Chapter 8

Undecidability

In this chapter we explore the well-known "halting problem," why it cannot be solved by any program or machine, and then introduce a powerful technique for demonstrating the unsolvability of other problems, or equivalently, the undecidability of other languages. The halting problem is an example of a *decision problem*, which is a collection of questions (or *problem instances*), each with a "yes" or "no" answer. In the case of the halting problem, the questions are of the form "Does machine M halt on input w." Each of these questions can be represented by a string, over some fixed alphabet, that encodes both M and w. In fact, the questions of every decision problem can be encoded as strings over some fixed alphabet; consequently *every decision problem is equivalent to a language recognition problem.* If we define the language L as consisting of just those (encodings of) questions that have "yes" answers, then deciding L is the same thing as solving the decision problem. This connection is so immediate and fundamental that we shall frequently speak of languages as *being* decision problems, and vise versa.

We also discuss a powerful theorem, known as Rice's theorem, for proving the undecidability of a large class of languages, and introduce *Post's correspondence problem*, which leads to a very simple-looking yet provably undecidable language.

8.1 The Halting Problem

Every program (or abstract machine) is a string over some alphabet; that is, it consists of a finite number of symbols drawn from some alphabet. For any alphabet Σ it is possible to *enumerate* all strings in the countable set Σ^* by listing them in lexicographic order. Consequently, the set of all possible programs is countable and can be enumerated. (For simplicity we will consider "nonsense" strings to be syntactically incorrect *programs*, so an enumeration of all strings is also an enumeration of all programs, both syntactically correct and syntactically incorrect.)

Imagine a table, infinite in two directions, that is formed by listing all strings over Σ across the top and down the left-hand side. We'll think of the strings along the side as representing programs (in some fixed but arbitrary language) or encodings of Turing machines, and the strings along the top as input strings to the programs/machines. We'll refer to the programs as P_0, P_1, $P_2, \ldots$, and the input strings as x_0, x_1, x_2, $\ldots$, even though the two lists of strings are actually identical. Each entry of the table describes what happens when a particular "program" is run on a particular "input," even though most of the strings will consist of syntactically incorrect programs or malformed input.

	x_0	x_1	x_2	x_3	x_4	$\ldots$
P_0	$\downarrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\downarrow$	$\ldots$
P_1	$\uparrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\uparrow$	$\ldots$
P_2	$\downarrow$	$\uparrow$	$\uparrow$	$\downarrow$	$\uparrow$	$\ldots$
P_3	$\downarrow$	$\uparrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\ldots$
P_4	$\uparrow$	$\downarrow$	$\uparrow$	$\uparrow$	$\downarrow$	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Here the (i, j) entry of the table is labeled "$\uparrow$" if P_i is a syntactically correct program that would loop forever on input x_j, and labeled "$\downarrow$" if P_i is syntactically incorrect, or the program would eventually halt, abort, or fail in any way on input x_j. That is, the (i, j) entry is "$\downarrow$" if program P_i would *not* loop forever on input x_j. According to this definition, every entry in the table is uniquely defined since every program either loops forever or it does not, regardless of whether or not we can actually make that determination in practice.[1] Let $H(P, x)$ be the predicate that returns $\top$ (true) if running program P on input x will not loop forever (due to syntax errors, correct termination, or aborting), and returns $\perp$ (false) if the program would go into an infinite loop. Using diagonalization, we will now show that *no program or machine can implement the predicate H.*

Let us begin by assuming that the predicate H *can* be implemented by some program; using this program we shall construct another program Q that is not listed anywhere along the left-hand side of the table, which results in a contradiction. We accomplish this using the standard diagonalization trick: we make the program Q disagree with (i.e. behave differently than) each of the programs $P_1, P_2, P_3, \ldots$ on *some* input, from which it follows that program Q differs from each of them. We simply define Q in such a way that

$$Q(x_i) = \begin{cases} \uparrow & \text{if } H(P_i, x_i) = \top \\ \downarrow & \text{if } H(P_i, x_i) = \perp, \end{cases}$$

where $\uparrow$ in this context means that the program goes into an infinite loop, and $\downarrow$ means that the program terminates. Program Q is trivial to write in any programming language that has conditional branching and looping constructs, assuming that we have access to H encapsulated as a subroutine. By construction, program Q exhibits different behavior than P_i when run on input string x_i. Therefore, $Q \neq P_i$ for all $i \in \mathbb{N}$, so Q cannot be in the list. Yet the list contains *all* programs! This contradiction forces us to conclude that the program implementing H cannot exist.

Note that in the definition of Q it is necessary to obtain program P_i based on the input x_i; but because of the way be set up our table, $P_i = x_i$, so we are actually evaluating $H(x_i, x_i)$. Had we insisted on only enumerating *syntactically correct* programs along the left-hand side of the table (which is

[1]We are making an important distinction here between the *truth* of the matter, and what we can actually say about the machine. That is, although we may not be able to actually determine whether or not a machine loops forever, clearly either it does or it does not. The table above should be thought of as containing the actual *truth* of the matter; hence, while the contents of such a table are not all available to us, we may nonetheless assert that such a table exists mathematically. Fortunately, all we need for the proof is its mere existence.

also possible), we would need to recover P_i from x_i by first determining the index i, then generating the i'th program in the enumeration. This is another valid but slightly more complicated way to construct the argument, but the conclusion remains the same; that is, H cannot be implemented by any program or machine.

Since the halting problem is a decision problem, it can be recast as a language recognition problem. The resulting language is an important one, so we will introduce a notation for it. We define

$$L_{\mathrm{H}} \overset{\mathrm{def}}{=} \{\langle M, u\rangle \mid \text{Turing machine } M \text{ eventually halts on input } w\}\,.$$

We refer to this language as the *halting language.* Here $\langle x\rangle$ simply means an *encoding* of x, whatever x may be, using some fixed (non-trivial) alphabet, Σ. Thus, $\langle M, w\rangle$ means a string that encodes a Turing machine (presumably as a 6-tuple) followed by an input string for the machine. Any Turing machine can be encoded using any alphabet, so long as the alphabet has at least two distinct symbols. We can think of replacing each symbol in the original (possibly larger) alphabet using a "code," or sequence of symbols in the fixed (possibly smaller) alphabet. Such encodings are commonplace. For example, each of the characters in the ASCII character set has a corresponding numerical value, which in turn can be encoded in binary using the alphabet $\{0, 1\}$.

In terms of the language L_{H}, the "Halting Problem" can be stated thus: L_{H} is undecidable. While L_{H} is recursively enumerable, the diagonalization argument given earlier shows that no program or machine can *decide* it. The language L_{H} is important for two reasons: First, it encodes a fundamentally unsolvable problem with practical implications, and secondly, because it can be used to demonstrate the undecidability of other languages. In the following section we introduce a powerful technique for establishing undecidability by *reducing* one language to another. We will also use the following theorem to show that some languages are not recursively enumerable.

Theorem 36 $\overline{L_{\mathrm{H}}}$ *is not recursively enumerable.*

Proof: Suppose that $\overline{L_{\mathrm{H}}}$ were r.e. Then since L_{H} is r.e. theorem 34 would imply that L_{H} is decidable, which is a contradiction. Therefore, $\overline{L_{\mathrm{H}}}$ cannot be r.e. $\square$

8.2 Reducing one Decision Problem to Another

We now describe a powerful technique for using the undecidability of L_{H} to prove that other languages are undecidable, and for using the fact that $\overline{L_{\mathrm{H}}}$ is not r.e. to prove that other languages are not r.e. We will use essentially the same technique later to demonstrate the *intractability* of languages. We shall define several transitive binary relations on the class of languages that will enable us to confer certain properties from one language to another; in particular, the properties of decidability, recursive enumerability, and the negations of these. These are listed in Figure 8.2. Of these, we will focus first on the *many-to-one reducibility* relation, "$\leq_m$," as this is the most commonly used, and will suffice for nearly all reductions that we will consider.

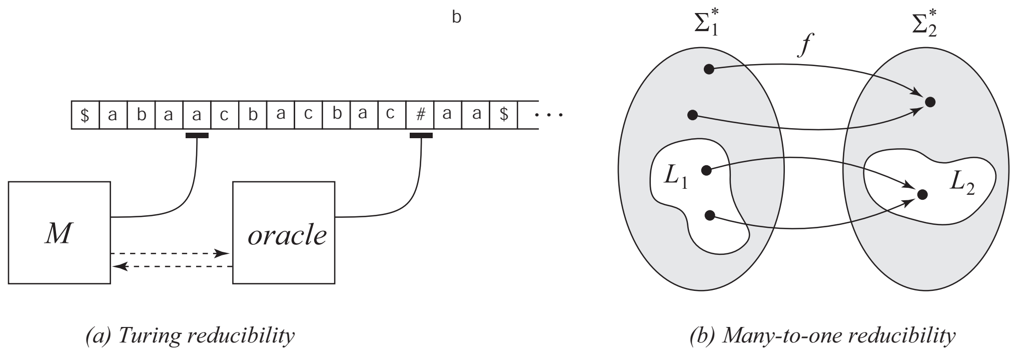

Figure 8.1: *Two fundamentally different notions of reducibility: (a) Turing reducibility assumes that one or more machines are available as oracles to another machine. (b) Many-to-one reducibility maps strings in one language, L_1, to strings in another, L_2. Strings that are outside of L_1 are mapped to strings that are outside of L_2.*

Symbol	Meaning	Generality
$\leq_T$	Turing reducible	most general, but rarely used
$\leq_m$	many-to-one reducible	quite general, and most used
$\leq_1$	one-to-one reducible	least general, rarely used

Figure 8.2: *Three closely related binary relations on languages. All are transitive, and all can be used to confer properties such as decidability and undecidability from one language to another. The many-to-one relation is used almost to the exclusion of the others.*

Definition 26 Let L_1 and L_2 be languages over the alphabets Σ_1 and Σ_2, respectively. We define the relation $L_1 \leq_m L_2$ to mean that there exists an algorithm for computing a function $f : \Sigma_1^* \to \Sigma_2^*$, where $w \in L_1$ if and only if $f(w) \in L_2$.

If $L_1 \leq_m L_2$ holds, we say that L_1 is *many-to-one reducible* to L_2. Intuitively, this means that L_2 is *at least as difficult* to decide as L_1, which makes the "$\leq$" notation very appropriate. Moreover, the relation $\leq_m$ shares some of the basic properties of the $\leq$ relation on the natural numbers. For example, it is easy to verify that $\leq_m$ is both reflexive and transitive. That is,

- Reflexive: $L \leq_m L$ for all languages L
- Transitive: $L_1 \leq_m L_2 \wedge L_2 \leq_m L_3 \implies L_1 \leq_m L_3$

The importance of this relation is made evident by the following theorem.

Theorem 37 *Let L_1 and L_2 be languages such that $L_1 \leq_m L_2$. Then if L_2 is r.e., so is L_1, and if L_2 is decidable, so is L_1.*

Proof: If there is an algorithm for f, where $w \in L_1$ if and only if $f(w) \in L_2$, and L_2 is decidable, then we can decide whether a string w is in L_1 by deciding whether $f(w)$ is in L_2. More precisely, we can construct a Turing machine that decides L_1 by combining two Turing machines; one that computes f and one that decides L_2. A similar argument holds when L_2 is merely Turing-acceptable. $\square$

Theorem 37 provides a powerful tool for proving that certain languages are undecidable. In particular, if $L_1 \leq_m L_2$ and L_1 is *undecidable*, then L_2 must also be *undecidable*. This implication is simply the contrapositive of the original implication. That is, the statements

- $D(L_2) \implies D(L_1)$
- $\neg D(L_1) \implies \neg D(L_2)$

are equivalent for any predicate D. Of course, in this context we take D to be the predicate such that $D(L)$ is true if and only if L is decidable. (Here D is an abstract mathematical object; we are assuming nothing about its computability.) We can therefore restate theorem 37 as follows.

Theorem 38 *Let L_1 and L_2 be languages such that $L_1 \leq_m L_2$. Then if L_1 is undecidable, so is L_2. Furthermore, if L_1 is not r.e., then neither is L_2.*

We now have a general strategy for showing that a language L_2 over alphabet Σ_2^* is undecidable (or not r.e.) without constructing a new diagonalization argument for each such language. There are three steps:

- Select another language L_1 over Σ_1^* that is already known to be undecidable (or not r.e).
- Show that there exists a mapping $f : \Sigma_1^* \to \Sigma_2^*$ such that $w \in L_1$ if and only if $f(w) \in L_2$.
- Show that f can be computed by some Turing machine.

We think of the function f as converting *problem instances of* L_1 into *problem instances of* L_2. The function f need not be either injective or surjective. In practice, however, the function f is frequently injective, mapping problem instances of L_1 to very similar problem instances of L_2, thereby letting the "decider" for L_2 do almost all the work.

We now demonstrate how to apply the reduction technique described in the previous section. We will show that the language

$$L_\varepsilon \overset{\text{def}}{=} \{u \in \Sigma^* \mid u = \langle M \rangle \text{ where Turing machine } M \text{ eventually halts on input } \varepsilon\}$$

is also undecidable. Note that the diagonalization argument that we used to prove the undecidability of L_{H} does not immediately apply in this case, since the table is only one-dimensional. However, if we can show that

$$L_{\text{H}} \leq_m L_\varepsilon,$$

then the undecidability of L_ε will be established, since we already know that L_{H} is undecidable. To show this, we need only prove that there exists a Turing computable function $f : \Sigma^* \rightarrow \Sigma^*$ such that $w \in L_{\text{H}}$ if and only if $f(w) \in L_\varepsilon$; that is, we must reduce problems in L_{H} to problems in L_ε. More precisely, the function f must take strings of the form $\langle M, w\rangle$ to strings of the form $\langle M'\rangle$ in such a way that M halts on w if and only if M' halts on ε. Of course, it is also necessary to show that f can be carried out by some Turing machine. In practice it is sufficient to describe an algorithm that computes f, with the implicit understanding that any algorithm can be carried out by a Turing machine.

We can easily show that such a mapping exists and can be carried out algorithmically. The idea is to construct a Turing machine M' from M and w that behaves as follows: M' first writes the string $\$w$ onto the tape, then positions the head over the first cell of the tape, and finally "runs" machine M; that is, it enters a state that begins a computation identical to what M would do. The machine M' is very easy to define given both M and w. We need only change the start state of M and add some new states that write the string w. Of course, the function f that carries out this transformation must begin by "decoding" the string $\langle M, w\rangle$, and end by "encoding" the new machine M' as a string.

This mapping has all of the required characteristics: M' run on ε will behave exactly like M run on w (it will halt if and only if M will halt on w), and each of the steps performed by f can be carried out by a simple program. We have thus established the undecidability of L_ε.

Intuitively, what we have shown is that we could build a "decider" for L_{H} if we were given a "decider" for L_ε; by contradiction then, L_ε must be undecidable. It is crucial to note the direction in which we have done the reduction, or, equivalently, which decider we are using as a "subroutine." We reduce the problem that is *known* to be undecidable (in this case L_{H}) to the problem whose undecidability we wish to establish (in this case L_ε). This is equivalent to showing that we could decide the former (L_{H}) given a subroutine for deciding the latter (L_ε).

Observe that the reversed relation, $L_\varepsilon \leq_m L_{\text{H}}$, holds quite trivially and tells us nothing at all about L_ε. That is, imagine that we had a decider for L_{H}. Then it would be trivial to build a decider for L_ε by mapping strings of the form $\langle M\rangle$ to strings of the form $\langle M, \varepsilon\rangle$ and then running the decider for L_{H} on the latter. However, this does *not* imply that L_ε is undecidable, as it does not preclude the existence of some other method for deciding L_ε that does not rely upon a decider for L_{H}.

8.3 Classes of Undecidable Languages

Let **R** denote the class of *recursive* (or *decidable*) languages, and let **RE** denote the class of *recursively enumerable* (or *Turing-acceptable*) languages. Clearly $\mathbf{R} \subset \mathbf{RE}$, since all Turing-decidable languages are automatically Turing-acceptable. However, $\mathbf{R} \neq \mathbf{RE}$, since there are undecidable languages that are r.e. Another distinct class of languages is denoted **co-RE**, and defined to consist of those languages whose *complements* are r.e., whether or not they are themselves r.e. These three classes are related as shown in Figure 8.3.

An example of a language in $\mathbf{RE} - \mathbf{R}$ is L_{H}. An example of a language in $\textbf{co-RE} - \mathbf{R}$ is $\overline{L_{\text{H}}}$. Of course, examples of languages in **R** abound since every regular language and every context-free

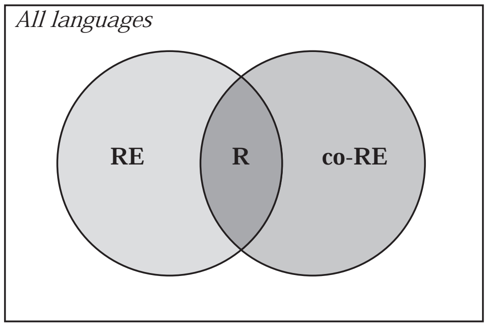

Figure 8.3: *All languages outside of* **R** *are undecidable. Those in* **RE** *are recursively enumerable, and the complements of those in* **co-RE** *are recursively enumerable. All three classes are distinct, and together they are a proper subset of all languages.*

language is in this set. But is every language either in **RE** or **co-RE**? A straightforward cardinality argument shows that this is not so; in fact, almost all languages fail to have either property.

8.4 Rice's Theorem

In this section we introduce a theorem that demonstrates in one broad stroke that a very large class of languages is undecidable. In essence, it says that we can decide almost nothing about languages based purely on the Turing machines that accept them.

Theorem 39 (Informal Statement of Rice's Theorem) *Any language consisting of (encodings of) Turing machines whose languages possess some non-trivial property is undecidable.*

To make this theorem precise, we need to define what is meant by a "non-trivial property" of a language. We may give this an extremely general interpretation: a "non-trivial" property of a language will simply mean membership in some subset of r.e. languages, provided that the subset is neither empty nor the entire class **RE**. Note that we can restrict our attention to only r.e. languages, since the theorem is stated in terms of the languages defined by (i.e. accepted by) Turing machines; hence, every language the theorem applies to is already in **RE**.

Theorem 40 (Formal Statement of Rice's Theorem) *Let S be any non-trivial subset of the class* **RE** *of r.e. languages. That is, $S \subset$* **RE** *with $S \neq \emptyset$ and $S \neq$* **RE**. *Then the language*

$$L_S = \{\langle M \rangle \mid \mathcal{L}(M) \in S\}$$

is undecidable.

Proof: We will show that for any non-trivial subset S of **RE**, $L_\varepsilon \leq_m L_S$. That is, we will show that there exists an algorithm for computing a function f such that M halts on ε if and only if

$\mathcal{L}(f(M)) \in S$. First, let L_0 be any language in S. Now define the function f in such a way that $f(\langle M \rangle) = \langle M' \rangle$, where M' is a Turing machine with the property that

$$\mathcal{L}(M') = \begin{cases} L_0 & \text{if } M \text{ halts on } \varepsilon \\ \emptyset & \text{if } M \text{ does not halt on } \varepsilon \end{cases} \tag{8.1}$$

The machine M' is easy to construct algorithmically from M and M_0, where M_0 is a Turing machine that accepts L_0. In particular, we construct M' so that it initially ignores its own input string w' and runs M on ε (in such a way that it does not disturb the string w'). If M eventually halts on ε, M' then runs M_0 on its own input string w' and accepts it if and only if M_0 accepts it. As a consequence of this construction, if M loops forever on ε, then the machine M' loops forever on *all* input strings and therefore $\mathcal{L}(M') = \emptyset$. On the other hand, if M halts on ε, then M' accepts the input string w' if and only if it is in L_0, which is to say $\mathcal{L}(M') = L_0$.

The function f *almost* has the desired properties; the only problem occurs when $\emptyset \in S$; that is, when S contains the empty language. When $\emptyset \in S$, the function f does not help us to distinguish between strings in L_ε and those outside L_ε, since both cases in Equation (8.1) are then languages in S. To fix this problem observe that if $\emptyset \in S$, then the construction above shows that the complement of S (with respect to the universe **RE**) is undecidable, since it does not contain the empty language $\emptyset$, and the above proof works in that case. But a language in **RE** is decidable if and only if its complement with respect to **RE** is decidable. Therefore, L_S is undecidable whether or not S contains the empty language. $\Box$

Note that Rice's theorem applies to properties of the *languages accepted by* Turing machines, not to the Turing machines themselves, or specifics of their operation. For example, the language

$$L_1 = \{\langle M \rangle \mid M \text{ has an even number of states}\}$$

is obviously decidable, since we need only decode and examine the machine's definition to determine how many states it has. But now consider the language

$$L_2 = \{\langle M \rangle \mid M \text{ never writes three consecutive blanks on its tape given input } \varepsilon\}\,.$$

The language L_2 is undecidable (in fact, it is not even r.e.), which can be easily demonstrated either by showing $\overline{L_\varepsilon} \leq_m L_2$, or $\overline{L_{\text{H}}} \leq_m L_2$. However, the undecidability of L_2 does not follow from Rice's theorem since it is not defined in terms of a property of the languages accepted by Turing machines.

8.5 Post's Correspondence Problem

Thus far, all of the undecidable languages that we have encountered have involved encodings of Turing machines, which makes them seem rather artificial. This section describes an unsolvable problem (which corresponds to an undecidable language) known as *Post's correspondence problem*, or PCP, that can be stated in terms of a simple "card game." In this "game" we are given a finite set of "cards," each with two non-empty strings of symbols on them; one across the top of the card and one across the bottom of the card. One of the cards is designated as the "starting" card. The problem is to determine whether there is a finite sequence of these cards, beginning with the starting

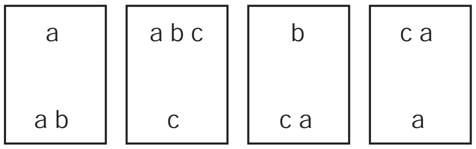

Figure 8.4: *This is an instance of PCP. Here the first card is to be the starting card.*

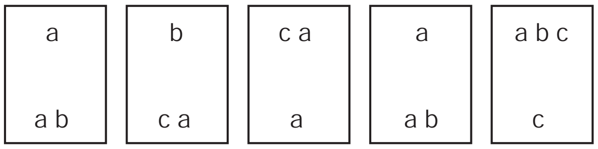

Figure 8.5: *This is a solution to the previous instance of PCP. Note that it starts with the designated starting card, and that one of the cards is repeated. In this example, all of the cards are used, although this needn't be the case in general.*

card and using any number of repetitions of each card, such that the string formed by concatenating the symbols across the tops of all the cards exactly matches the string formed by concatenating the symbols across the bottoms of the cards. For example, consider the collection of cards shown in Figure 8.4, where the first card is designated as the "starting" card. In this case, there is a solution to the Post correspondence problem, as shown in Figure 8.5. In the solution, both the top and bottom rows read "abcaaabc", and this sequence begins with the designated starting card. Notice that one of the cards has been used twice; not all cards need appear in a solution. More formally, we will define an instance of Post's correspondence problem to be a 3-tuple, (Σ, C, S), where Σ is an alphabet, C is a finite subset of $(\Sigma^* - \{\varepsilon\}) \times (\Sigma^* - \{\varepsilon\})$ representing the "cards", and $S \in C$ is the starting card. Since PCP is a decision problem, we may express it as a language recognition problem. Accordingly, we define L_{PCP} to be the language consisting of (encodings of) all PCP problem instances that have solutions. In other words, testing a string for membership in L_{PCP} is equivalent to determining whether the encoded PCP has a solution. The following theorem states a surprising fact about Post's correspondence problem.

Theorem 41 *The language L_{PCP} is undecidable.*

Proof: We will show that $L_{\text{A}} \leq_m L_{\text{PCP}}$, where $L_{\text{A}} = \{\langle M, w\rangle \mid w \in \mathcal{L}(M)\}$, which is undecidable. We exhibit an easily computed mapping f from (M, w) to an instance of PCP that is solvable if and only if $w \in \mathcal{L}(M)$. The PCP instance is constructed in such a way that the only feasible sequences of cards are those that mimic the sequence of configurations that result when M is run on input w; the sequence terminates, resulting in a solution, if and only if M ultimately accepts w.

Figure 8.6 shows the nine different patters of cards that are defined for a specific Turing machine M and input w; we assume here that the tape alphabet Γ of M does not contain the symbol "#", and that $\Gamma \cap Q = \emptyset$. Card number 1 is the starting card, and its bottom string depicts the initial configuration of M with input w, flanked by the special character "#". A card with pattern 3 is added for each symbol a in Γ. Pattern 4 is created for each transition of the form (p, a, q, b, R) in

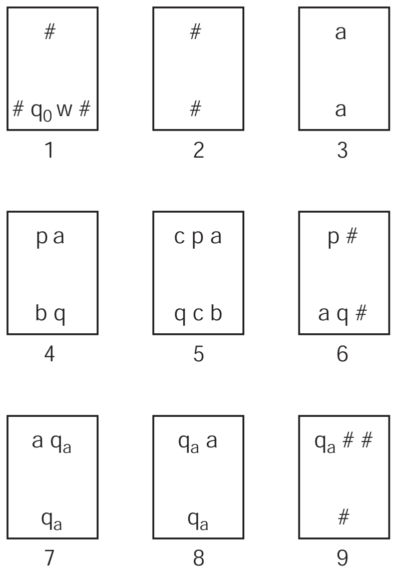

Figure 8.6: *Nine types of cards are required to mimic the operation of a given Turing machine on a given string w with an instance of PCP. Card 1 is the starting card, and every type of card except 1 and 2 have multiple versions corresponding to all the transitions and tape symbols of the Turing machine.*

the transition relation Δ of M. Pattern 5 is created for each transition $(p, a, q, b, L) \in \Delta$ and each symbol $c \in \Gamma$. Pattern 6 is created for each transition $(p, \$, q, a, R) \in \Delta$, where "\$" is the blank symbol. Finally, patters 7, 8, and 9 are created for the accept state q_h and each symbol $a \in \Gamma$.

To generate a solution for this PCP instance, it is necessary to generate the string appearing at the bottom of the starting card (minus the leading "#") across the tops of the subsequent cards. The cards are designed so that this will always be possible, and that such a sequence will automatically generate the *next* configuration of the Turing machine along the *bottom* row. By again adding a sequence of cards to match this new string along the top, we thereby generate the next configuration along the bottom, which now needs to be matched, and so on. Cards 3, 4, 5, and 6 are the key to this behavior. Card 3 allows us to copy the contents of the tape, but not the state symbol. Cards 4 and 5 allow us to copy the state symbol along the top, while also creating a string along the bottom that depicts the new configuration, after the head moves right or left, respectively. Card 6 allows more symbols to be added as the head moves beyond the last non-blank symbol currently on the tape. This process continues until the accept state is entered. Cards 7 and 8 then allow the top row to finally catch up by gradually "erasing" each symbol from an accepting configuration. Finally, card 9 finishes the sequence of cards by "erasing" the accept state. This scenario is demonstrated in Figure 8.8, which shows the sequence of cards that mimic the sequence of configurations

$$\begin{aligned} (\$, q_0, a\$\$) &\vdash (\$b, q_2, \$\$) \\ &\vdash (\$b\$, q_h, \$), \end{aligned}$$

where "\$" is the blank symbol, q_0 is the start state, and q_h is the accept state.

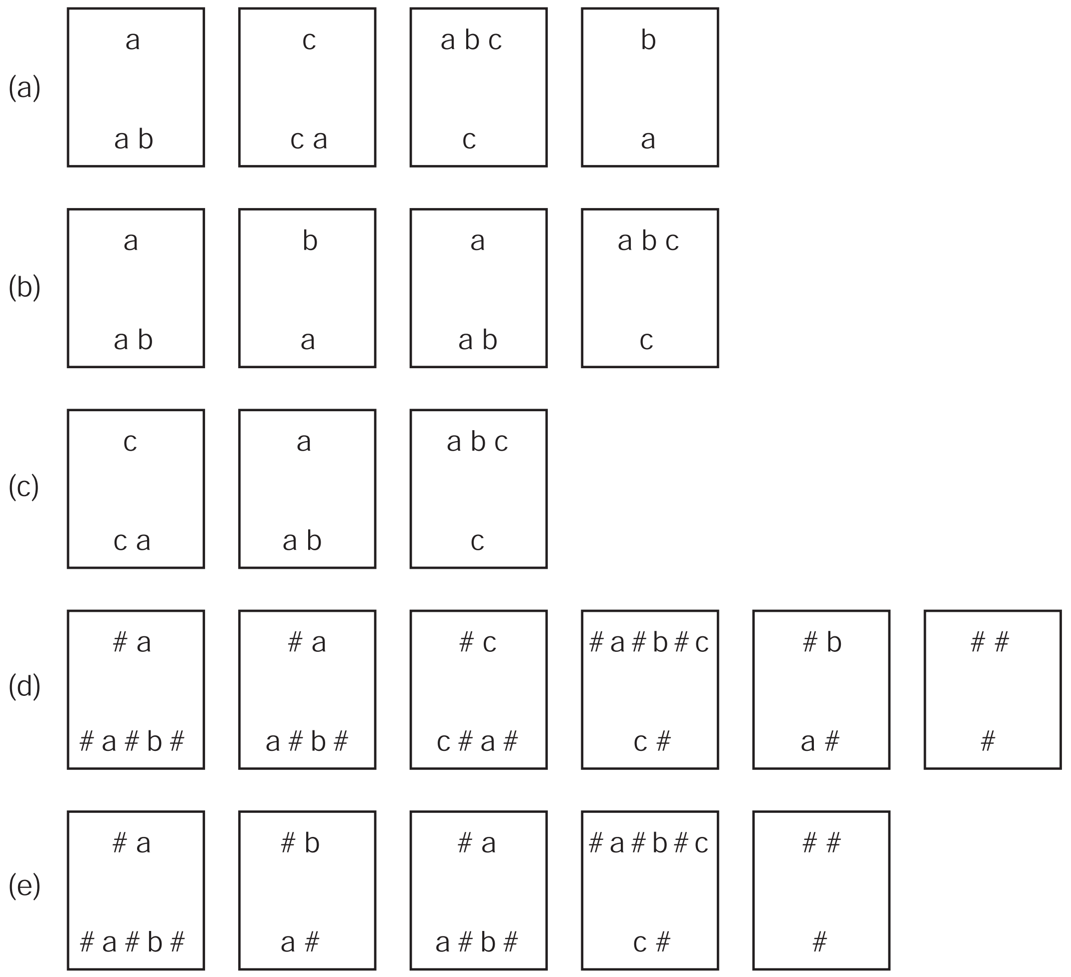

Figure 8.7: *Line (a) shows an instance of PCP, where the first card is to be the starting card. This instance has a solution, shown in line (b). If we allow any card to be the starting card, there is also another solution, shown in line (c). Line (d) shows the modified cards, with one new card, where "#" is a symbol not in the original alphabet. The card corresponding to the original starting card is now the* only *card that can begin a solution. Line (e) shows a solution with the modified cards.*

Post's correspondence problem is frequently stated in a slightly different form, where a solution can begin with any of the cards. We'll call this version the *Unconstrained* PCP, or UPCP, since we are free to start with any card. We shall denote the corresponding language by L_{UPCP}. While L_{UPCP} is also undecidable, this fact is not an immediate consequence of the undecidability of L_{PCP}; it requires a proof.

Theorem 42 *The language L_{UPCP}, in which a PCP solution is allowed to start with any of the cards, is undecidable.*

Proof: We shall show that $L_{\text{PCP}} \leq_m L_{\text{UPCP}}$, which implies that L_{UPCP} is undecidable. To do this, we must show how to transform an instance of PCP into an instance of UPCP in such a way that the former has a solution if and only if the latter has a solution. One way to accomplish this is to alter the cards slightly so that any existing solution (beginning with the designated card) is preserved, yet it is provably impossible to form a solution beginning with any card except the one that is designated to be the starting card. We can do this with the construction shown in Figure 8.7, where "#" is assumed to be a letter not in the original alphabet. If the original problem instance has n cards, the modified problem instance has $n+2$ cards. The new instance includes two versions of the starting card; one that is easily seen to be the only candidate for beginning a solution, and one that can occur anywhere else in a solution. The new instance also includes the card $(\#\#, \#)$, which is the only candidate for ending a solution; however, this card cannot begin a solution, since none of the other cards can possibly follow it. □

8.6 Subsets and Supersets of Undecidable Languages

In general, subsets of undecidable languages needn't be undecidable themselves since every finite subset is decidable; in addition, there may well be infinite subsets consisting of "easy" instances that can be decided. For example, L_{PCP} has an infinite subset that is trivially decidable: the set of all instances consisting of a single card. These trivial instances have a solution if and only if the top and bottom strings on the single card match. As another example, the subset of L_{PCP} consisting of instances with $|\Sigma| = 1$ is also decidable.

Similarly, supersets of undecidable languages needn't be undecidable themselves. As a trivial example, observe that $L_{\text{H}} \subset \Sigma^*$, where Σ is the alphabet of L_{H}. Clearly Σ^* is decidable, since it is regular. However, there is an important special case in which supersets of undecidable languages *are* undecidable. Suppose that $L_1 = S \cap L_2$, where S is a decidable language. Then $L_1 \subset L_2$, and L_2 is undecidable if L_1 is. This follows from the fact that decidable languages are closed under intersection. We think of L_1 as being a *subproblem* of L_2; that is, L_1 consists of those elements in L_2 that also have some additional property that is easy to discern. For example, let L^*_{PCP} denote the language consisting of generalized PCP instances that have solutions. Here "generalized" means that the strings can contain "wild symbols" that match any symbol, by definition. We can represent such an instance using the same (Σ, C, S) convention by allowing the use of symbols that are not in Σ and declaring any such symbol to be a "wild symbol". Then

$$L_{\text{PCP}} = S \cap L^*_{\text{PCP}},$$

where S consists of all PCP instances in which *no* wild symbols appear; that is, all the symbols appearing in the strings are in Σ. Since S is easy to decide, and L_{PCP} is undecidable, it follows that the generalized problem L^*_{PCP} is also undecidable. The rule is this: If a language L is undecidable, then so is any superset of L provided that there is an algorithm to distinguish the new elements from the original elements of L.

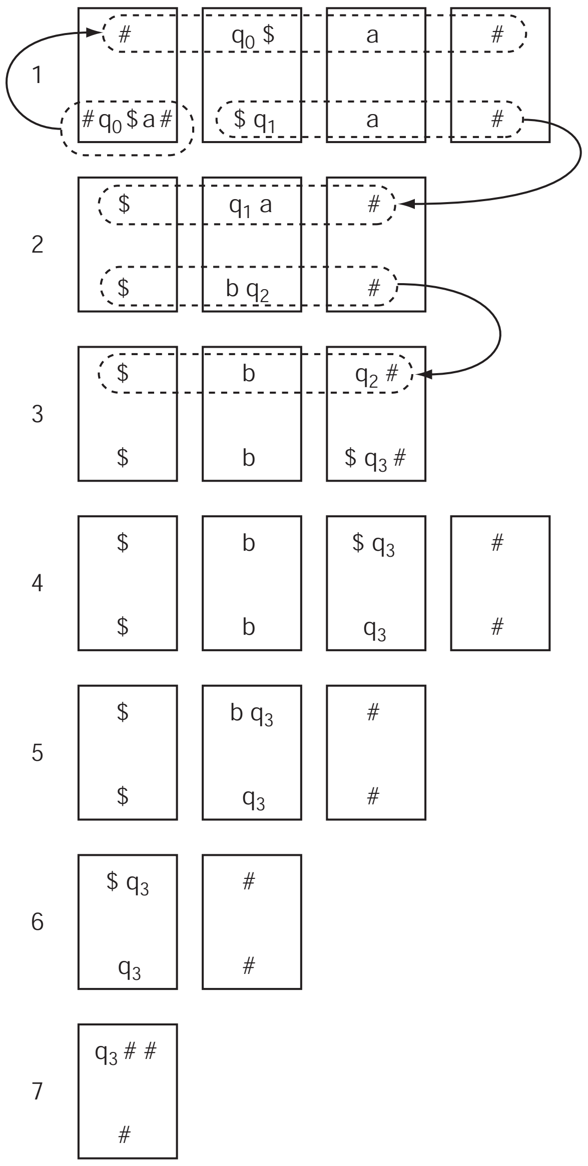

Figure 8.8: *In this example, the cards of a PCP instance simulate the operation of a Turing machine that overwrites the string "a" with the string "b" and then accepts. The seven rows of cards are to be placed in sequence from top to bottom; they are depicted in rows to emphasize the sequence of configurations. The pairs of dashed regions connected by arrows indicate strings that match. Following the starting card the only feasible strategy is to add cards that copy the hitherto unmatched portion of the bottom string to the top. Doing so creates a new (unmatched) string along the bottom that encodes the next configuration of the Turing machine. Rows 5, 6, and 7 show how the accepting configuration is matched and the sequence terminated.*

8.7 Exercises

1. Consider the language

$$L = \{\langle M, w\rangle \mid M \text{ never writes a blank on its tape given input } w\}$$

Prove that L is not recursively enumerable.

2. The following questions concern Post's Correspondence Problem, or the language L_{PCP}. If ϕ represents a specific PCP problem instance (i.e. a specific set of "cards"), then let $n(\phi)$ denote the number of pairs (i.e. the number of "cards"), let $m(\phi)$ denote the number of distinct symbols used, and let $\ell(\phi)$ denote the length of the longest string of symbols in any pair (i.e. the longest string appearing on any "card"). Finally, let L_{pcp} denote the language of *all* PCP problem instances that have solutions. (Note that the strings of this language clearly must be encoded in some fixed alphabet. Also, redundancy in the form of re-ordered cards is of no concern here. That is, the same cards in a different order correspond to different strings in the language.)

3. Show that when the symbols are restricted to a fixed alphabet Σ with $|\Sigma| = 1$, then PCP is solvable. Here "solvable" means that there is a procedure by which we can determine *whether or not* any given problem instance has a matching sequence of pairs (i.e. show that L_{pcp} is *decidable*).

4. Give an explicit formula for a function $f : \mathbb{N}^3 \to \mathbb{N}$ such that $f(i, j, k)$ is an *upper bound* on the number of distinct PCP problem instances ϕ with $n(\phi) = i$, $m(\phi) = j$, and $\ell(\phi) = k$. (Note that simply re-naming the symbols does not result in a distinct problem instance, otherwise there would be infinitely many instances for any fixed i, j, and k. Also, note that you needn't give the exact number of instances – any finite number that is larger will suffice.)

5. Show that there exists a well-defined mathematical function $g : \mathbb{N}^3 \to \mathbb{N}$ with the following property. If ϕ is any PCP problem instance, then either there is a solution consisting of $g(n(\phi), m(\phi), \ell(\phi))$ or fewer pairs ("cards"), or ϕ has no solution. That is, the three basic facts about the problem instance (number of cards, number of symbols, and length of longest string) is sufficient to bound the length of the shortest solution. You *cannot* give an explicit formula for g in terms of elementary functions, but you *can* show that such a function exists! This would not be possible unless $f(i, j, k)$ were always finite. Hint: Every problem instance either has a solution or it does not, regardless of whether or not you can compute which of these situations holds.

6. Show that if you had a method for computing the function g defined above, then PCP would be solvable (L_{pcp} would be decidable). Show this by exhibiting pseudo-code or actual Lisp code that would solve PCP, given a hypothetical function PCP-BOUND that computes the function g. Conclude that such a function is therefore not computable.

7. Given that PCP is *not* solvable (L_{pcp} is undecidable), show that $g(n, n, n) > A(n!, n!)!$, for infinitely many $n \in \mathbb{N}$, where A is Ackerman's function, and "!" means *factorial.* Clearly, this function exhibits unimaginably explosive growth. Hint: This requires *no* special knowledge of Ackerman's function whatsoever.

8. Show that PCP is not solvable (i.e. L_{pcp} is *undecidable*) when the symbols are restricted to a fixed alphabet Σ with $|\Sigma| = 2$. You may use the fact that it is unsolvable (i.e. L_{pcp} is undecidable) when there is no restriction on the alphabet.

9. Show that the relation $\leq_m$ is transitive.

10. Give an upper bound on the number of distinct Turing machines that can be defined using n states and $k \geq 2$ symbols in the tape alphabet (ignoring the size of the input alphabet)?

Note: Most valid Turing machines are utterly useless, such as the machine with an empty transition relation, Δ, or with no transitions leading out of the initial state. These are nonetheless considered to be Turing machines. Also, two Turing machines will not be considered distinct if they differ only in the names of the states, or the names of the symbols in their tape alphabets, or the ordering of the elements in the transition relation.

11. Prove that there exists a function $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ with the following property. If M is a Turing machine with n states and k symbols in its tape alphabet, then when M is run on the empty string, it will either halt in $f(n, k)$ or fewer steps, or run forever. Hint: do not attempt to write an expression for f; simply show that such a function exists in principle (that is, mathematically).

12. Prove that for *any* function $g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ for which there exists an algorithm (i.e. g can be computed by some Turing machine), $f(i, j) > g(i, j)$ for some i and j in N, where f is the function from part 11 above. As one example, $f(i, j) > A(i, j)!$ for some i and j. Yes, $A(i, j)!$ really is the *factorial* of Ackermann's function, which means that f must exhibit truly astonishing growth. What can you conclude about the function f? Hint: Use the undecidability of L_{H}, and a proof by contradiction.

Bibliography

[1] Wilhelm Ackermann. On Hilbert's construction of the real numbers. In J. van Heijenoort, editor, *From Frege to Gödel. A Source Book in Mathematical Logic, 1879–1931*, pages 493–507. Harvard University Press, Cambridge, Massachusetts, 1967.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[3] Rudolf Carnap. *Introduction to Symbolic Logic and its Applications.* Dover Publications, New York, 1958.

[4] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Annual ACM Symposium on the Theory of Computing*, volume 3, pages 151–158, Ohio, May 1971.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw-Hill, New York, 1990.

[6] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science.* Academic Press, New York, second edition, 1994.

[7] Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of Presburger arithmetic. In *Proceedings of the SIAM-AMS Symposium in Applied Mathematics*, volume 7, pages 27–41, 1974.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeeman and Company, San Francisco, 1979.

[9] Paul R. Halmos. *Naive Set Theory.* Springer-Verlag, New York, 1998.

[10] Fred Hennie. *Introduction to Computability.* Addison-Wesley, Reading, Massachusetts, 1977.

[11] John M. Howie. *Automata and Languages.* Oxford University Press, New York, 1991.

[12] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[13] Stephen C. Kleene. Origins of recursive function theory. In *20th Annual Symposium on the Foundations of Computer Science*, pages 371–382, San Juan, Puerto Rico, October 1979.

[14] Dexter C. Kozen. *The Design and Analysis of Algorithms.* Springer-Verlag, New York, 1992.

[15] Dexter C. Kozen. *Automata and Computability.* Springer-Verlag, New York, 1997.

[16] E. V. Krishnamurthy. *Introductory Theory of Computer Science.* Springer-Verlag, New York, 1983.

[17] Harry R. Lewis and Christos H. Papadimitiou. *Elements of the Theory of Computation.* Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1998.

[18] Peter Linz. *An Introduction to Formal Languages and Automata.* Jones and Bartlett Publishers, Boston, second edition, 1997.

[19] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms.* Theory of Computation Series. North Holland, New York, 1978.

[20] Anil Nerode and Richard A. Shore. *Logic for Applications.* Graduate Texts in Computer Science. Springer-Verlag, New York, second edition, 1997.

[21] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, Boston, 1997.

[22] L. J. Stockmeyer. Planar 3-colorability is NP-complete. *SIGACT News*, 5(3):19–25, 1973.

[23] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science.* Addison-Wesley, Reading, Massachusetts, second edition, 1997.

[24] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, November 1936. (Also see correction, 43:544–546, 1937).

[25] Andrew Wiles. Modular elliptic curves and Fermat's last theorem. *Annals of Mathematics*, 141(3):443–551, May 1995.