CS 4820: Limits of Computability

Eshan Chattopadhyay, Robert Kleinberg, 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.

Informal 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 specify any other limit on the running time; it could require exponentially many steps or arbitrarily large amounts of memory to run an algorithm that solves this problem, but if the problem 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 *Models of computation and the Church-Turing thesis*

The definition of computability given above is informal because we have not yet given formal mathematical meaning to terms such as *algorithm* and *computer*. This same issue, of course, affects everything we've said about algorithms up until this point in the course. However, it not an important issue when asserting the existence of a particular algorithm to solve a particular problem. The particular algorithm being analyzed is generally presented in a mathematically precise way, and the assertion that it is possible to run the algorithm on a computer is not presented as a mathematical assertion but as a factual claim that you can verify by coding up the algorithm yourself and running it.

On the other hand, when asserting the *non-existence* of algorithms to solve a particular problem (i.e. when asserting that a problem is uncomputable) the lack of a precise definition of algorithms and computation becomes a fatal flaw in the logic. If one is imprecise about the meanings of these terms, one might always wonder, "Is it really impossible to design an algorithm to solve this problem, or has mankind just failed to come up with the right hardware or software feature that makes the problem solvable?"

In the 1930's, mathematicians formalized the definitions of algorithms and computation in different ways. Perhaps the two most famous and influential such definitions are the *lambda calculus* the *Turing machine*, both proposed in the 1930's by Alonzo Church and Alan Turing, respectively. Lambda calculus is briefly discussed in CS 3110 and constitutes a core topic of CS 4110. Turing machines, which we'll encounter later in these notes, are basically discrete finite automata augmented with an infinite memory. These two models of computation, and all of the other universal models of computation proposed before or since, have been found to be *computationally equivalent*, meaning that any problem which is computable in one model is computable in all of them. These computationally equivalent models are collectively referred to as *Turing-complete*. Because Turing-complete models are capable of describing every computational process that computer scientists, mathematicians, philosophers, and natural scientists have ever been able to imagine, the scientific community generally subscribes to the *Church-Turing thesis*, asserting that any computation that can be feasibly performed in our universe can be performed using the lambda calculus, a Turing machine, or any other Turing-complete model of computation. The Church-Turing thesis is not a mathematical theorem, it is a *meta-mathematical* statement about the meaning of algorithms and computation. In that sense, when we say a problem is uncomputable, the real meaning of the statement is, "No algorithm can solve this problem in a universe where the capablities of algorithms and computers obey the Church-Turing thesis, and there is strong consensus in the scientific community that our universe is one such universe." The discovery of a computer that can violate the Church-Turing thesis would be a flabbergasting discovery, akin to the discovery of faster-than-light transportation, but it is not a possibility that can be ruled out mathematically.

Definition Two models of computation are *computationally equivalent* if they define the same set of computable problems. The *Turing-complete* models of computation are those that model the capabilities of general-purpose computers.

Turing-complete models of computation tend to have some key characteristics in common. These are:

1. Algorithms are described by finite-sized programs.

2. Computers execute algorithms using a finite amount of internal state and an infinite amount of memory.

3. The program describing one algorithm can be used as the input to another algorithm.

4. The programming language is sophisticated enough that it is capable of *universality*: one can write an algorithm that can simulate the execution of any other algorithm, given that algorithm's

program as input.

The assumption that algorithms run on computers with an infinite amount of memory is clearly not realistic. There are two justifications for making this assumption. First, modern computers have access to so much memory (especially if they are attached to a network) that for most practical purposes one doesn't have to worry about the limitations on how much data they can store. Second, since our main intent in this part of the course is to prove *limitations on the power of computation*, if we can prove that a problem is impossible to solve even on a hypothetical computer equipped with infinite memory, it implies *a fortiori* that the problem is also unsolvable on a real-world computer with finite memory.

You have already learned at least one programming language, e.g. Java, that is Turing-complete. Certain features of the Java language (classes, methods, inheritance, generic types, exception handling, importing packages, etc.) are convenient for the purpose of writing programs but inconvenient for the purpose of reasoning formally about their semantics. Therefore, in these notes, we will formalize the model of computation using two computationally equivalent models: Turing machines, and a very simple programming language we will define called SJava or Simplified Java, whose syntax resembles Java minus the object-oriented features of the language. Turing machines furnish an economical definition of computation with the minimum nnumber of conceptual ingredients; however, they are ill-suited for expressing algorithms in a human-readable form. SJava is designed to be comparable to Java code in terms of readability, with the benefit that its semantics are much easier to comprehend and formalize than the semantics of Java.

3 *Turing machines*

A Turing machine can be thought of as a finite state machine sitting on an infinitely long tape containing symbols from some finite alphabet Σ. Based on the symbol it's currently reading, and its current state, the Turing machine writes a new symbol in that location (possibly the same as the previous one), moves left or right or stays in place, and enters a new state. It may also decide to halt. The machine's *transition function* is the "program" that specifies each of these actions (next state, next symbol to write, and direction to move on the tape) given the current state and the symbol the machine is currently reading.

Actually, we will base our model of computation on a generalization of this idea, the *multi-tape Turing machine*, which has a finite number of infinite tapes (potentially more than one) each with its own read-write head that can move independently of the others. A single finite state controller jointly controls all of the read-write heads.

Specification of a Turing machine

Definition 1. A Turing machine is specified by:

1. a finite set Σ called the *alphabet*, with a distinguished subset Ω called the set of *input symbols* and a distinguished element of $\Sigma \setminus \Omega$ called the *blank symbol* and denoted by underscore (_);

2. a finite set of states Q with two distinguished elements: s (the starting state) and t (the terminal or halting state);

3. a finite set of tapes $[T] = \{1, 2, \ldots, T\}$ with two distinguished subsets I (the input tapes) and O (the output tapes);

4. a *transition function*

$$\delta : Q \times \Sigma^T \to Q \times \Sigma^T \times \{-1, 0, 1\}^T$$

   that specifies, for any given state and T-tuple of symbols, what the machine should do next: the state to which it transitions, the T-tuple of symbols that it writes on its tapes, and the T-tuple of directions that it moves on each tape.

Often, a Turing machine is defined as having a single tape that serves as both the input and the output tape (i.e., the case $T = 1$ and $I = O = [T]$). This definition is conceptually simpler and is computationally equivalent in the sense that any function computable by a multi-tape Turing machine is also computable by a single-tape Turing machine. However algorithms implemented on single-tape Turing machines often have asymptotically slower running times than the same algorithm implemented on modern computing hardware with random-access memory. Multi-tape Turing machines, which lacking an abstraction of random access memory, tend to permit implementations of algorithms whose running time has the same asymptotic order of growth as if they were implemented on a random-access machine. For this reason, we prefer the multi-tape formalism in CS 4820.

Configurations and computations

Having defined the *specification* of a Turing machine, we must now pin down a definition of *how they operate* and *what they compute*. This has been informally described above, but it's time to make it formal. That begins with formally defining the configuration of the Turing machine at any time (the contents of its tapes, as well as the machine's own state and its position on each tape) and the rules for how its configuration changes over time.

The set Σ^* is the set of all finite sequences of elements of Σ, and Σ_c^∞ is the set of all infinite sequences of elements of Σ that are *finitely supported*, meaning that all but finitely many elements of the sequence are equal to _. When an element of Σ^* is denoted by a letter such as x, then the elements of the sequence x are denoted by $x[0], x[1], x[2], \ldots, x[n-1]$, where n is the length of x and is denoted by $|x|$. Similarly for an infinite sequence $x \in \Sigma_c^\infty$, the elements of x are denoted by $x[0], x[1], \ldots$.

A *configuration* of a Turing machine is an ordered triple $(q, \boldsymbol{x}, \boldsymbol{k}) \in Q \times (\Sigma_c^\infty)^T \times \mathbb{N}^T$, where q denotes the machine's current state, $\boldsymbol{x} = (x_1, x_2, \ldots, x_T)$ denotes T-tuple of strings on the tapes, and $\boldsymbol{k} = (k_1, k_2, \ldots, k_T)$ denotes the T-tuple of positions of the machine on the tapes.

Suppose M is a Turing machine and $(q, \boldsymbol{x}, \boldsymbol{k})$ is its configuration at any point in time. If $q \neq t$ (the machine hasn't halted) then the configuration at the following point in time, $(q', \boldsymbol{x}', \boldsymbol{k}')$, is determined as follows. Let $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_T)$ denote the T-tuple of symbols that the machine is reading, i.e. $\sigma_i = x_i[k_i]$ for all $i \in [T]$. Let $(q', \boldsymbol{\rho}, \boldsymbol{\ell}) = \delta(q, \boldsymbol{\sigma})$. For all $i \in [T]$ the new string on tape i, x_i', is obtained from x_i by changing $x_i[k]$ to ρ_i. The new position $k'[i]$ is equal to $k[i] + \ell[i]$ unless $k[i] + \ell[i] = -1$, in which case $k'[i] = 0$. We say that M *transitions* from $(q, \boldsymbol{x}, \boldsymbol{k})$ to $(q', \boldsymbol{x}', \boldsymbol{k}')$.

A *computation* of a Turing machine is a sequence of configurations $(q^j, \boldsymbol{x}^j, \boldsymbol{k}^j)$ indexed by a sequence of consecutive time steps j starting from 0, that satisfies:

- The machine starts in a valid starting configuration, meaning that $q^0 = s$ and $\boldsymbol{k}^0 = (0, 0, \ldots, 0)$.
- Each pair of consecutive configurations represents a valid transition of M.
- If the sequence is infinite we say that the computation *does not halt*.
- If the sequence is finite and its length in m, we require that the final state q^{m-1} is equal to the halting state t, we say that the computation *halts*, and we refer to m as the *running time* of the computation.

Finally, having specified *how Turing machines compute*, let us now specify *what they compute*. First we must define two functions

$$\text{pad} : \Sigma^* \to \Sigma_c^\infty, \quad \text{unpad} : \Sigma_c^\infty \to \Sigma^*$$

as follows. If $y \in \Sigma^*$ then $\mathrm{pad}(y)$ is formed from y by appending an infinite sequence of blank symbols. If $x \in \Sigma_c^\infty$ then $\mathrm{unpad}(x)$ is the longest initial subsequence of x that contains no blank symbols. Note that if $y \in \Sigma^*$ contains no blank symbols then $\mathrm{unpad}(\mathrm{pad}(y))$ is equal to y but otherwise it is strictly shorter than y.

Suppose we are given an I-tuple of strings, $y = (y_i)_{i \in I}$. The *computation of M with input y* is the unique computation of M that starts with tape contents $x_i = \mathrm{pad}(y_i)$ for $i \in I$ and $x_i = \mathrm{pad}(\epsilon)$ for $i \notin I$. Here ϵ denotes the empty string, so $\mathrm{pad}(\epsilon)$ denotes a tape containing nothing but blank symbols.

If the computation of M with input y halts, we say that *M halts on y*. Letting $\boldsymbol{x} = (x_1, \ldots, x_T)$ denote the tape contents in the final configuration of the computation, and letting $z_i = \mathrm{unpad}(x_i)$ for each i, we call the O-tuple $z = (z_i)_{i \in O}$ the output of the Turing machine, and we write $M(y) = z$. If M does not halt on y we write $M(y) = \nearrow$.

A *partial function* between two sets A and B is a function $f : D \to B$ where D is a subset of A called the *domain* of f. If $f : (\Omega^*)^I \to (\Sigma^*)^O$ is a partial function from I-tuples of input strings to O-tuples of output strings, and if D is the domain of f, then we say that Turing machine M *computes* f if $M(y) = f(y)$ for all $y \in D$ and $M(y) = \nearrow$ for all $y \notin D$.

Example: computing the substring relation

To illustrate how Turing machines work, in this section we present a Turing machine that computes whether one string is a substring of another. Recall that x_1 is said to be a substring of x_2 if the symbols in x_1 form a contiguous subsequence of the symbols in x_2. We will design a Turing machine M with input alphabet $\Omega = \{0, 1\}$ such that $M(x_1, x_2) = 1$ if x_1 is a substring of x_2 and $M(x_1, x_2) = 0$ otherwise.

Our machine will have three tapes: two input tapes containing x_1 and x_2 respectively, and an output tape that is only used in the final transition of the computation to write a 1 or 0. The algorithm that we will use to test if x_1 is a substring of x_2 is the obvious one: for each starting position $p = 0, 1, \ldots, |x_2| - |x_1|$ we will test if a substring of x_2 starting at p matches x_1. Thus, there will be an outer loop that iterates over p, and an inner loop that iterates over the symbols in x_1. Of course, loops are not explicitly defined in the semantics of Turing machine computations, but we'll use the state transition rules to implement a loop. A more challenging problem concerns the fact that $|Q|$, the size of the state set, might be much smaller than $|x_1|$ and $|x_2|$, so the loop counters can't be stored in the Turing machine's internal state. Instead loop counters have to be stored, either explicitly or implicitly, in the form of information on the tapes or information about the positions of the tape heads. In particular, when an iteration of the inner loop ends without finding a match between x_1 and a contiguous subsequence of x_2, we need to return to the position in x_2 where began checking for a match. However, the Turing machine's internal state can't store enough information to locate that position. Instead we'll record the location by writing a blank symbol (_) on the input tape containing x_2. This will overwrite one of the symbols of x_2, but it won't matter because the overwritten symbol cannot belong to a substring that matches x_1.

In more detail, the Turing machine has states s, c, r, t with the following meanings and functionalities.

- s is the *starting* state. This state is used not only at the start of the entire computation but at the start of each iteration of the outer loop. In state s we check whether the first symbol of x_1 matches the earliest symbol of x_2 that hasn't yet been tested as a potential beginning of substring that matches x_1. We also overwrite this symbol of x_2 with _ to mark the position on the tape where this iteration of the outer loop began.

- c is the *comparing* state. This state checks whether a symbol of x_1 matches the corresponding symbol of x_2.

- r is the *returning* state. It is used after an iteration of the outer loop fails to find a match, to return to the position where we began the outer loop iteration.

- t is the *terminating* state. It is used to halt the Turing machine after writing the output.

The transition function is represented in Table 1. For brevity, we have omitted the lines of the table that specify the value of $\delta(q, \sigma_1, \sigma_2, \sigma_3)$ when the symbol on the output tape, σ_3, is not the blank symbol. That is because those transitions are irrelevant: the machine we are designing never writes a non-blank symbol on the output tape until the moment that it terminates. For the same reason, we have omitted the lines of the table that specify the value of $\delta(t, \sigma_1, \sigma_2, \sigma_3)$ since there cannot be transitions out of the terminal state t.

4 SJava: *a simplified Java-like syntax for expressing algorithms*

It's probably clear from the example in Table 1 that Turing machine transition functions are actually a terrible way of presenting algorithms to human beings. For the purpose of reasoning about algorithms and computability in these notes, it will be convenient to have a syntax for expressing algorithms which is human-readable,

Table 1: Turing machine to compute the substring relation

				$\delta(q,\sigma_1,\sigma_2,\sigma_3)$							
q	σ_1	σ_2	σ_3	q'	ρ_1	ρ_2	ρ_3	ℓ_1	ℓ_2	ℓ_3	comment
s	_	0	_	t	_	0	1	0	0	0	x_1 is the empty string
s	_	1	_	t	_	1	1	0	0	0	
s	_	_	_	t	_	_	1	0	0	0	
s	0	0	_	c	0	_	_	+1	+1	0	symbols match; keep checking
s	1	1	_	c	1	_	_	+1	+1	0	
s	0	1	_	r	0	_	_	0	0	0	mismatch
s	1	0	_	r	1	_	_	0	0	0	
s	0	_	_	t	0	_	0	0	0	0	reached end of x_2; x_1 can't be a substring
s	1	_	_	t	1	_	0	0	0	0	
c	_	0	_	t	_	0	1	0	0	0	reached end of x_1; match found!
c	_	1	_	t	_	1	1	0	0	0	
c	_	_	_	t	_	_	1	0	0	0	
c	0	0	_	c	0	0	_	+1	+1	0	symbols match; keep checking
c	1	1	_	c	1	1	_	+1	+1	0	
c	0	1	_	r	0	1	_	0	0	0	mismatch
c	1	0	_	r	1	0	_	0	0	0	
c	0	_	_	t	0	_	0	0	0	0	reached end of x_2; x_1 can't be a substring
c	1	_	_	t	1	_	0	0	0	0	
r	_	0	_	t	_	0	0	0	0	0	this line irrelevant: σ_1 is never _in state r
r	_	1	_	t	_	1	0	0	0	0	
r	_	_	_	t	_	_	0	0	0	0	
r	0	0	_	r	0	0	_	−1	−1	0	keep moving left; don't overwrite symbols
r	0	1	_	r	0	1	_	−1	−1	0	
r	1	0	_	r	1	0	_	−1	−1	0	
r	1	1	_	r	1	1	_	−1	−1	0	
r	0	_	_	s	0	_	_	0	+1	0	returned to _on tape 2; start again at next symbol
r	1	_	_	s	1	_	_	0	+1	0	

like pseudocode. On the other hand, since our aim is to reason in a logically rigorous way about the limits of computation, rather than using pseudocode containing natural-language phrases whose meaning is subject to interpretation, it will be convenient when possible to express algorithms in a language with precisely-defined semanics.

Since you are all familiar with Java, we have chosen in these notes to express algorithms in a language with Java-like syntax, but omitting superfluous object-oriented features of Java that are irrelevant for present purposes. We keep only the bare essentials of an imperative programming language: variables, arrays, functions, assignment statements, control flow. We call this language Simplified Java, or SJava.

By way of illustration, we have rewritten the program for computing whether x_1 is a substring or x_2 in SJava in Figure 1.

```
boolean substring(string x1, string x2) {
  pos2 = 0;
  while (pos2 < length(x2)) {
    if (testMatch(x1, x2, pos2)) {
      return true;
    }
    pos2 = pos2 + 1;
  }
  return false;
}

boolean testMatch(string s1, string s2, int offset) {
                              // Does s1 match s2 starting from offset2?
  if (length(s1) > (length(s2) - offset)) {
    return false;             // s1 is too long
  }
  pos1 = 0;
  pos2 = offset;
  while (pos1 < length(s1)) { // main loop: look for mismatches
    if (s1[pos1] != s2[pos2]) {
      return false;
    }
    pos1 = pos1 + 1;
    pos2 = pos2 + 1;
  }                           // loop completed; no mismatch found
  return true;
}
```

Figure 1: SJava program to compute the substring relation.

Data types

Each expression in a SJava program has a value in one of the following data types.

- **boolean**: a value that is either **true** or **false**.
- **int**: a value in $\mathbb{Z} \cup \{\perp\}$, where $\perp$ is a special "not a number" value used to handle cases such as division by zero.
- **char**: a character from a pre-specified set Σ. Specifying the character set Σ is part of specifying the model of computation. For concreteness, we take Σ to be the ASCII character set.
- **string**: a finite sequence of characters.
- **array**: a finite sequence of integers.

There is no limitation on the number of characters in a string or the number of elements in an array. Note that SJava only allows arrays of integers. A string is, in effect, the same thing as an array of characters. But one cannot create an array of Booleans, strings, or arrays, nor can one create an array whose contents constitute a mixture of integers and characters.

Each data type has a *default value*: **false** for Booleans, 0 for integers, _ for characters, ϵ for strings, and $\emptyset$ for arrays.

Program structure

A SJava program is organized as a sequence of function definitions. The first function in the sequence (called the base function) is interpreted as the "main" function (even if it is not called "main") and the value it returns is interpreted as the program's output.

The body of a function is enclosed in curly braces and consists of a sequence of statements. Each statement is either an assignment, a control-flow statement (if, else, while, or return), or a curly brace denoting the end of a block of code. These are discussed further below. Programs may contain comments, for the sake of readability. If a line of code contains two consecutive slashes ("//") then any characters from the double-slash until the next new-line are ignored.

Variables

Variables are named by strings that may not contain whitespace and must start with a letter of the English alphabet. The scope of a variable is local to the function in which it is used;. The first time a variable is used in a function must either be in the function declaration (where the variable is declared to be one of the function's arguments) or on the left side of an assignment statement. In the latter case the variable's type is inferred from the expression on the right side of the assignment. Variables are automatically initialized with the default value for their data type.

Expressions

An expression is a segment of code that can be evaluated to yield a value. It must be one of the following.

1. a variable name, such as n
2. a constant, such as 2
3. an array element or string element, such as x[n]
4. a function applied to a tuple of expressions, such as testMatch("Hello",str,n+m)
5. a binary operation applied to two expressions, such as testMatch("Hello",str,0) || testMatch("World",str,0)
6. a string or array element, such as str[n+1]

Assignment statements

Assignment statements are in one of two forms.

```
x = expr;
x[n] = expr;
```

In both cases x is a variable and expr is an expression (which may contain the variables x and n as sub-expressions). Execution of the assignment statement begins by evaluating the expression. The resulting value then replaces the value of x in the first case, or x[n] in the second case, which requires x to be a string or array. If n is less than zero, then x is unchanged. If n+1 exceeds the length of x, then after the expression on the right side is evaluated, x is padded with copies of the relevant default value (i.e., _ for the characters of a string, 0 for the integers in an array) until its length equals n+1, and then the value of the expression on the right side is substituted for x[n].

Control flow

The semantics of if, else, and while statements are exactly as in Java. The conditional expression in an if statement or while loop must be an expression of Boolean type, enclosed in parentheses.

A return statement must be followed by an expression whose type matches the return-value type of the function in which it appears.

When executing a return statement, the expression is first evaluated and then the value is substituted in the expression which called the function. (An exception is the base function which was called when the program first started running; its return value is treated as the program's output.)

If the execution of a function reaches the last line of the function's body without executing a return command, the function returns the default value for its return-value type.

Small-step semantics of SJava

We have explained the syntax of the SJava language and we have written informally about its semantics, i.e. what it means to execute a SJava program. Towards describing how to write a SJava interpreter, we must now specify its *small-step semantics*. That is, we will specify how the *execution state* is represented as a program executes, and how the state is updated as steps of the program are executed.

Function state: the state of a function as it executes is an ordered pair (c, ϕ) where:

- c is the *program counter*, a natural number denoting the position of the next instruction to be executed. The value of c is the absolute position of the first character of this instruction in the SJava program to which the function belongs.
- ϕ is a dictionary mapping all the variable names and constant expressions that occur in the function to their current values.

Execution state: Since functions in a SJava program may call other functions, the overall state of a program can be conceptualized as a stack of function states, where the top of the stack is the state of the function currently being executed, and the other elements of the stack are the states of the functions whose execution is paused pending a return value from the function above them in the stack. Mathematically, we will represent this stack as a finite sequence of ordered pairs $\{(c_i, \phi_i)\}_{i=0}^{h}$, where the last element of the sequence, (c_h, ϕ_h), represents the top element in the stack. "Popping the stack" refers to modifying this sequence by deleting its last element, whereas "pushing an element onto the stack" refers to modifying the sequence by appending one element at the end.

Initial state of a function: Every function f has a well-defined initial state $(c_{\text{init}}(f), \phi_{\text{init}}(f))$. The initial program counter position, $c_{\text{init}}(f)$, is the location of the first non-whitespace character after the curly brace that starts the function body. The initial dictionary, $\phi_{\text{init}}(f)$, assigns to each variable the default value of its associated type.

Initial execution state of a program: The initial execution state of a SJAVA program is a one-element stack consisting of the function state $(c_{\text{init}}(b), \phi_{\text{init}}(b))$, where b is the base function of the program.

Evaluating an expression: The rules for evaluating an expression in a given execution state vary according to the format of the expression.

1. If the expression is a term its value is read from the dictionary phi_h.
2. If it is a string or array element its value is determined by the rules specified above under "Operations on strings and arrays".
3. If the expression applies a binary operation or built-in function (such as length$(\cdot)$) to one or two terms, the values of those terms are extracted from the dictionary and the expression's value is determined as explained above when we described SJAVA's binary operations and built-in functions.
4. If the expression involves application of a function f, the execution state is changed by pushing the initial state of f onto the stack.

The first three cases don't change the execution state but they do yield a value; we will call this *immediate evaluation*. The remaining case, which we will call *pending evaluation*, changes the program state by pushing a new function state onto the stack.

Updating the execution state: SJAVA programs are deterministic, meaning that for every execution state there is a uniquely defined subsequent state. The subsequent state depends on the current state $\{(c_i, \phi_i)\}_{i=0}^{h}$ and on the instructions contained in the line of code that starts at position c_h in the program, i.e. the line of code pointed to by the program counter of the function state at the top of the stack. After discarding comments from the end of the line, there are six types of lines of code, and each of them changes the execution state in a different way. In the following discussion, "program counter" and "dictionary" refer to c_h and ϕ_h, the program counter and dictionary of the function state residing at the top of the stack before executing the line of code.

1. **Blank lines:** A line consisting entirely of whitespace is executed by advancing the program counter to the next line (i.e., the character immediately following the $\hookleftarrow$ at the end of the line).
2. **Assignment statements:** The right side of an assignment statement is an expression. To execute the assignment statement, the first step is to evaluate the expression on the right side. An

immediate evaluation yields a value, and the execution state is modified as follows. The dictionary is modified by substituting that value for the value of the variable appearing on the left side of the assignment statement. The program counter is advanced to the next line. A pending evaluation doesn't change ϕ_h or c_h but increases the stack depth.

3. **if statements:** The conditional expression is evaluated. In the case of an immediate evaluation, the program counter is updated to the first (non-whitespace) character of the "then block" or the "else block", depending if the value is **true** or **false**. A pending evaluation doesn't change ϕ_h or c_h but increases the stack depth.

4. **while statements:** The conditional expression is evaluated. In the case of an immediate evaluation yielding **true**, the program counter is updated to the first (non-whitespace) character inside the body of the while loop. In the case of an immediate evaluation yielding **false**, the program counter is updated to the first (non-whitespace) character following the body of the while loop. A pending evaluation doesn't change ϕ_h or c_h but increases the stack depth.

5. **return statements:** The expression following the keyword return is evaluated. A pending evaluation doesn't change ϕ_h or c_h but increases the stack depth. An immediate evaluation yielding value v is more interesting. If $h = 0$ then the stack contains only one function state, namely the base function. Execution terminates and the program outputs v. If $h > 0$ then the stack is popped. The function state (c_{h-1}, ϕ_{h-1}) is now at the top of the stack, and the program counter c_{h-1} is at the start of a line of code whose execution produced a pending expression evaluation. The execution state is now updated as if the expression in that line of code had evaluated immediately to v.

6. **End of a block:** A right curly brace represents the end of a function body, a while loop, or one of the two blocks of an if statement. Upon reaching the end of a function body, the execution state changes as if it had reached a return statement whose expression immediately evaluated to the default value of the function's return type. Reaching the end of a while loop moves the program counter back to the beginning of the line containing the corresponding while statement. Reaching the end of one of the blocks of an if statement moves the program counter to the first non-whitespace character following the conclusion of all blocks of the if statement.

We conclude this section with a remark about running times. It is tempting to model the execution of one line of a SJava program as taking $O(1)$ time. Unfortunately that model doesn't accurately reflect the amount of time it takes to run a program on an actual computer. The issue is that the integer, string, and array data types in SJava store an unbounded number of bits. Operations performed on these data types in a single line of code (such as multiplying two integers, or copying the value of a string to another variable) would take an amount of time that depends on the number of bits required to represent the operands.

To illustrate the issue concretely, consider the following SJava program.

```
int doubleExponential(int n) {
  x = 3;
  while (n > 0) {
    x = x * x;
    n = n − 1;
  }
  return x;
}
```

This program performs n loop iterations, each of which performs only 2 arithmetic operations. Yet it returns the integer 3^{2^n}; merely writing the return value in binary requires $2^{O(n)}$ digits. So the running time of this innocuous-looking program is actually exponential in n.

The standard model of computation assumes that when one executes a program on an input of size n bits, operations performed on blocks of bits of size $O(\log n)$ take constant time. For operations that manipulate objects larger than $O(\log n)$ in a single line of code, the algorithm designer is responsible for thinking about how to implement the operation as a sequence of constant-time steps each of which reads and writes at most $O(\log n)$ bits of data. This assumption that the number of bits that can be manipulated in one time step scales logarithmically with the size of the program's input appears strange at first, but it turns out to be roughly consistent with the history of how datasets and computer architecture have co-evolved. Computers with 16-bit architectures used to be commonplace, they were eventually supplanted by 32-bit computers, which in turn were supplanted by 64-bit computers. Meanwhile, with each doubling of the number of bits, the size of the datasets encountered in typical computing workloads roughly squared.

5 *The universal* SJava *program*

Recall that a key characteristic of Turing-complete models of computation is *universality*, the ability for one single algorithm to simulate the execution of any other algorithm, given a program describing that algorithm as input. In this section we will illustrate that SJava has this property. In other words, we will show that it is possible to write a SJava interpreter in SJava. The code for such an interpreter is called a *universal* SJava *program*. We won't actually present the source code of a universal SJava program in these notes — that would be quite tedious — but we'll explain the key ingredients that need to be assembled in order to write one.

A SJava interpreter should simulate the execution of any SJava program running on any valid input. The difficulty of using one program to simulate every possible program becomes apparent as soon as we ask ourselves the question, "What should be the return value type of the SJava interpreter's base function?" The interpreter needs to simulate a function whose return value could be any of bool, int, char, string, or array. A similar question arises when we consider how the *interpreter's* input should represent the input to the program it is simulating. A SJava program's base function can have any finite number of arguments; this tuple of arguments constitutes the input to the program. The interpreter, on the other hand, must have a pre-specified number of arguments and they must be of pre-specified types. How is a program with, say, two string arguments supposed to simulate a program with five integer arguments?

Encoding data as strings

The key to getting around this issue is to *represent everything as a string*. Values of variables and expressions, the tuple of values that constitutes the input to a SJava program, the value that the program outputs, its function states, and its execution state — all of these will be represented as strings by the interpreter. We now specify how the string representation of each of these elements is defined.

Boolean values **true** is represented as the four-character string "true", **false** is represented as the five-character string "false".

Integers An integer is represented in base 10, as a string of digits potentially preceded by the '-' character in the case of negative numbers. The integer $\perp$ is represented using the underscore character.

Characters A character is represented using a single quote, followed by one character, followed by another single quote.

Strings A string is represented using a double quote, followed by a sequence of characters each preceded by a backslash, followed by a double quote.

Arrays An array is represented using a left square bracket, followed by a comma-separated list of strings representing integers, followed by a right square bracket.

Program input The input to a SJava program is a tuple of values. This is represented using a left parenthesis, followed by a comma-separated list of strings each representing a value, followed by a right parenthesis.

Program output The program outputs a value of type bool, int, char, string, or array. The rules for formatting each of these types as a string have already been explained.

Function state A function state is represented by a string in the format $(c, \{\text{name}_1 = \text{val}_1, \text{name}_2 = \text{val}_2, \ldots, \text{name}_m = \text{val}_m\})$. Here, c is a string representing the integer value of the program counter. Each name_i $(i = 1, \ldots, m)$ is the name of a variable and val_i is a string representing the variable's value.

Execution state The execution state of a program is a sequence of function states. To represent this sequence as a string, we simply concatenate the strings representing each of the function states in the sequence. The execution state of a program that has terminated is represented by the string that represents the value of the program's output.

Simulating execution of the program

Since we defined program execution in terms of small-step semantics, it makes sense for the interpreter's code to have the overall structure of the skeleton code in Figure 2.

6 *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[1] SJava program M for a decision problem with input x. There are three possible outcomes:

[1] If M itself does not compile, or does not return a Boolean value, we treat it as rejecting all input.

1. *accept*: the program terminates and returns **true**, denoting "yes",

2. *reject*: the program terminates and returns **false**, denoting "no", or

```
string interpreter(string program, string input) {
  execState = initialExecState(program, input);
  done = false;
  while (not(done)) {
    execState = singleStep(program, execState);
    done = testIfHalted(execState);
  }

  # Program has halted, so execState is
  # a string representing its return value.

  return execState;
}

boolean testIfHalted(string execState) {
  firstChar = execState[0];
  if (firstChar == '(') then {
    return false;
  }
  else {
    return true;
  }
}

string singleStep(string program, string execState) {
  # code for single-step semantics goes here
}

string initialExecState(string program, string input) {
  # code for setting up initial execution state goes here
}
```

Figure 2: SJava interpreter skeleton code.

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 $\mathcal{L}_M$ as the collection of all x accepted by M, i.e., such that $M(x)$ terminates and returns **true**.[2]We can think of $\mathcal{L}_M$ 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 $\mathcal{L}_M$, 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 $\mathcal{L}_M$.

[2] The set $\mathcal{L}_M$ 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 $\mathcal{L}_M$—meaning outcome 1 is reached by M for input x if and only if $x \in \mathcal{L}_M$—we say that M *recognizes* $\mathcal{L}_M$, or that $\mathcal{L}_M$ 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* $\mathcal{L}_M$ if M not only returns **true** for all x in $\mathcal{L}_M$, but also returns **false** in finite time for all x not in $\mathcal{L}_M$. In other words, M must make the correct *decision* about whether x is in $\mathcal{L}_M$ 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.

7 *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 $\mathcal{L}_{\text{HALT}}$:

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.

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

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.

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 $(1, 8)$ [3] 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 $digit(r, i) \neq digit(real(i), i)$.

[3] 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 $digit(real(1), 1)$.

number	digits	1	2	3	4	5	6	
real(1)	0.	<u>2</u>	1	2	5	6	7	...
real(2)	12.	6	<u>3</u>	3	3	6	7	...
real(3)	1.	5	0	<u>0</u>	0	0	0	...
real(4)	-5.	9	8	1	<u>4</u>	4	8	...
...								
r	0.	3	4	1	5			...

Table 2: 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*.

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 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.

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.

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:*

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

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` with the following specifications:

- **Arguments:** `haltChecker` takes in two arguments: `program`, the source code for a SJava program, and `input`, the input that should be fed to the SJava 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 its source code in Figure 3.

The base function of `diagonalizer` takes in only one string as an argument: the code of a program `program`. The method uses `haltChecker` to check if `program` halts when fed *its own source code* as input.[4] If `haltChecker(program, program)` returns **true**, meaning the program would halt, then we will make the `diagonalizer` program go into an infinite loop by calling a function `runForever` that recursively calls itself ad infinitum. However, if `haltChecker(program,program)` returns **false**, meaning the program would not halt, then we will tell our `diagonalizer` program to immediately return **true** and halt.

[4] 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.

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(`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. □

```
boolean diagonalizer(string program) {
  if (haltChecker(program,program)) {
    return runForever();
  }
  else {
    return true;
  }
}

boolean runForever() {
  return runForever();
}

boolean haltChecker(string program, string input) {
  #
  # body of haltChecker goes here
  #
}

#
# additional functions used by haltChecker, if any, go here
#
```

Figure 3: Skeleton code for `diagonalizer`. Everything beginning with the line `boolean haltChecker(string program, string input)` is the source code of `haltChecker` itself.

We can dig more deeply into where a contradiction arises (assuming existence of `haltChecker`) by looking at what must happen in execution: if `outsidehaltChecker` is identical to `haltChecker` and `string diagonalizerCode` is the code for `diagonalizer`, we could define the boolean `out` as:

```
boolean out = outsidehaltChecker(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` 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` 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.

8 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:*

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

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. See Figure 5 for source code of M'. `haltChecker` then passes the source code of M' and x to the `acceptanceChecker`. Figure 4 shows source code of `haltChecker`.

- 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 M never halted on its input, as otherwise M' would have returned **true** and thus accepted x.

These two bullet points establish that the `haltChecker` program in Figure 4 decides $\mathcal{L}_{\text{HALT}}$, assuming that `acceptanceChecker` decides $\mathcal{L}_{\text{ACCEPT}}$. In other words, $\mathcal{L}_{\text{HALT}} \leq \mathcal{L}_{\text{ACCEPT}}$. Since there is no algorithm to decide the halting problem, there can be no algorithm to decide the acceptance problem. □

Remark 1. In the above proof, we required modifying the source code of M to get a program M' which has a new function as its main function. It is important to ensure that the name of this function is distinct from the function names appearing in M. Since given the string that corresponds to M, we can extract the function names appearing in M, one easy way of ensuring the new function has a distinct name is by just concatenating the function names appearing in M, and using this as the main function of M'.

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`.

```
boolean haltChecker(string M, string x) {
        M'=modify(M);
        return acceptanceChecker(M',x);
   }

  string modify(string M) {
        # modify string M to string M' such that
        # M' corresponds to the code in Figure 5.
  }

boolean acceptanceChecker(string program, string input) {
   #
   # body of acceptanceChecker goes here
   #
}

#
# additional functions used by acceptanceChecker, if any, go here
#
```

Figure 4: Skeleton code for `haltChecker`. Everything beginning with the line `boolean acceptanceChecker(string M, string x)` is the source code of `acceptanceChecker` itself.

```
boolean distinct(string w) {
        a = main(w); #assuming main is the first function of M
        return true;
}

#
# code for M goes here
#
```

Figure 5: Skeleton code for `M'` where `distinct` is a function name that does not appear in the code of M.

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

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

This problem is undecidable.

Proof. We will suppose there exists a program `zeroChecker` that takes in the source code for a program and returns **true** if that program accepts the input `0`, otherwise **false**. We are going to use this program to create a halt checker as follows: `haltChecker` on being given as input the code of a program M and input x does the following:

- writes the code of a program M' that on input w has the following behavior: if $w = \mathtt{0}$, it uses the universal SJava program to simulate M on x. If M halts and produces an output, M' ignores it and outputs **true**. If $w \neq \mathtt{0}$, it outputs **false** (and halts).
- supplies the code of M' as input to `zeroChecker`, and returns the output of `zeroChecker`.

The source code of `haltChecker` is supplied in Figure 6.

To prove correctness of the `haltChecker`, suppose that M halts on x. Then clearly M' on input `0` outputs **true**, and thus `zeroChecker` outputs **true** as well when supplied with the code of M' as input.

Now suppose M does not halt on x. Then, M' on input `0` does not halt as well and thus M' does not accept `0` in this case. Hence `zeroChecker` outputs **false** on being given the code of M' as input. This concludes the proof that `haltChecker` indeed decides $\mathcal{L}_{\text{HALT}}$ correctly, assuming `zeroChecker` decides $\mathcal{L}_0$ correctly. Since the halting problem is undecidable, it must be the case that `zeroChecker` does not decide $\mathcal{L}_0$ correctly, i.e. $\mathcal{L}_0$ is undecidable. □

9 *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

```
boolean haltChecker(string M, string x) {
        M'= zerofocus(M,x);
        return zeroChecker(M');
}

 string zerofocus(string M, string x) {
           # return a string that corresponds to the code of the program
           # given in Figure 7.
 }

 boolean zeroChecker(string program) {
     #
     # body of zeroChecker goes here
     #
 }

  #
  # additional functions used by zeroChecker, if any, go here
  #
```

Figure 6: Skeleton code for `haltChecker`.

```
 boolean distinct(string w) {
           if (w == ''0'') {
                     a = interpreter(M,x);
                     return true;
           }
           return false;
 }

 #
 # code for interpreter (see Figure 2) goes here
 #
```

Figure 7: Skeleton code for `M'` where `distinct` is a function name that does not appear in the code of M.

`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 $\mathcal{L}_1$ that is nontrivial, i.e. for which there is at least one program A that is in $\mathcal{L}_1$ and at least one program A' that is not in $\mathcal{L}_1$. More specifically, we will assume the program A' not in $\mathcal{L}_1$ is the program that loops infinitely for all inputs, therefore not accepting any input. We can assume this because, if this program was in $\mathcal{L}_1$, we could simply prove the undecidability of the complement of $\mathcal{L}_1$ instead, using the method we will describe. We can do this because the complement of an undecidable language is also undecidable.[5]

[5] 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(`M`)` will return **true** if M is in $\mathcal{L}_1$, and **false** if M is not in $\mathcal{L}_1$ 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`(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 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 and A'. After constructing the source code for M', the `HaltChecker` will feed M' to `propertyChecker` and output the result. See Figure 8 for a skeleton code of M'.

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

- If M halts on input x, then M' will respond exactly like A for all inputs. By definition, this means the program is in $\mathcal{L}_{\mathrm{I}}$. `propertyChecker`(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', the program that halts on no inputs, which is not in $\mathcal{L}_{\mathrm{I}}$. `propertyChecker`(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. $\square$

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'.

```
boolean haltChecker(string M, string x) {
    M'= propertyfocus(M,x);
    return propertyChecker(M');
}

string propertyfocus(string M, string x) {
    # return a string that corresponds to the code of the program
    # given in Figure 9.
}

boolean propertyChecker(string program) {
    #
    # body of propertyChecker goes here
    #
}

#
# additional functions used by propertyChecker, if any, go here
#
```

Figure 8: Skeleton code for `haltChecker`.

```
boolean distinct(string y) {

        a = interpreter(M,x);
        return main(y); #assuming main is the first
                        #function of program A

}

#
# code for interpreter (see Figure 2) goes here
#

#
# code for A goes here
#
```

Figure 9: Skeleton code for `M'` where `distinct` is a function name that does not appear in the code of M.

10 Proving a problem is not recognizable

We have seen how to prove a problem $\mathcal{L}$ is undecidable by reducing $\mathcal{L}_{\text{HALT}}$ to $\mathcal{L}$. In this section we will see how to prove a problem is not even recognizable, by reducing from the complement of the halting problem. The reason this works is that the complement of the halting problem is not recognizable. More concretely, there is no algorithm that, given as an input a pair $\langle M, x \rangle$, accepts the input if and only if M never halts on input x.

Theorem 5 (The co-halting problem is not recognizable.). *Let $\mathcal{L}_{\text{CO-HALT}}$ denote the complement of the halting problem, i.e. the set of all pairs $\langle M, x \rangle$ such that M never halts on input x. The language $\mathcal{L}_{\text{CO-HALT}}$ is not recognizable.*

Proof. Consider any program `antiHaltChecker` that computes a Boolean function of two strings. We will prove that `antiHaltChecker` fails to recognize $\mathcal{L}_{\text{CO-HALT}}$, by building another program called `diagonalizer2` using the code for `antiHaltChecker`. The construction is shown in Figure 10 and is very similar to the diagonalizer developed in Figure 3 when we proved that the halting problem is undecidable.

```
boolean diagonalizer2(string program) {
  if (antiHaltChecker(program,program)) {
    return true;
  }
  else {
    return runForever();
  }
}

boolean runForever() {
  return runForever();
}

boolean antiHaltChecker(string program, string input) {
  #
  # body of haltChecker goes here
  #
}

#
# additional functions used by antiHaltChecker, if any, go here
#
```

Figure 10: Skeleton code for `diagonalizer2`. Everything beginning with the line `boolean antiHaltChecker(string program, string input)` is the source code of `antiHaltChecker` itself.

Let `progDiag2` denote a string constituting the code for the `diagonalizer2` program. Consider what happens when one calls the function `antiHaltChecker(progDiag2,progDiag2)`.

There are two cases.

1. If `antiHaltChecker(progDiag2,progDiag2)` returns **true**, then according to the code for the `diagonalizer2` program, it will return **true** on input string `progDiag2`. In particular, this means that the program encoded by the string `progDiag2` halts on input `progDiag2`, i.e. the pair $\langle \texttt{progDiag2}, \texttt{progDiag2} \rangle$ does not belong to $\mathcal{L}_{\text{CO-HALT}}$. Thus, `antiHaltChecker` fails to recognize $\mathcal{L}_{\text{CO-HALT}}$ because it accepts the pair $\langle \texttt{progDiag2}, \texttt{progDiag2} \rangle$.

2. If `antiHaltChecker(progDiag2,progDiag2)` returns **false**, or if it runs forever without terminating, then according to the code for the `diagonalizer2` program, it will run forever on input string `progDiag2`. In particular, this means that the program encoded by the string `progDiag2` does not halt on input `progDiag2`, i.e. the pair $\langle \texttt{progDiag2}, \texttt{progDiag2} \rangle$ belongs to $\mathcal{L}_{\text{CO-HALT}}$. Thus, `antiHaltChecker` fails to recognize $\mathcal{L}_{\text{CO-HALT}}$ because it does not accept the pair $\langle \texttt{progDiag2}, \texttt{progDiag2} \rangle$.

In both cases `antiHaltChecker` fails to recognize $\mathcal{L}_{\text{CO-HALT}}$. Since `antiHaltChecker` was an arbitrary program that computes a Boolean function of two strings, this confirms that there is no program that recognizes $\mathcal{L}_{\text{CO-HALT}}$. $\square$

To illustrate the process of proving that another language is not recognizable, by reducing from $\mathcal{L}_{\text{CO-HALT}}$, we present the following theorem.

Theorem 6. *Let*

$\mathcal{L}_{\text{INF}} = \{\langle M \rangle \mid M$ *is a* SJAVA *program that accepts infinitely many distinct inputs*$\}$.

The language $\mathcal{L}_{\text{INF}}$ is not recognizable.

Proof. As promised, we will prove this theorem by showing that $\mathcal{L}_{\text{CO-HALT}} \leq \mathcal{L}_{\text{INF}}$. In other words, given an SJAVA program that accepts the language $\mathcal{L}_{\text{INF}}$ we will produce an SJAVA program that accepts the language $\mathcal{L}_{\text{CO-HALT}}$. Equivalently, we assume we are given a SJAVA program that, on input M', halts and outputs **true** if and only if M' is a SJAVA program that accepts infinitely many distinct inputs. We want to use this as a subroutine in a program that accepts some other pair $\langle M, x \rangle$ if and only if M does not halt on input x. Hence, the essential question we must ask ourselves is: *how can the fact that one program accepts infinitely many different inputs constitute evidence that another program never halts when it runs on a specified input?* As one ponders this question an elegant answer begins to take shape: for a given pair $\langle M, x \rangle$ suppose that M' is a program which accepts its input string y if and only if M runs for more than length(y) steps when processing input x. If the execution of M on input x terminates after a finite number of steps, t, then M' accepts only strings of length less than t, so the number of distinct strings that M' accepts is finite. If M never halts on input x then M' accepts every string, so the number of distinct strings that M' accepts is infinite. We have shown that $M' \in \mathcal{L}_{\text{INF}}$ if and only if $\langle M, x\rangle \in \mathcal{L}_{\text{CO-HALT}}$, indicating that the transformation from $\langle M, x\rangle$ to M' is indeed a correct reduction from recognizing $\mathcal{L}_{\text{CO-HALT}}$ to recognizing $\mathcal{L}_{\text{INF}}$. The only thing that remains is to prove that the reduction itself — i.e., the function mapping the pair $\langle M, x\rangle$ to the program M' — is a computable function. To substantiate this, we use the code in Figure 12. The code makes use of four functions. One of these is called `infiniteChecker` is assumed to be a program that recognizes $\mathcal{L}_{\text{INF}}$. The other three perform string-processing tasks that are, in principle, easy to implement in SJava so we omit the code for these string processing routines.

1. `substitute` takes three strings x, y, z and transforms x by replacing the first occurrence of substring y in x (if any) with string z.
2. `universalSJavaProgram` is a function that takes no arguments and outputs a gigantic string constituting the universal SJava program.
3. `bigString` is a function that takes no arguments and outputs a string which is equal to the piece of code in Figure 11.

□

```
boolean compareLengthToRunningTime(string y) {
  prog = ''SUBST_PROG'';
  execState = initialExecState (prog,SUBST_INPUT);
  countdown = length(y);
  done = (countdown == 0);
  while (!done) {
    execState = singleStep(prog,execState);
    countdown = countdown − 1;
    done = (testIfHalted(execState)) || (countdown == 0);
  }
  return !testIfHalted(execState);
}

SUBST_USJP
```

Figure 11: String to be output by `bigString()` function.

Figure 12: Skeleton code for reduction from $\mathcal{L}_{\text{CO-HALT}}$ to $\mathcal{L}_{\text{INF}}$.

```
boolean antiHaltChecker(string program, string input) {
  universalProg = universalSJavaProgram();
  tempString1 = bigString();
  tempString2 = substitute(tempString1,''SUBST_PROG'',program);
  tempString3 = substitute(tempString2,''SUBST_INPUT'',input);
  Mprime = substitute(tempString3,''SUBST_USJP'',universalProg);
  return infiniteChecker(Mprime);
}

string substitute(string template, string macro, string replacement) {
  #
  # body of substitute goes here.
  #
}

string universalSJavaProgram() {
  # simply return the code of the universal SJava program.
}

string bigString() {
  # simply return the big string specifying the template of the reduction.
}

boolean infiniteChecker(string prog) {
  #
  # body of infiniteChecker goes here.
  #
}

#
# additional functions used by infiniteChecker go here.
#
```

11 Cook-Levin Theorem

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 7 (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.[6]

[6] 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 single tape Turing Machine, `CheckerTM`. Indeed this is possible by application of the Church-Turing hypothesis (discussed in Section 2). 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 $\xi_{i,t,\sigma} = \mathit{True}$ 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 $\mathcal{L} \leq_p$ SAT. $\square$

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.