CS 4820: Limits of Computability

Xanda Schofield and Éva Tardos, Cornell University

1 *What is computability?*

Up until this point, the course has focused on *tractability*: whether problems can be solved by a computer algorithm in a reasonable amount of time with respect to the size of the problem. Here, we consider the broader question of *computability*, or what can be achieved by a computer at all.

Definition A problem is *computable* if an algorithm exists that can solve that problem in finite time.

Computability is a property of a computational problem. More specifically, a problem is computable if there exists an algorithm that solves the problem that can be performed by a computer with unlimited memory in finite time. We do not further specify what the upper bound of "finite time" is; it could require exponentially many steps or arbitrarily large amounts of memory to run an algorithm that solves this problem, but if it is computable, we know that the algorithm will conclude in a finite amount of time. Even with this unrestricted a bound on what is required to be computable, however, there still exist problems that are *uncomputable*, i.e. for which we can prove that no such algorithm exists to solve them.

2 *Java as a model of computation*

In order to prove things about the computability — or, more often, the uncomputability — of problems, we need some model of what a computer can do. You may in the past have encountered simple models for processing input data such as deterministic finite automata (DFAs) as seen in CS 2800. These are useful for solving specific types of problems, but they lack memory: a finite state machine only knows its current state, and in looking at the next character in an input string, it cannot reference any prior information about the string or its previous states. A full model of computation should be able to use unlimited memory, and should be capable of making decisions about what steps of computation it takes based upon any region of memory.

The most famous model used for proofs like this is called a *Turing machine*, a simplistic model created in 1936 by Alan Turing. These machines are effectively finite state machines with memory: they can only read and write one character at a time, but they have infinite space to read and write characters. They also have a finite set of states, much like finite state machines, but they can transition between these states based on reading characters written in memory.

We will discuss Turing machines later in the course.

In this course, we have used Java to write code for homework assignments, so perhaps the most natural definition of computability for the course is anything that can be computed using Java. We will assume that our Java program has access to an infinite memory. [1] To keep things simple, we will be using a simplified form of Java as our model of computation. We limit ourselves to a small portion of what you might typically use in your programs. The parts of Java that we need for computation are sequences of statements, conditionals, and loops, all of which are supported with no additional libraries, though we can assume that any code from a library may simply be added to that program. We therefore only consider a few pieces of the Java specification:

[1] We assume throughout this unit that our models of computation have infinite free memory. Obviously, real-world computers do not actually offer infinite memory, but for most reasonable computation, they offer a sufficiently large amount of memory that we don't have to worry about this bound.

- primitive data types **`int`**, **`char`**, and **`boolean`**, and basic operations for them;
- control flow in the form of **`if...else`** statements and **`for`** and **`while`** loops;
- the `String` class;
- primitive Java arrays, e.g. `String[]`; and
- the ability to create named methods callable from other methods.

Traditionally, Java methods can input and output any type of primitive or Object, while Java `main` functions must take in an array of Strings and have `void`, or nothing, as their return value. This is not to say that Java programs do not have any kind of output at all, but the output it produces is based on writing text to a buffer representing a file on a computer or text on the screen. We would like to be able to use Java programs to solve decision problems, such as the problems we discussed so far in the course. We can always think of the input as a string, but we need to find some way of retaining the true/false result of an entire program instead of a single internal method. To do this, we will assume that each program we write contains a `buffer` member variable that we can access to see what output was written by the simulated program. The primary functionality of a program will be stored in the function `execute`, which will take a list of `String` arguments, while the `main` function will be used to ensure that `execute` is used to write to the `buffer`. For decision problems, we can assume the only possible String values after a program has run are **`true`** and **`false`**. Before a program terminates, `buffer` will be equal to the empty string, ϵ or `""`. We make this variable public so that a program can use the output of this to execute another program, and then inspect the buffer afterwards.

Useful properties of computational models. For the proofs we will write, there are a few key properties of Java that are useful to observe.

1. Source code can be passed as an argument in Java code. Because a Java program can be represented by a string of the source code, we are able to consider programs themselves as objects in our code. This means we can write a method in a Java class that takes as input a String intended to represent another Java program. Note that, as a String, there is no inherent power of the code to perform computation on its own; it has to be passed to some program that can interpret that code as a set of instructions and run those instructions.

2. Java can simulate itself. Not only is it possible to pass around Java programs inside Java, it is also possible to write a Java program that can execute an arbitrary Java program so long as the code is valid Java. This is a property called *universality*: because Java can simulate any possible computation, and the execution of a Java program is a possible computation, Java can simulate itself.

While a programming language simulating itself might seem a little confusingly recursive, it is common practice in code interpreters and compilers. The first compiler for Java could not be written in Java, as nothing would exist to compile it; however, once a Java compiler exists, it is completely possible to write and execute a new Java compiler in Java, using the old Java compiler to execute it.[2] This may also be convenient for developers working on such a compiler, as they need only be proficient in Java to continue developing for it.

Though it is completely possible to write a Java interpreter in Java, doing so requires a significant amount of code and effort that is beyond the scope of this course. For our proofs, we will assume this work has already been performed. We will posit the existence a class `UniversalJavaProgram` whose `execute` method takes as input a string containing the Java program to be executed and a list of strings containing that program's arguments, returning a string output. We show the source code of this program in Figure 1.

What about errors? One question we have not resolved is we validate inputs to our programs: how do we ensure that the Strings passed to some program can be used as inputs by the `execute` function? In general, we assume that we can pass any String or array of Strings to the `main` function of this program: it is the responsibility of the program to validate the input; if it fails to validate, it should simply output **`false`** to its buffer. We assume the same thing occurs if an error is thrown: if, for instance, the `UniversalJavaProgram` is given uncompilable code, it will find this and simply write **`false`** to

[2] In fact, this has already been done for Java; see Espresso at `http://types.bu.edu/Espresso/report/Espresso.html`. More popular examples of writing language compilers and interpreters in their own language include Clang for C++ (`https://clang.llvm.org`) and PyPy for Python (`https://pypy.org`).

```java
public class UniversalJavaProgram
{
  public String buffer;

  public UniversalJavaProgram()
  {
    buffer = "";   // Where we write output
  }

  public String execute(
    String programCode,
    String[] argumentArray)
  {
    // Simulate the execution of the Java program
    // ‘programCode‘ when given the arguments
    // ‘argumentArray‘ as input, and return the output
    // (a String).
  }

  public static void main(String[] args)
  {
    // Validate the number of arguments
    if (args.length !== 2) {
      buffer = "false";
      return;
    }
    // Execute the intended program
    String programCode = args[0];
    String[] argumentArray = new String[args.length - 1];
    for (int i = 1; i < args.length; i++) {
      argumentArray[i - 1] = args[i];
    }
    UniversalJavaProgram ujp = new UniversalJavaProgram();
    ujp.buffer = ujp.execute(programCode, argumentArray);
  }
}
```

Figure 1: An outline of the interface of a universal Java program.

the buffer. In this sense, we assume that `UniversalJavaProgram` is a perfect interpreter of Java code, and has no bugs: if the code itself inputted is buggy or incorrect, it will handle the error and return **`false`**.

3 *Formalizing uncomputability*

Now that we have a model of computation, we need to formalize what it means for something to be computable under a model of computation. To simplify how we consider computation, we are going to describe all of our problems as *decision problems*, or problems with a **`true`** or **`false`** answer. While this might seem like a restriction, this does not affect what we can or cannot compute, as we can use decision problems to reconstruct more complex solutions bit by bit.

Suppose we execute some valid[3] Java program M for a decision problem with input x. There are three possible outcomes:

[3] If M itself does not compile, we treat it as rejecting all input.

1. *accept*: the program terminates and returns a **`true`** or "yes" answer,
2. *reject*: the program terminates and returns a **`false`** or "no" answer, or
3. the program *never terminates*; i.e., it reaches an infinite loop.

Definition A program M *halts* on an input x if it either accepts or rejects that input in finite time.

If the program M reaches outcome 1 or 2 for input x, where the program terminates, we say that M *halts* on input x. We define the set $\mathcal{L}_{\mathrm{M}}$ as the collection of all x accepted by M, i.e., such that $M(x)$ terminates and returns **`true`**.[4]We can think of $\mathcal{L}_{\mathrm{M}}$ as specifying a decision problem solved by M: inputs where M returns **`true`** are inputs where the answer to the decision problem is "true" or "yes," which belong in $\mathcal{L}_{\mathrm{M}}$, and inputs where M returns **`false`** or doesn't return at all are inputs where the answer is "false" or "no," which do not belong in $\mathcal{L}_{\mathrm{M}}$.

[4] The set $\mathcal{L}_{\mathrm{M}}$ of all input strings accepted by M is often referred to as the *language* of M.

There are two definitions of interest to us, then, with respect to these outcomes:

- **recognizability:** If the program M accepts all and only inputs from $\mathcal{L}_{\mathrm{M}}$—meaning outcome 1 is reached by M for input x if and only if $x \in \mathcal{L}_{\mathrm{M}}$—we say that M *recognizes* $\mathcal{L}_{\mathrm{M}}$, or that $\mathcal{L}_{\mathrm{M}}$ is a problem recognized by M. We can also consider a language of strings $\mathcal{L}$ independent of a specific program, such as "the set of all binary strings with an even number of `0`s" or "the set of all descriptions of a graph for which there exists a Hamiltonian cycle." We describe a language $\mathcal{L}$ as being *recognizable* if there exists any program M for which $M(x)$ returns **`true`** if and only if x is in $\mathcal{L}$. If x is not in $\mathcal{L}$ for some decision program M, M may either return **`false`** (outcome 2) or never terminate (outcome 3).

- **decidability:** We say that M *decides* $\mathcal{L}_{\mathrm{M}}$ if M not only returns **`true`** for all x in $\mathcal{L}_{\mathrm{M}}$, but also returns **`false`** in finite time for all x not in $\mathcal{L}_{\mathrm{M}}$. In other words, M must make the correct *decision* about whether x is in $\mathcal{L}_{\mathrm{M}}$ in finite time; it can never achieve outcome 3 above. Just as a language is *recognizable* if there exists a program that recognizes it, a language is *decidable* if there exists a program that decides it. Note that decidability is a strictly stronger requirement than recognizability; any program M that decides a language also recognizes it, and any language $\mathcal{L}$ that is decidable must also be recognizable.

We have encountered a wide variety of decidable problems so far in this course. For instance, SAT (the Boolean satisfiability problem) is NP-complete, but it is still decidable. We can show this by construction: we can write a program that, given a formula, methodically iterates through every single possible assignment of variables to **`true`** or **`false`** to see if any assignment satisfies the given formula. If the program finds a satisfying assignment, it outputs **`true`**; otherwise, after iterating through all possible assignments, it outputs **`false`**. In practice, we would never want to run this program for large n, as for n different variables, it would take exponential time, $O(2^n)$, before it would output **`false`** for an unsatisfiable formula. However, exponential time is still finite, so this program still decides SAT.

When we prove a problem to be uncomputable, we specify one of the definitions above which the problem does not satisfy. For example, we might want to prove that a problem described by some language $\mathcal{L}$ is *undecidable*, i.e., that there exists no program M that can for any input x determine whether x is in L in finite time. Much like in NP-completeness, for most undecidable problems, we prove that they are undecidable using reductions: if we can reduce a known undecidable problem to an unknown problem, then we can show that unknown problem is also undecidable. However, just like in NP-completeness, to use reductions, we need to bootstrap our set of undecidable problems by proving a single problem is undecidable without a reduction. For NP-completeness, that problem is SAT; for uncomputability, that problem is the halting problem, which we will get to in the next section.

4 *Diagonalization and the halting problem*

When first learning recursion in a programming class, you may have accidentally written code that infinitely recursed, producing a notorious "stack overflow" error. As your computer wound to a halt, you may have wished that there were some way to have known

this would happen before the code ran. Wouldn't it be great if there were a program that, given the code for any program, M, and some input to that program, x, would tell you whether or not M would finish running on x in finite time?

This problem is called the *halting problem*, and we can write it out formally as a language $\mathcal{L}_{\text{HALT}}$:

$$\mathcal{L}_{\text{HALT}} = \{\langle M, x\rangle | M \text{ is a Java program, and computing } M(x) \text{ halts.}\}$$

We use the notation $\langle M, x\rangle$ to refer to a tuple of two strings. We can represent these in one string by having a special reserved character that we use to show where one part of the input starts and the next ends.

We can first establish that this problem is *recognizable*: to prove it, we can simply imagine a program $H(M, x)$ that first executes program M on input x and, after the program halts, returns **`true`**. We can prove that this program H recognizes $\mathcal{L}_{\text{HALT}}$ by showing that it returns **`true`** if and only if M halts on input x:

- if M's execution halts on x, then H's simulation of M's execution on x will also halt, and H will subsequently return **`true`**;
- if H returned true, then H must have finished simulating M running on input x in finite time, implying that it halted.

Of course, recognizability does not really solve the problem we are interested in: the case that a student learning recursion would be interested in is not learning when a program will halt, but learning when it won't. Unfortunately for them and for us, however, $\mathcal{L}_{\text{HALT}}$ is not decidable. To prove this, we will use a tool called *diagonalization*.

A quick tutorial on diagonalization

Diagonalization is a proof strategy that was first introduced by Georg Cantor in 1873 for describing relative magnitudes of infinite sets. The observation Cantor used was that, in order to show two different sets were of the same size, one could use a matching argument: if every single element of a set A is matched to exactly one element from set B and vice-versa, then the sets must have the same number of elements. This one-to-one matching, or *bijection*, can be extended to sets of infinite sizes: for infinite A and B, if there is a bijection from A to B, then A and B have the same *cardinality* of infinity.

Cantor used this to show that there were multiple cardinalities of infinity. For instance, one cardinality is represented by *countability*, where a set S is countable if there is a bijection from S to the natural numbers $\mathcal{N} = 0, 1, 2, 3, \ldots$. Integers and all possible strings over a fixed set of characters are countable: you can write a function that enumerates every possible string (starting from the empty string ϵ, 'a', 'b', ..., 'aa', 'ab', and so on) such that, given a string, you could compute the finite index corresponding to that number.

number	digits	1	2	3	4	5	6	
$real(1)$	0.	$\underline{\mathbf{2}}$	1	2	5	6	7	...
$real(2)$	12.	6	$\underline{\mathbf{3}}$	3	3	6	7	...
$real(3)$	1.	5	0	$\underline{\mathbf{0}}$	0	0	0	...
$real(4)$	-5.	9	8	1	$\underline{\mathbf{4}}$	4	8	...
...								
r	0.	3	4	1	5			...

Table 1: How we choose digits for assembling r based on a function *real* enumerating real numbers. Notice how we specifically differentiate each digit of r from a digit on the diagonal produced by aligning the real numbers—hence, *diagonalization*.

However, real numbers are not countable. We construct a proof by contradiction of this by iterating through an imagined enumeration of real numbers and constructing an element that we show is not in the enumeration of those reals. Suppose we had some enumeration function $real(n)$, where n is a positive natural number input and $real(n)$ outputs the nth real number in the enumeration starting from $n = 1$. We will also write a helper function, $digit(s, i)$, which returns the ith digit after the decimal place of a real number s. Construct a real number r as follows:

- r has no nonzero digits before the decimal point.
- r's first digit after the decimal point is a digit in the range $(1, 8)$ [5] that is *not* the first digit after the decimal point for $real(1)$; that is, $digit(r, 1) \neq digit(real(1), 1)$.
- r's second digit is another digit in the range $(1, 8)$ that is not equal to the second digit of the second real number, or $digit(real(2), 2)$.
- We continue to generate digits infinitely, with the ith digit of r being some digit between 1 and 8 inclusive such that $digit(r, i) \neq digit(real(i), i)$.

[5] We are using (1, 8) instead of (0, 9) to avoid the problem where $0.999999\cdots = 1.00000\ldots$. However, this still leaves more than enough options to ensure that we can choose a digit that does not match $digit(real(1), 1)$.

Consider the real number r produced this way. We know from the way it is constructed that it cannot equal any enumerated real number produced by *real*, as for any output $real(n)$, r will differ from it by at least the nth digit. We also know it is a real number, as it can be represented using a decimal representation, even if that representation is infinite. Thus, for any possible enumerative function *real*, we can construct a real number that it will never produce, implying that no such valid enumeration function could exist. This disproves the possibility of a bijection existing between the natural numbers and real numbers, and thus shows that the real numbers are not countable.

This argument relies on constructing r to differ explicitly from every single thing in our infinite list. The specific different element we use comes from the "diagonal" of the list: we differ from number i at the ith digit, and so on. We use this intuition to describe a style

Definition *Diagonalization* is a proof argument structure that shows that a set of elements cannot be enumerated by supposing the existence of an enumerated list of the elements from the set, then constructing some element that should be on that list but explicitly differs from every element in the list.

of proof argument called *diagonalization*, in which we argue that we cannot make a comprehensive list of some set of things (e.g., the real numbers) by constructing an element that should belong in such a list, but would explicitly differ from every other element in that comprehensive list of things (e.g., r). We can use this same kind of construction in the case of the halting problem to construct an adversarial program, `Diagonalizer`, that makes it impossible for a hypothetical program that decides the halting problem, `HaltChecker`, to work. Next, we show the proof by contradiction that does this.

Proving the halting problem undecidable

Now we prove the primary result of this section: that the halting problem is undecidable.

Theorem 1 (Undecidability of the Halting Problem.)**.** *Consider the following decision problem that takes as input program M and input to the program x:*

$$\mathcal{L}_{\text{HALT}} = \{\langle M, x\rangle | M \textit{ is a Java program, and computing } M(x) \textit{ halts.}\}$$

This problem, called the halting problem*, is undecidable.*

Proof. We know if a language is decidable, then there exists some program that actually decides it. So, let's assume we have access to that program, `HaltChecker`:

- **Arguments:** Our program takes in two arguments: `program`, the source code for a Java program, and `input`, the input that should be fed to the Java program.
- **Output:** For *any* possible `program` and `input`, `HaltChecker` will output *in finite time* whether the execution of `program(input)` will complete in finite time (either **`true`** or **`false`**).

Now, we will construct an additional program that uses `HaltChecker` as a subroutine. This program is going to force `HaltChecker` to contradict itself. Let's call this program `Diagonalizer`. We show the outline of its source code in Figure 3.

The core method of `Diagonalizer`, `execute`, takes in only one string as an argument: the code of a program `program`. The method uses a `HaltChecker` to check if `program` halts when fed *its own source code* as input.[6] If `execute(program, input)` returns **`true`**, meaning the program would halt, then we will make the `execute` method go into an infinite `while` loop. However, if `execute(program, input)` returns **`false`**, meaning the program would not halt, then we will tell our `execute` method to immediately return **`true`** and halt.

[6] This might seem like a nonsense input for most programs, but if the code either fails to validate the input or throws some kind of error, we can still treat that as giving a **`false`** return value in finite time.

```java
public class HaltChecker
{
  public String execute(String program, String input) {
      // returns "true" if program would halt on input,
      // otherwise returns "false".
      // Note this cannot do this by calling execute on
      // program and input, as this might never terminate.
  }

  public static void main(String[] args)
  {
    String program = args[0];
    String input = args[1];
    HaltChecker hc = new HaltChecker();
    hc.buffer = hc.execute(program, input);
  }
}
```

Figure 2: Skeleton code for a HaltChecker.

Just like how we constructed a real number different from any other possible real number in our list, `Diagonalizer` explicitly behaves differently from each other program M on at least one input: the source code corresponding to that M. This relies on the countability of strings from before: we can enumerate every possible string input, and for each input, also try to treat that same input as a source code M. If some M does not have valid source code, then we may instantly halt and reject when trying to simulate M. This is enough for the `Diagonalizer`: no matter the reason M halts when fed its own source code, whether it failed to compile M or immediately accepts the input M, the `Diagonalizer` will take this predicted **`true`** outcome from `haltChecker.execute(`M`, `M`)` and proceed to loop infinitely.

If M halts on itself, the `Diagonalizer` will not halt; if M does not halt on itself, `Diagonalizer` will halt. Now comes the weird part. What happens if we feed our halt-checking program two copies of the source code of `Diagonalizer`? This would require the `Diagonalizer` to differ from itself: because `Diagonalizer` is a valid program if `haltChecker` exists, it must disagree with itself when fed its own source code as input. This is enough to produce a contradiction - this `haltChecker` cannot be part of a program.

We can dig more deeply into where a contradiction arises by looking at what must happen in execution: if `outsideHaltChecker` is a `HaltChecker` and `String diagonalizerCode` is the code for

```java
public class Diagonalizer
{
  public HaltChecker haltChecker;

  public Diagonalizer(String) {
      haltChecker = new HaltChecker();
      this.input = input;
  }

  public String execute(String program) {
      String wouldTerminate = haltChecker.execute(program, program);
      if (wouldTerminate == "true") {
          while (true) {
              continue;
          }
      } else {
          return "true";
      }
  }

  public static void main(String[] args)
  {
    String program = args[0];
    Diagonalizer diag = new Diagonalizer();
    diag.buffer = diag.execute(program);
  }
}
```

Figure 3: Skeleton code for our Diagonalizer.

`Diagonalizer`, we could define the boolean `out` as:

```
boolean out = outsideHaltChecker.execute(diagonalizerCode, diagonalizerCode);
```

After this line executes, what does the variable `out` equal?

- Suppose `out` is **`true`**. This implies that the `Diagonalizer` halts when fed its own source code. However, this only happens if `haltChecker.execute` returned **`false`** inside the `Diagonalizer`. This indicates that the two `HaltCheckers`, `haltChecker` and `outsideHaltChecker`, disagreed on the same inputs, two copies of `diagonalizerCode`: `haltChecker` thought that `Diagonalizer` would not halt, but `outsideHaltChecker` did. This is a contradiction, so `outsideHaltChecker` could not have returned **`true`**.
- Suppose `out` instead is **`false`**. This implies the opposite of before, that the `Diagonalizer` would not halt when fed its own source code. However, this infinite loop only happens if `haltChecker.execute` returned **`true`**. This again implies that the two `HaltCheckers`, `haltChecker` and `outsideHaltChecker`, disagreed on the same inputs: : `haltChecker` thought that `Diagonalizer` would halt, but `outsideHaltChecker` did not. Because of this contradiction, `outsideHaltChecker` could not have returned **`false`**, either.

In effect, we have shown that there is no possible way for a hypothetical `HaltChecker` to give a correct answer about whether `Diagonalizer` would halt if fed its own source code. However, we said `HaltChecker` *decided* this problem: for it to do so, it must be able to return a correct answer for *any* program in finite time, even our adversarial `Diagonalizer`! So, no such `HaltChecker` can exist. And, if no program can exist to decide the halting problem $\mathcal{L}_{\text{HALT}}$, then $\mathcal{L}_{\text{HALT}}$ is undecidable. □

5 *Uncomputability via reduction*

The proof that the halting problem is uncomputable relies on a cool diagonalization that allows us to "trick" our imaginary halt-checker into contradicting itself. However, constructing this adversarial example is a little complex and confusing. Instead, as in many other places in algorithmic analysis, we often use *reductions* to prove that problems are uncomputable. Much like when we prove a problem NP-hard, we can use a reduction to show that a program that decides $\mathcal{L}_{\text{NEW}}$ could be used as a subroutine to decide something we know is undecidable (e.g., $\mathcal{L}_{\text{HALT}}$, the halting problem). The direction of this is important: when we reduce $\mathcal{L}_{\text{HALT}}$ to $\mathcal{L}_{\text{NEW}}$, we show that if $\mathcal{L}_{\text{NEW}}$ were decidable, it would make the impossible possible. We use the

notation $\mathcal{L}_{\text{HALT}} \leq \mathcal{L}_{\text{NEW}}$ to indicate that $\mathcal{L}_{\text{HALT}}$ is "at least as easy as" $\mathcal{L}_{\text{NEW}}$. This means is $\mathcal{L}_{\text{HALT}}$ is impossible to decide, then $\mathcal{L}_{\text{NEW}}$ must be "at least" impossible.

Let's take as an initial problem one very similar in description to the halting problem, the *acceptance problem*. This problem has the same inputs as the halting problem, a program M and an input x, but instead asks: does this program *accept* this input? Unsurprisingly, given how close this problem sounds to the halting problem, the acceptance problem is also undecidable, which we will prove below.

Theorem 2 (Undecidability of the Acceptance Problem.)**.** *Consider the following decision problem that takes as input a program M, and an input to the program x:*

$$\mathcal{L}_{\text{ACCEPT}} = \{\langle M, x\rangle | M \textit{ is a Java program, } x \textit{ is a string input, and } M(x) \textit{ returns } \texttt{\textbf{true}}.\}$$

This problem, called the acceptance problem*, is undecidable.*

It is totally possible to recreate the diagonalization argument for this problem to prove it is uncomputable. However, we will use a much simpler strategy: we will *reduce* the problem we already know is undecidable, the halting problem, to our new problem, the acceptance problem. In other words, we show that if we had a decider for the acceptance program, we could program a decider for the halting problem. This implies a contradiction: a decider for the halting problem cannot exist, so a decider for the acceptance problem could not exist, either.

Proof. Suppose we have some program `AcceptanceChecker` that decides the acceptance problem. We can write a `HaltChecker` program as follows: for any input program M to the `HaltChecker`, we first create a slightly modified version of this program $M'(x)$ that first calls $M(x)$, then returns **`true`** after $M(x)$ finishes no matter what it returned. `HaltChecker` then passes the source code of M' and x to the `AcceptanceChecker`.

- If the `AcceptanceChecker` returns **`true`**, then we know $M(x)$ halted: otherwise, we would not ever reach the point where M' returns.
- If the `AcceptanceChecker` returns **`false`**, then we know the program inside the wrapper never halted on its input, as otherwise M' would have returned **`true`**and thus accepted x.

This program for the `HaltChecker` decides the halting problem, so $\mathcal{L}_{\text{HALT}} \leq \mathcal{L}_{\text{ACCEPT}}$. This cannot exist, so we cannot have a decider for the acceptance problem. □

As another example, consider what looks like a simpler problem: we can't tell if a program halts, but can we tell if it at least accepts the input `0`? We describe $\mathcal{L}_0$ as the set of programs (represented by their source code) that will return **`true`** for the input `0`. We will show that even this simple condition is undecidable: no program exists that can determine in finite time for all programs whether or not they will accept `0`.

Theorem 3 (Undecidability of the Zero-Input Problem.)**.** *Consider the following decision problem that takes as input program M:*

$$\mathcal{L}_0 = \{\langle M \rangle | M \textit{ is a Java program, and } M(0) \textit{ returns } \textbf{\texttt{true}}.\}$$

This problem is undecidable.

Proof. We will suppose there exists a program `ZeroChecker` that does this, with a method `execute(String program)` that takes in the source code for a program and returns **`true`** if that program accepts the input `0`, otherwise **`false`**. The way we are going to use this program to create a halt checker is by "wrapping" any given program in a way that corresponds with our problem. In this case, we write out source code for some `ProgramWrapper` class that is initialized with the source code for one class, `program`, and the arguments to that program `input`. The `execute` function of this wrapper is going to take as an argument any string, but it will ignore it, and instead executes `program` on `input` (based on the code from our UniversalJavaProgram). Once `execute(program, input)` terminates, this program will return **`true`**.

We show that this wrapper and the existence of our program `ZeroChecker` are all we need to write a new program for a `HaltChecker`. To do this, we write the `HaltChecker`'s `execute` method as follows: first, we use our Wrapper class:

```
wrappedProgram = ProgramWrapper(program, input).
```

We then pass the *source code* of `wrappedProgram` in to `ZeroChecker`. The output of the `ZeroChecker` then can be passed on as the return value of our `HaltChecker`:

- If the `ZeroChecker` returns **`true`**, then we know the program inside the wrapper would halt on its input, as the wrapper would never execute the step returning **`true`** otherwise.
- If the `ZeroChecker` returns **`false`**, then we know the program inside the wrapper would never halt on its input, as if it did, then it would necessarily execute the step that returns true.

These conditions show that the input program, `program`, will halt if and only if the wrapped version of that program is accepted by

the `ZeroChecker`. We also observe that if `ZeroChecker` decides $\mathcal{L}_0$, then this program will always finish in finite time, as the other steps all conclude in finite time. We have therefore reduced $\mathcal{L}_{\text{HALT}}$, the halt-checking problem, to $\mathcal{L}_0$, checking if a program accepts a zero input: $\mathcal{L}_{\text{HALT}} \leq \mathcal{L}_0$. This is equivalent to saying that the halting problem is *at least as easy as* the acceptance of a zero input problem in the space of computability: if we can compute $\mathcal{L}_0$, we can compute $\mathcal{L}_{\text{HALT}}$, too. We have proven this constructively: if we can create a `ZeroChecker` that decides whether a program accepts the input `0`, then we know how to use that to write a `HaltChecker` that decides the halting problem. But of course, no `HaltChecker` can exist, so we have arrived at a contradiction: a `ZeroChecker` program deciding $\mathcal{L}_0$ cannot exist, either. We therefore know that $\mathcal{L}_0$ is an undecidable language as well. □

6 *Rice's Theorem*

The program that we showed could not exist above, `ZeroChecker`, is performing a particular type of check: it takes in the source code of a program, and decides something about what inputs that program accepts or rejects. This is an important distinction from other questions we could ask about an input program, like whether the program itself contains a `for` loop, or whether it takes more than 10 steps to process the input `0`. These two questions are about the content of the input program and how the program would be executed, and they are decidable: we can write a program that looks for the syntax of a `for` loop or that executes the first ten steps of a program to see if it halts.

However, the `ZeroChecker` is looking at properties of the *language* accepted by an input program. Because `ZeroChecker` is checking something about the final behavior of the program, whether it accepts or rejects a specific input regardless of the execution up until that point, it is possible to turn it into a halt-checker by wrapping an arbitrary program in code that matches the specific condition `ZeroChecker` cares about. We call a property of the language that a program M accepts a *semantic property of of the program*, distinguishing it from properties that consider the program itself, and not the language the program accepts.

In fact, there is nothing special about checking if `0` is in the language accepted, we can make the same argument for *any nontrivial* property of the language that a program accepts. We say that a property of the language is nontrivial if there are languages that satisfy the property, and there are also languages that do not satisfy this property. For example, whether `0` is in the language is a nontrivial

property, some languages contain `0` and others do not. This result is called Rice's Theorem after Henry Gordon Rice, who published this result in 1951.

Theorem 4 (Rice's Theorem)**.** *Take $\mathcal{L}$ to be a semantic property of programs, i.e., one where the membership of a program M in $\mathcal{L}$ is determined by some condition on which inputs M accepts. If $\mathcal{L}$ is not vacuously true or false for all programs, then it is undecidable.*

Proof. To prove this, we will generalize the intuition we used for `ZeroChecker`: to an arbitrary decision problem $\mathcal{L}_{\text{I}}$ that is nontrivial, i.e. for which there is at least one program A_i that is in $\mathcal{L}_{\text{I}}$ and at least one program A_i' that is not in $\mathcal{L}_{\text{I}}$. More specifically, we will assume the program A_i' not in $\mathcal{L}_{\text{I}}$ is the program that loops infinitely for all inputs, therefore not accepting any input. We can assume this because, if this program was in $\mathcal{L}_{\text{I}}$, we could simply prove the undecidability of the complement of $\mathcal{L}_{\text{I}}$ instead, using the method we will describe. We can do this because the complement of an undecidable language is also undecidable.[7]

[7] If the complement were decidable, one could simply negate the output of the program that decided it to produce a program that decided the original program.

Suppose by way of contradiction that we have some general program, `PropertyChecker`, that decides whether or not a property holds for some input program M. That is, `PropertyChecker.execute(`M`)` will return **`true`** if M is in $\mathcal{L}_{\text{I}}$, and **`false`** if M is not in $\mathcal{L}_{\text{I}}$ in finite time. We want to describe a way to turn an arbitrary input to the halting problem of program and input $\langle M, x\rangle$ into an input that `PropertyChecker` can decide.

We therefore use the following protocol to make a hypothesized `HaltChecker.execute(`M, x`)`: first, we generate the source code for a program $M'(y)$ that does the following:

- Executes program M the corresponding input x,
- If M halts on x, then execute program A_i on this y, and return the result.

Note that M' is specified for a particular M and x; you can think of these as "magic numbers" or constants in the code that is written out based on the halting problem. Writing this code itself takes finite time; it is a deterministic modification of the existing source code given A_i and A_i'. After constructing the source code for M', the `HaltChecker` will feed M' to `PropertyChecker` and output the result.

We can show that if `PropertyChecker` decides the property $\mathcal{L}_{\text{I}}$, then this program will decide the halting problem:

- If M halts on input x, then M' will respond exactly like A_i for all inputs. By definition, this means the program is in $\mathcal{L}_{\text{I}}$. `PropertyChecker.execute(`M'`)` therefore must return **`true`**.
- If M does not halt on input x, then M' will not halt on any input. This means the program will behave like A_i', the program that

halts on no inputs, which is not in $\mathcal{L}_{\mathrm{I}}$. `PropertyChecker.execute(`M'`)` therefore must return **`false`**.

The program sketched above for `HaltChecker` never executes M or M'. Instead, it relies only on the existence of a program `PropertyChecker` that decides $\mathcal{L}_{\mathrm{I}}$, i.e., that always returns a result in finite time. Because a program that decides $\mathcal{L}_{\mathrm{HALT}}$ cannot exist, we arrive at a contradiction, implying that no such `PropertyChecker` program can exist. This is sufficient to show that $\mathcal{L}_{\mathrm{I}}$ for an arbitrary semantic property is undecidable. □

Rice's Theorem is a powerful tool for proving undecidability: it takes the structure of the proof we used to reduce the halting problem to `ZeroChecker`, and generalizes it to any nontrivial property of a problem space. However, it is important to remember that this is limited to properties that only look at the output. If `PropertyChecker` were to measure some other property about how M' is executed on input y, it might not consider M' to behave equivalently to A or A'.

7 *Turing Machines*

Next, we'll talk about *Turing machines*. Until now, we used a simple form of Java as our model of computation. Working with simple Java is more intuitive, as Java is a programming language we all use a lot, but it required some awkwardness in earlier sections to formalize it as a model for solving decision problems. In contrast, extreme simplicity of Turing machines makes them easier to use in proofs, but much more awkward as a real form of computation.

A *Turing machine* is effective a form of a Deterministic Finite Automata (DFA) with memory. Turing machines are described using low-level operations that consider only one position in memory at a time. While this can make writing programs for them difficult, the simplicity of their operations makes them easy to use in proofs: if there are only a few types of action a Turing Machine can take, it is easier to exhaustively prove a property holds for any possible behavior of a Turing machine. According to the Church-Turing Thesis, an unprovable, but widely upheld computer science hypothesis, any computation that can be performed on any kind of computer can also be performed on a Turing machine.[8] We can prove more specific equivalences between Turing machines and programming languages: for instance, that Turing machines can compute any function that we can compute with one of our simple Java programs, and the converse, that anything computable by a Turing machine is also computable by a simple Java program.

[8] This hypothesis is unprovable because we don't have a definition of what we mean by "any kind of computer."

Recall the definition of a Deterministic Finite Automata (DFA).

An example of a DFA is depicted on the graph in Figure 4 below. The DFA has a set of states K, nodes on the graph below with a start state $s \in K$, and a transition function $\delta : K \times \Sigma \to K$, where Σ is an *alphabet*, a set of possible symbols that could appear in the input. The δ function is represented by the edges in our graph, where edges are labeled with the symbols in our alphabet $\sigma \in \Sigma$. Each node $q \in K$ has an edge leaving q labeled with each possible symbols. The DFA will read the input symbol-by-symbol, starting at state s: for each symbol, it will transition using the rules of the δ function.

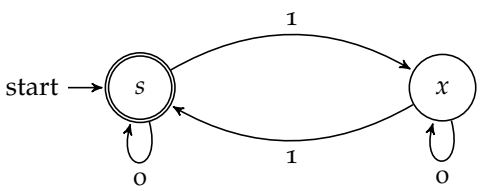

Figure 4: The DFA counting the parity of the number of 1st in a `0/1` input string.

The DFA in the example starts at state s and switches between states s and x each time it sees a `1`. This means it will return to state s whenever it has read an even number of `1`s, and will be in state x when it has read an odd number of `1`s. While tracking even or odd numbers of `1`s is straightforward for a DFA, other problems are harder. You may recall from CS 2800 that while a DFA can check if the number of `1`s is even, it cannot check if the string has the same number of `1`s and `0`s. The main limitation of a DFA in this case is that it cannot write: to remember an unbounded number, such as how many more `1`s than `0`s have occurred so far, would require an infinite number of states to store every possible value of that number. Without memory and with only finite states, this problem is impossible for a DFA.

A Turing machine also has a set of states K with a start state s, just like a DFA, but it also has an infinite amount of space to write and remember information, which we call the *tape*. We think of the tape as an infinite array of positions $M[0, 1, \ldots]$. The Turing machine will also have a reading head that points to some single position on the tape. The head of the Turing machine acts like a DFA: based on an observation about input, it decides which state to go to next. Specifically, at each time step, if the reading head is currently pointed at position i, it

(i) reads the character at position i,

(ii) chooses which state to be in next,

(iii) possibly writes a new character at position i, and

(iv) moves one position left or right or stays in the same position.

We'll need a few special characters to define things. Let the character μ denote that a position is blank. We also define $\triangleright$ as a special character used at the left end of the tape ($M[0] = \triangleright$) to let the reading head know not to move further left than this position. A Turing machine starts with the input of length n written on positions $M[1, \ldots, n]$, $M[0] = \triangleright$, and $M[i] = \mu$ for all $i > n$.

As an example, any DFA can also be implemented as a Turing machine. To implement the DFA given above, the machine will have two states $\{s, x\}$ with s as the start state. It will move one to the right after reading any non-blank symbol, and change its state whenever it sees a 1. The machine will be in state x when reaching the first black if the input has an odd number of 1s. However, this program uses very little of the power available to Turing machines: it does not write any characters, nor does it attempt to move left. These simple abilities to write and move in either direction make Turing machines much more powerful than DFAs.

Formalization of a Turing Machine We provide a consolidated formal definition of a Turing machine as a combination of the following:

- A set of **states** K with a start state $s \in K$ and two special states r and a. The machine will enter one of these two special states when it finished computation and then remain in this state indefinitely, effectively *halt*: state a means that the machine accepts, while state r means that the machine rejects.[9]
- An **alphabet** or set of symbols Σ, with two special symbols $\triangleright, \mu$ that are outside of the input alphabet and reserved only for the start of the tape and to mark blank tape positions, respectively.
- A **reading head** of the Turing machine that points to a specific tape position $h \geq 0$ at all times, starting at $h = 0$.
- Finally, a **transition function** $\delta : K \times \Sigma \to K \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. We interpret the transition function as follows: if $\delta(q, \sigma) = (q', \sigma', \rightarrow)$ then when in state q with the dead at position h, reading letter $M[h] = \sigma$, the Turing machine writes $M[h] = \sigma'$, transitions to state q', and moves the head h one right, while - is interpreted as the head not changing position.

[9] To turn our previous parity checker into such a machine, we need to add two new states a and r, and make the machine transition to these states when it sees the first μ character, choosing between the two depending on which of s or x it is in.

We say that the Turing machine *halts and accepts* if it enters the state a, *halts and rejects* if it enters the state r, and doesn't halt if it never enters these two states. This corresponds with the three conditions from Section 3: just like Java, the options for a decision problem are to accept, reject, or never terminate.

The definitions for decidability and recognizability also match those we gave when describing Java as a model of computation. We

can consider a decision problem to be expressed as a *language*, or a set of strings $\mathcal{L}$ containing all input strings for the decision problem that should output **`true`**. We say that a problem or language $\mathcal{L}$ is *recognizable* by a Turing machine if there is a Turing machine M such that, when run on input x, it accepts if and only if $x \in \mathcal{L}$. (The program may either reject or not terminate on inputs not in $\mathcal{L}$.) A language is *decidable* by a Turing machine if there exists a Turing machine M such that when run on input $x \in \mathcal{L}$ it halts and accepts, and if run on any other input it halts and rejects.

A language is recognizable by a Turing machine if and only if it's recognizable by a Java program, and similarly a language is decidable by a Turing machine if and only if it's decidable by a Java program. This is a provable part of the Church-Turing thesis: for the special case of Turing machines and Java programs, this assertion is not that hard to prove. However, it is rather tedious, so we will not include the proof here. Instead, we will illustrate the power of a Turing machine by a few simple examples.

Consider an input string that is defined to be a sequence of n 1s. We have seen above that the parity of 1s can be decided by a DFA, and also by a Turing machine. Now, we want to check if n, the number of 1s in the input, is a power of 2. We can do so in a couple of passes, using our ability to move the head back on the tape, and to write. Before we formalize the proof, we want to outline the main idea:

(a) If there is a single 1 on the tape, we accept; if there are no 1s at all, we reject.

(b) Otherwise, starting at state s, the head moves right every step, changing between two states (say x and y) each time it sees a symbol $\sigma = 1$. However, in state y, the state it enters after seeing an odd number of 1s, we will also write $\sigma' = 0$ to replace the 1 that we see while transitioning to state x. If this process ends in state y we reject, as we had an odd number of 1s in that pass.

(c) At the end of each pass, return to the left end of the tape and start again at state s, repeating until it rejects or there is only a single 1 left on the tape in which case it accepts.

To implement this process by a Turing machine, we need to add a few more details:

(a) We need a start state s. In this state, we move right until we see a 1 or μ. If we see μ, we switch to state r and reject; if we see a 1 (a first 1), we enter a new state s' and continue right, If we see μ before another 1 on the tape, we have reached the end of the

input, so we enter state a to accept. If we see a second 1, we enter a new state b (for going back).

(b) In state b, we go left until we reach the start symbol $\triangleright$ without writing or changing states. When the head reaches $\triangleright$, we change the state to x, corresponding to having observed an even number of 1s so far (in this case, 0).

(c) In state x, we read through the tape, moving right at each step until we reach a symbol $\sigma = 1$. If we reach this symbol, we transition to state y, indicating we have seen an odd number of 1s. If we see a μ, we have reached the end of the input and observed an even number of 1s, so we enter a new state b'.

(d) In state y, similar to state x, we continue to read the tape and progress one step right at each step until we see a symbol $\sigma = 1$. In this case, we replace that 1 by writing $\sigma' = 0$ and transition back to state x. If we see a μ in the tape while in state y, we have an odd number of 1s on the tape, so we switch to state r to reject.

(e) If we reach state b', there were an even number of 1s on the tape, and half of these 1s have been replaced with 0s. In state b', we go left until we reach the input start symbol $\triangleright$ without writing or changing states. When reaching $\triangleright$, we change the state back to s and start again.

A second example is to consider an input a sequence of 0s followed by 1s, and we want to decide if the number of 0s is the same as the number of 1s. A Turing machine can go back and forth on the tape; at each turn, it can change one 0 to a 2, and one 1 to a 2. If this process ends with a tape of all 2s, the machine accepts; if there were any 1s or 0s left, the machine rejects.

These two example illustrate that much of what is computable by traditional computers can also be done by a Turing machine, though doing this on a Turing machine is more complicated and much slower. In particular, executing these programs involves making the head run back and forth along the tape, which takes much more time than accessing variables in memory on a typical computer. Perhaps surprisingly, however, anything computable by a Java program in polynomial time is also computable by a Turing machine in polynomial time. The Turing machine, of course, will be quite a bit slower: if a Java program can do a computation in time T using D memory, then an equivalent program on a Turing machine will take at least $O(TD)$ time, as between every two steps of the Java program, the Turing machine will have to slowly move to the location of the new part of the memory needed. However, the amount of memory a T

time Turing machine can use is at most $D \leq$T, so a computation is in polynomial time as a Java program, it remains polynomial time when described for a Turing machine, only the degree of the polynomial increased by maybe a factor of 2, but remains a constant. The idea of a Turing machine is to offer a super simple abstraction of a computation, not to actually use it to compute.

8 *SAT is NP-complete*

As an application to our definition of computation using Turing Machines, we can prove that SAT is NP-complete. Recall that we have been using SAT as our "first" NP-complete problem, and showed other problems NP-complete by reductions from SAT. We still need a proof that SAT is actually NP-complete.

Theorem 5 (Cook-Levin Theorem)**.** *SAT is NP-complete.*

Proof. We know that SAT is in NP. To prove that SAT is NP-complete, we need to show that for any other problem $\mathcal{L}$ in NP, $\mathcal{L} \leq_P$ SAT, where we use $\leq_P$ to denote the polynomial time reductions we used for proving problems NP-complete.[10]

[10] We use $\leq_P$ to distinguish polynomial-time reductions from the $\leq$ reduction used for general computability.

Because the problem $\mathcal{L}$ is in NP, we know there exists a polynomial time verification algorithm for the problem, an algorithm `Checker` that takes two inputs x and y with $|y| \leq |x|^\ell$ for some know constant ℓ. The algorithm `Checker` satisfies the following

- For any input x, if $x \in \mathcal{L}$, then there exists a y with $|y| \leq |x|^\ell$ such that `Checker`(x, y) accepts in time at most $O(|x|^k)$ for constants ℓ and k.
- For any input x, if $x \notin \mathcal{L}$, then for any input y, `Checker`(x, y) rejects in time at most $O(|x|^k)$ for constant k.

As a first step in our proof, we replace the algorithm `Checker` with a Turing Machine, `CheckerTM`, which can do the same computation by our discussion in the previous section. This change in the form of computation can increase the running time of the `CheckerTM`. We will assume that the running time is bounded by at most $T = a|x|^c$ for constants a and c. Note that in T time, the Turing machine can use at most the first T positions on the tape.

Now we area ready to show that $\mathcal{L} \leq_P$ SAT. To define the SAT problem, we start by defining a large number of variables.

- We use a variable $\xi_{i,t,\sigma}$ for each position $1 \leq i \leq T$ of the tape, each time step $1 \leq t \leq T$, and each symbol $\sigma \in \Sigma$. The idea here is that we will set $\xi_{i,t,\sigma} = True$ if and only if at time t, the character $M[i]$ has value σ. It is enough to consider the first first T positions

of the tape, as in T steps the Turing machine will not get to further away positions.

- We set a variable $\nu_{t,i}$ for each position $1 \leq i \leq T$ of the tape. For each time step $1 \leq t \leq T$, $\nu_{t,i}$ will be *True* if the head is in position i at time t and *False* otherwise.

- We set a variable $\zeta_{t,q}$ for each state $q \in K$ and each time step $1 \leq t \leq T$, which will be *True* if the state of the machine at time t is q and *False* otherwise.

Note that the number of variables is is polynomial: we have at most $T^2|\Sigma|$ variables of the type $\xi_{i,t,\sigma}$,at most T^2 of the type $\nu_{t,i}$, and at most $T|Q|$ variables of the type $\zeta_{t,q}$. This is polynomial in T (as $|Q|$ and $|\Sigma|$ are constants), and T is polynomial in the input size.

Next, we need to encode as a SAT formula the rules that make this a valid accepting computation of `CheckerTM`. While we will not provide the exact formalization of every clause that must be made, we will list all the issues that need to be encoded as clauses, with an outline of how to do this:

- To encode that the Turing machine accepts in time at most T, we need one single variable clause $\zeta_{T,a}$, representing that at the final time T the machine is in the acceptance state a.

- To encode that the first part of the input is x, we need a set of single variable clauses $\wedge_{1 \leq i \leq n} \xi_{i,1,x_i}$. This restriction stops before the rest of the tape, as that is where the second input y will come. We will produce similar clauses to enforce that y directly follows x and takes no more than n^ℓ space on the tape, and that after y, the tape is blank.

- To encode the start configuration of the Turing machine, we also need two single-variable clauses $\zeta_{1,s} \wedge \nu_{1,0}$, showing we start at the start state in the `0`th position of the tape.

- We need clauses that make sure that exactly one of the $\zeta_{t,q}$ variables is *True* at any time (i.e. the machine has only one state at a time). To express this, we need $\vee_q \zeta_{t,q}$ for all times t, saying that the head is in at least one state. Further, for all pairs $p \neq q$ we need $(\overline{\zeta}_{t,p} \vee \overline{\zeta}_{t,q})$. These clauses say that two states or two positions cannot be *True* at the same time step.

- Similarly, we need clauses that make sure at each time, and each tape position there is exactly one character written, as well as clauses that make sure that the head is in exactly one position at each time step. These get expressed using variables $\xi_{i,t,\sigma}$ and $\nu_{t,i}$

similarly to how we expressed that exactly one of the $\zeta_{t,q}$ variables is 1 at each time.

- Finally, we need to encode that this is a valid computation:
  - For any location where the head isn't pointed at time t (that is, $\nu_{t,i} = 0$), the character written there doesn't change, so $\nu_{t,i} = 0 \Rightarrow \xi_{i,t,\sigma} = \xi_{i,t+1,\sigma}$, which is expressed as the two clauses $(\nu_{t,i} \vee \xi_{i,t,\sigma} \vee \overline{\xi}_{i,t+1,\sigma}) \wedge (\nu_{t,i} \vee \overline{\xi}_{i,t,\sigma} \vee \xi_{i,t+1,\sigma})$.
  - For a location where the head is pointed at time t, the movement of the head, change of state, and symbol written should all match the δ function of the Turing machine. This requires a rather elaborate set of clauses to encode these, so we omit them here for brevity.

With the encoding claimed above, the Turing machine `CheckerTM` proves that an input x is in $\mathcal{L}$, if and only if the SAT formula so created is satisfiable, showing that $\mathcal{L} \leq \text{SAT}$. □

References

Michael George. Lecture 9: Countability. In *CS2800 Lecture Notes*. Cornell University, 2018. https://courses.cs.cornell.edu/cs2800/wiki/index.php/SP18:Lecture_9_Countability.

Robert D. Kleinberg. Notes on Turing machines. In *CS4820 Lecture Notes*. Cornell University, 2018. http://www.cs.cornell.edu/courses/cs4820/2018sp/handouts/turingm.pdf.

Dexter Kozen. *Automata and Computability*. Springer-Verlag, New York, 1997.

Andrew Myers. Hard and incomputable problems. In *CS2112 Lecture Notes*. Cornell University, 2015. http://www.cs.cornell.edu/courses/cs2112/2015fa/lectures/lecture.html?id=undecidability.

William J. Rapaport. The Boehm-Jacopini theorem and structured programming. In *CSE 111 Lecture Notes*. University of Buffalo Dept. of Computer Science and Engineering, 2004. https://cse.buffalo.edu/ rapaport/111F04/greatidea3.html.

Steven Zeil. Turing completeness. In *CS390 Lecture Notes*. Old Dominion University, 2016. https://www.cs.odu.edu/ zeil/cs390/latest/Public/turing-complete/index.html.