بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۰۱)

نظریهٔ زبان‌ها و ماشین‌ها

حسین فلسفین

Undecidability

In this session, we prove one of the most philosophically important theorems of the theory of computation: There is a specific problem that is algorithmically unsolvable. Computers appear to be so powerful that you may believe that all problems will eventually yield to them. The theorem presented here demonstrates that computers are limited in a fundamental way. What sorts of problems are unsolvable by computer? Are they esoteric, dwelling only in the minds of theoreticians? No! Even some ordinary problems that people want to solve turn out to be computationally unsolvable.

مثالی از یک مسئلهٔ حل‌ناپذیر توسط کامپیوتر

In one type of unsolvable problem, you are given a computer program and a precise specification of what that program is supposed to do (e.g., sort a list of numbers). You need to verify that the program performs as specified (i.e., that it is correct). Because both the program and the specification are mathematically precise objects, you hope to automate the process of verification by feeding these objects into a suitably programmed computer. However, you will be disappointed. The general problem of software verification is not solvable by computer.

اولین مسئله‌ای که حل‌ناپذیری‌اش را اثبات خواهیم کرد: A_{TM}

We aim to help you develop a feeling for the types of problems that are unsolvable and to learn techniques for proving unsolvability.

Now we turn to our first theorem that establishes the undecidability of a specific language: the problem of determining whether a Turing machine accepts a given input string. We call it A_{TM} by analogy with A_{DFA} and A_{CFG}. But, whereas A_{DFA} and A_{CFG} are decidable, A_{TM} is not. Let

$$A_{TM} = \{\langle M, w\rangle | M \text{ is a TM and } M \text{ accepts } w\}.$$

Theorem 1: A_{TM} is undecidable.

Before we get to the proof, let's first observe that A_{TM} is Turing-recognizable. Thus, this theorem shows that recognizers are more powerful than deciders. Requiring a TM to halt on all inputs restricts the kinds of languages that it can recognize. The following Turing machine U recognizes A_{TM}.

U = "On input $\langle M, w \rangle$, where M is a TM and w is a string:
1. Simulate M on input w.
2. If M ever enters its accept state, accept; if M ever enters its reject state, reject."

Note that this machine loops on input $\langle M, w \rangle$ if M loops on w, which is why this machine does not decide A_{TM}. If the algorithm had some way to determine that M was not halting on w, it could reject in this case. However, an algorithm has no way to make this determination, as we shall see.

The Diagonalization Method

The proof of the undecidability of A_{TM} uses a technique called diagonalization, discovered by mathematician Georg Cantor in 1873. Cantor was concerned with the problem of measuring the sizes of infinite sets. If we have two infinite sets, how can we tell whether one is larger than the other or whether they are of the same size? For finite sets, of course, answering these questions is easy. We simply count the elements in a finite set, and the resulting number is its size. But if we try to count the elements of an infinite set, we will never finish! So we can't use the counting method to determine the relative sizes of infinite sets. For example, take the set of even integers and the set of all strings over $\{0, 1\}$. Both sets are infinite and thus larger than any finite set, but is one of the two larger than the other? How can we compare their relative size?

Cantor proposed a rather nice solution to this problem. He observed that two finite sets have the same size if the elements of one set can be paired with the elements of the other set. This method compares the sizes without resorting to counting. We can extend this idea to infinite sets. Here it is more precisely.

Definition: *Assume that we have sets A and B and a function f from A to B. Say that f is one-to-one if it never maps two different elements to the same place—that is, if $f(a) \neq f(b)$ whenever $a \neq b$. Say that f is onto if it hits every element of B—that is, if for every $b \in B$ there is an $a \in A$ such that $f(a) = b$. Say that A and B are the same size if there is a one-to-one, onto function $f : A \mapsto B$. A function that is both one-to-one and onto is called a correspondence. In a correspondence, every element of A maps to a unique element of B and each element of B has a unique element of A mapping to it. A correspondence is simply a way of pairing the elements of A with the elements of B.*

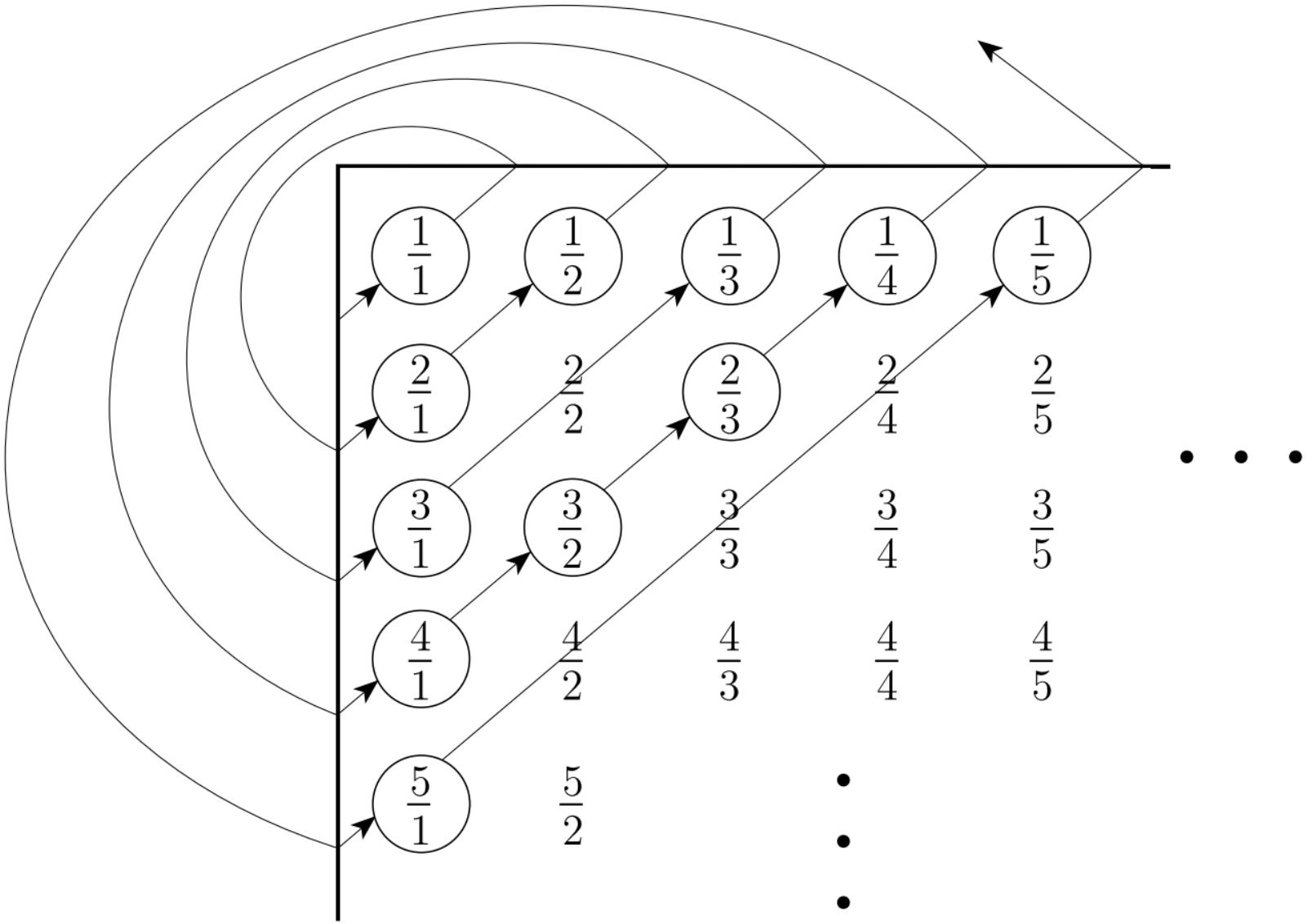
Alternative common terminology for these types of functions is injective for one-to-one, surjective for onto, and bijective for one-to-one and onto.

Example: Let $\mathbb{N}$ be the set of natural numbers $\{1, 2, 3, \ldots\}$ and let $\mathbb{E}$ be the set of even natural numbers $\{2, 4, 6, \ldots\}$. Using Cantor's definition of size, we can see that $\mathbb{N}$ and $\mathbb{E}$ have the same size. The correspondence f mapping $\mathbb{N}$ to $\mathbb{E}$ is simply $f(n) = 2n$. Of course, this example seems bizarre. Intuitively, $\mathbb{E}$ seems smaller than $\mathbb{N}$ because $\mathbb{E}$ is a proper subset of $\mathbb{N}$. But pairing each member of $\mathbb{N}$ with its own member of $\mathbb{E}$ is possible, so we declare these two sets to be the same size.

Definition: *A set A is **countable** if either it is finite or it has the same size as $\mathbb{N}$.*

Example: *Now we turn to an even stranger example. If we let $\mathbb{Q}^+ = \{\frac{m}{n} | m, n \in \mathbb{N}\}$ be the set of positive rational numbers, $\mathbb{Q}^+$ seems to be much larger than $\mathbb{N}$. **Yet these two sets are the same size according to our definition.** We give a correspondence with $\mathbb{N}$ to show that $\mathbb{Q}^+$ **is countable**. One easy way to do so is to list all the elements of $\mathbb{Q}^+$. Then we pair the first element on the list with the number 1 from $\mathbb{N}$, the second element on the list with the number 2 from $\mathbb{N}$, and so on. We must ensure that every member of $\mathbb{Q}^+$ appears only once on the list. To get this list, we make an infinite matrix containing all the positive rational numbers, as shown in the following figure. The ith row contains all numbers with numerator i and the jth column has all numbers with denominator j. So the number $\frac{i}{j}$ occurs in the ith row and jth column.*

A correspondence of $\mathbb{N}$ and $\mathbb{Q}^+$



Now we turn this matrix into a list.

☞ *One (bad) way to attempt it would be to begin the list with all the elements in the first row. That isn't a good approach because the first row is infinite, so the list would never get to the second row.*

Instead we list the elements on the diagonals, which are superimposed on the diagram, starting from the corner. The first diagonal contains the single element $\frac{1}{1}$, and the second diagonal contains the two elements $\frac{2}{1}$ and $\frac{1}{2}$. So the first three elements on the list are $\frac{1}{1}$, $\frac{2}{1}$, and $\frac{1}{2}$. In the third diagonal, a complication arises. It contains $\frac{3}{1}$, $\frac{2}{2}$, and $\frac{1}{3}$. If we simply added these to the list, we would repeat $\frac{1}{1} = \frac{2}{2}$. We avoid doing so by skipping an element when it would cause a repetition. So we add only the two new elements $\frac{3}{1}$ and $\frac{1}{3}$. Continuing in this way, we obtain a list of all the elements of $\mathbb{Q}^+$.

The Set of Turing Machines Is Countable

To show that the set of all Turing machines is countable, we first observe that the set of all strings Σ^* is countable for any alphabet Σ. (We defined an alphabet to be any nonempty finite set.) With only finitely many strings of each length, we may form a list of Σ^* by writing down all strings of length 0, length 1, length 2, and so on. The set of all Turing machines is countable because each Turing machine M has an encoding into a string $\langle M \rangle$. If we simply omit those strings that are not legal encodings of Turing machines, we can obtain a list of all Turing machines.

بیان دیگر برای اثبات شمارا بودن مجموعهٔ دربردارندهٔ همهٔ ماشین‌های تورینگ

Let $\mathcal{T}$ represent the set of Turing machines. A TM T can be represented by the string $e(T) \in \{0,1\}^*$, and a string can represent at most one TM. Therefore, the resulting function e from $\mathcal{T}$ to $\{0,1\}^*$ is one-to-one, and we may think of it as a bijection from $\mathcal{T}$ to a subset of $\{0,1\}^*$. Because $\{0,1\}^*$ is countable, every subset is, and we can conclude that $\mathcal{T}$ is countable. Saying that the set of Turing machines with input alphabet Σ is countable is approximately the same as saying that the set $\mathcal{RE}(\Sigma)$ of recursively enumerable languages over Σ is countable. A language $L \in \mathcal{RE}(\Sigma)$ can be accepted by a TM T with input alphabet, and a TM can accept only one language over its input alphabet. Therefore, since T is countable, the same argument we have just used shows that $\mathcal{RE}(\Sigma)$ is countable.

An Undecidable Language

Now we are ready to prove Theorem 1, the undecidability of the language $A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$.

Proof: We assume that A_{TM} is decidable and obtain a contradiction. Suppose that H is a decider for A_{TM}. On input $\langle M, w\rangle$, where M is a TM and w is a string, H halts and accepts if M accepts w. Furthermore, H halts and rejects if M fails to accept w. In other words, we assume that H is a TM, where

$$H(\langle M, w\rangle) = \begin{cases} \text{accept}, & \text{if } M \text{ accepts } w, \\ \text{reject}, & \text{if } M \text{ does not accept } w. \end{cases}$$

Now we construct a new Turing machine D with H as a subroutine. This new TM calls H to determine what M does when the input to M is its own description $\langle M \rangle$. Once D has determined this information, it does the opposite. That is, it rejects if M accepts and accepts if M does not accept. The following is a description of D.

$D =$ *"On input $\langle M \rangle$, where M is a TM:*
1. Run H on input $\langle M, \langle M \rangle \rangle$.
2. Output the opposite of what H outputs. That is, if H accepts, reject; and if H rejects, accept."

In summary,

$$D(\langle M \rangle) = \begin{cases} \textit{accept}, & \textit{if } M \textit{ does not accept } \langle M \rangle, \\ \textit{reject}, & \textit{if } M \textit{ accepts } \langle M \rangle. \end{cases}$$

What happens when we run D with its own description $\langle D \rangle$ as input? In that case, we get

$$D(\langle D \rangle) = \begin{cases} \textit{accept}, & \textit{if } M \textit{ does not accept } \langle D \rangle, \\ \textit{reject}, & \textit{if } M \textit{ accepts } \langle D \rangle. \end{cases}$$

No matter what D does, it is forced to do the opposite, which is obviously a contradiction. Thus, neither TM D nor TM H can exist.

Let's review the steps of this proof.

Assume that a TM H decides A_{TM}. Use H to build a TM D that takes an input $\langle M \rangle$, where D accepts its input $\langle M \rangle$ exactly when M does not accept its input $\langle M \rangle$. Finally, run D on itself. Thus, the machines take the following actions, with the last line being the contradiction.

* ***H accepts $\langle M, w \rangle$ exactly when M accepts w.***
* ***D rejects $\langle M \rangle$ exactly when M accepts $\langle M \rangle$.***
* ***D rejects $\langle D \rangle$ exactly when D accepts $\langle D \rangle$.***

Where is the diagonalization in the proof of Theorem 1?

It becomes apparent when you examine tables of behavior for TMs H ***and*** D***. In these tables we list all TMs down the rows,*** $M_1, M_2, \ldots$***, and all their descriptions across the columns,*** $\langle M_1 \rangle, \langle M_2 \rangle, \ldots$***. The entries tell whether the machine in a given row accepts the input in a given column. The entry is accept if the machine accepts the input but is blank if it rejects or loops on that input. We made up the entries in the following figure to illustrate the idea.***

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*		*accept*		
M_2	*accept*	*accept*	*accept*	*accept*	
M_3					$\cdots$
M_4	*accept*	*accept*			
$\vdots$			$\vdots$		

Entry i, j ***is accept if*** M_i ***accepts*** $\langle M_j \rangle$

In the following figure, the entries are the results of running H on inputs corresponding to the above figure. So, if M_3 does not accept input $\langle M_2 \rangle$, the entry for row M_3 and column $\langle M_2 \rangle$ is reject because H rejects input $\langle M_3, \langle M_2 \rangle \rangle$.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*	
M_4	*accept*	*accept*	*reject*	*reject*	
$\vdots$			$\vdots$		

Entry i, j is the value of H on input $\langle M_i, \langle M_j \rangle \rangle$

In the following figure, we added D to the above figure. By our assumption, H is a TM and so is D. Therefore, it must occur on the list $M_1, M_2, \ldots$ of all TMs. Note that D computes the opposite of the diagonal entries. The contradiction occurs at the point of the question mark where the entry must be the opposite of itself.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*		*reject*	
M_4	*accept*	*accept*	*reject*	*reject*		*accept*	
$\vdots$		$\vdots$			$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		?	
$\vdots$		$\vdots$					$\ddots$

If D is in the figure, a contradiction occurs at "?"

Reducibility

Here, we examine several additional unsolvable problems. In doing so, we introduce the primary method for proving that problems are computationally unsolvable. It is called reducibility. A reduction is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem.

Reducibility always involves two problems, which we call A and B. If A reduces to B, we can use a solution to B to solve A. Note that reducibility says nothing about solving A or B alone, but only about the solvability of A in the presence of a solution to B.

دربارهٔ اهمیت حیاتی مفهوم کاهش‌پذیری برای اثبات حل‌ناپذیری مسائل

Reducibility plays an important role in classifying problems by decidability, and later in complexity theory as well. When A is reducible to B, solving A cannot be harder than solving B because a solution to B gives a solution to A. In terms of computability theory, if A is reducible to B and B is decidable, A also is decidable. Equivalently, if A is undecidable and reducible to B, B is undecidable. This last version is key to proving that various problems are undecidable. In short, our method for proving that a problem is undecidable will be to show that some other problem already known to be undecidable reduces to it.

We have already established the undecidability of A_{TM}, the problem of determining whether a Turing machine accepts a given input. Let's consider a related problem, $HALT_{TM}$, the problem of determining whether a Turing machine halts (by accepting or rejecting) on a given input. This problem is widely known as the halting problem. We use the undecidability of A_{TM} to prove the undecidability of the halting problem by reducing A_{TM} to $HALT_{TM}$.

$$HALT_{TM} = \{\langle M, w\rangle | M \text{ is a TM and } M \text{ halts on input } w\}.$$

This is something that occurs several times in theoretical computer science—the first problem is difficult to prove, and after that, one can use the first problem and the idea of reducibility to prove other problems are in the same category.

Theorem: $HALT_{TM}$ *is undecidable.*

Proof Idea: This proof is by contradiction. We assume that $HALT_{TM}$ *is decidable and use that assumption to show that* A_{TM} *is decidable, contradicting Theorem 1. The key idea is to show that* A_{TM} *is reducible to* $HALT_{TM}$*. Let's assume that we have a TM* R *that decides* $HALT_{TM}$*. Then we use* R *to construct* S*, a TM that decides* A_{TM}*. To get a feel for the way to construct* S*, pretend that you are* S*. Your task is to decide* A_{TM}*. You are given an input of the form* $\langle M, w \rangle$*. You must output accept if* M *accepts* w*, and you must output reject if* M *loops or rejects on* w*. Try simulating* M *on* w*. If it accepts or rejects, do the same. But you may not be able to determine whether* M *is looping, and in that case your simulation will not terminate. That's bad because you are a decider and thus never permitted to loop. So this idea by itself does not work. Instead, use the assumption that you have TM* R *that decides* $HALT_{TM}$*. With* R*, you can test whether* M *halts on* w*. If* R *indicates that* M *doesn't*

halt on w, reject because $\langle M, w \rangle$ isn't in A_{TM}. However, if R indicates that M does halt on w, you can do the simulation without any danger of looping. Thus, if TM R exists, we can decide A_{TM}, but we know that A_{TM} is undecidable. By virtue of this contradiction, we can conclude that R does not exist. Therefore, $HALT_{TM}$ is undecidable.

Proof: Let's assume for the purpose of obtaining a contradiction that TM R decides $HALT_{TM}$. We construct TM S to decide A_{TM}, with S operating as follows.

S = *"On input $\langle M, w \rangle$, an encoding of a TM M and a string w:*

1. *Run TM R on input $\langle M, w \rangle$.*
2. *If R rejects, reject.*
3. *If R accepts, simulate M on w until it halts.*
4. *If M has accepted, accept; if M has rejected, reject."*

Clearly, if R decides $HALT_{TM}$, then S decides A_{TM}. Because A_{TM} is undecidable, $HALT_{TM}$ also must be undecidable.

The above theorem, illustrates our strategy for proving that a problem is undecidable. This strategy is common to most proofs of undecidability, except for the undecidability of A_{TM} itself, which is proved directly via the diagonalization method.

We now present several other theorems and their proofs as further examples of the reducibility method for proving undecidability. Let

$$E_{TM} = \{\langle M \rangle | M \text{ is a TM and } L(M) = \varnothing\}.$$

Theorem: E_{TM} *is undecidable.*

Proof Idea: We follow the pattern adopted in the previous theorem. We assume that E_{TM} *is decidable and then show that* A_{TM} *is decidable—a contradiction. Let* R *be a TM that decides* E_{TM}*. We use* R *to construct TM* S *that decides* A_{TM}*. How will* S *work when it receives input* $\langle M, w \rangle$*? One idea is for* S *to run* R *on input* $\langle M \rangle$ *and see whether it accepts. If it does, we know that* $L(M)$ *is empty and therefore that* M *does not accept* w*. But if* R *rejects* $\langle M \rangle$*, all we know is that* $L(M)$ *is not empty and therefore that* M *accepts some string—but we still do not know whether* M *accepts the particular string* w*. So we need to use a different idea. Instead of running* R *on* $\langle M \rangle$*, we run* R *on a modification of* $\langle M \rangle$*.*

We modify $\langle M \rangle$ to guarantee that M rejects all strings except w, but on input w it works as usual. Then we use R to determine whether the modified machine recognizes the empty language. The only string the machine can now accept is w, so its language will be nonempty iff it accepts w. If R accepts when it is fed a description of the modified machine, we know that the modified machine doesn't accept anything and that M doesn't accept w.

Proof: Let's write the modified machine described in the proof idea using our standard notation. We call it M_1.

M_1 = *"On input x:*

1. *If $x \neq w$, reject.*
2. *If $x = w$, run M on input w and accept if M does."*

This machine has the string w as part of its description. It conducts the test of whether $x = w$ in the obvious way, by scanning the

input and comparing it character by character with w to determine whether they are the same. Putting all this together, we assume that TM R decides E_{TM} and construct TM S that decides A_{TM} as follows.

S = "On input $\langle M, w \rangle$, an encoding of a TM M and a string w:
1. Use the description of M and w to construct the TM M_1 just described.
2. Run R on input $\langle M_1 \rangle$.
3. If R accepts, reject; if R rejects, accept."

Note that S must actually be able to compute a description of M_1 from a description of M and w. It is able to do so because it only needs to add extra states to M that perform the $x = w$ test. If R were a decider for E_{TM}, S would be a decider for A_{TM}. A decider for A_{TM} cannot exist, so we know that E_{TM} must be undecidable.

So far, our strategy for proving languages undecidable involves a reduction from A_{TM}. Sometimes reducing from some other undecidable language, such as E_{TM}, is more convenient when we are showing that certain languages are undecidable.

The following theorem shows that testing the equivalence of two Turing machines is an undecidable problem. We could prove it by a reduction from A_{TM}, but we use this opportunity to give an example of an undecidability proof by reduction from E_{TM}. Let

$$EQ_{TM} = \{\langle M_1, M_2\rangle | M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}.$$

Theorem: EQ_{TM} is undecidable.

Proof Idea: Show that if EQ_{TM} were decidable, E_{TM} also would be decidable by giving a reduction from E_{TM} to EQ_{TM}. The idea is simple. E_{TM} is the problem of determining whether the language of a TM is empty. EQ_{TM} is the problem of determining whether the languages of two TMs are the same. If one of these languages happens

to be $\varnothing$, we end up with the problem of determining whether the language of the other machine is empty—that is, the E_{TM} problem. So in a sense, the E_{TM} problem is a special case of the EQ_{TM} problem wherein one of the machines is fixed to recognize the empty language. This idea makes giving the reduction easy.

Proof: We let TM R decide EQ_{TM} and construct TM S to decide E_{TM} as follows.

S = "On input $\langle M \rangle$, where M is a TM:

1. Run R on input $\langle M, M_1 \rangle$, where M_1 is a TM that rejects all inputs.

2. If R accepts, accept; if R rejects, reject."

If R decides EQ_{TM}, S decides E_{TM}. But E_{TM} is undecidable, so EQ_{TM} also must be undecidable.

تمرین: اثبات کنید که مسئلهٔ زیر تصمیم‌ناپذیر است.

The Problem View: ***Given two TMs*** M_1 ***and*** M_2***, is*** $L(M_1) \subseteq L(M_2)$***?***

The Language View:

$\textbf{SUBSET}_{TM} = \{\langle M_1, M_2\rangle | M_1$ ***and*** M_2 ***are TMs and*** $L(M_1) \subseteq L(M_2)\}$.