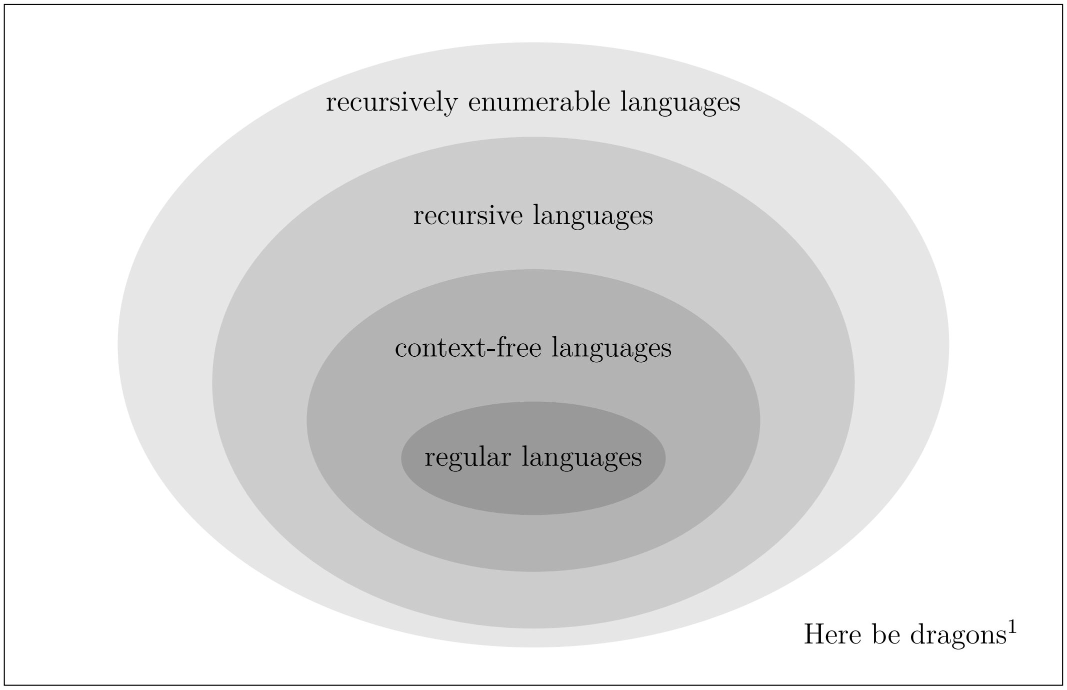

Figure 14.4: All classes of languages in this book

14.11 CHAPTER SUMMARY AND KEY CONCEPTS

- **Variations** of Turing-machine models include ones with **multiple tracks** and **multiple tapes**. These models are no more powerful than the standard Turing machine model.
- The **Church-Turing thesis** states that all mechanical computation can be performed by a Turing machine.
- The **Universal Turing machine** is one that takes as input the encoding of a Turing machine and an input and simulates that Turing machine on that input.
- A language is called **recursive** or **computable** or **decidable** if there exists a Turing machine that accepts it and halts on all inputs.

[1] You have learned that there are problems unsolvable by any computer—dragons!—hence the cover of this book.

- A language is called **recursively enumerable** if there exists a Turing machine that accepts it.
- A language is recursive if and only only if its complement is recursive.
- A language is recursive if and only if it and its complement are recursively enumerable.
- The **diagonal language** D and its complement can be shown to be non-recursive using **diagonalization**.
- **Reductions** are a way of showing that a language is not recursive. To do this, you must reduce from a problem known not to be recursive to the problem being shown is not recursive.
- Many problems, such as determining (1) whether a Turing machine (program) will accept a given input, (2) whether two Turing machines (programs) accept the same language, and (3) whether a Turing machine (program) will ever halt are all known to be non-recursive.
- A **class** of problem is a set of languages that have the same computational difficulty. These include **regular**, **context-free**, **recursive**, and **recursively enumerable** languages.

EXERCISES

Read the entries in the Stanford Encyclopedia of Philosophy on Computability (https://plato.stanford.edu/entries/computability/) and Alan Turing (https://plato.stanford.edu/entries/turing/) and other sources to give short answers to the following questions:

14.1 Who designed the Lambda Calculus?

14.2 What was Gödel's Incompleteness Theorem?

14.3 What was the *entscheidungsproblem*?

14.4 What was the title of the paper in which Turing introduced the Turing machine?

14.5 What is the Turing Test?

14.6 Do you think that a Turing machine can simulate a human brain?

Prove that the following problems are undecidable using reductions. For each problem, write a sentence explaining in words what the implication of the undecidability of the language is for computer programs.

14.7 $ALL = \{i : L(M_i) = \Sigma^*\}$

14.8 $INFINITE = \{i : L(M_i) \text{ is infinite}\}$