NAVODAYA INSTITUTE OF TECHNOLOGY, RAICHUR

DEPARMENT OF COMPUTER SCIENCE & ENGINEERING

MODULE 5

Chapter 8

Introduction to Turing Machines

In this chapter we change our direction significantly. Until now, we have been interested primarily in simple classes of languages and the ways that they can be used for relatively constrained problems, such as analyzing protocols, searching text, or parsing programs. Now, we shall start looking at the question of what languages can be defined by any computational device whatsoever. This question is tantamount to the question of what computers can do, since recognizing the strings in a language is a formal way of expressing any problem, and solving a problem is a reasonable surrogate for what it is that computers do.

We begin with an informal argument, using an assumed knowledge of C programming, to show that there are specific problems we cannot solve using a computer. These problems are called "undecidable." We then introduce a venerable formalism for computers, called the Turing machine. While a Turing machine looks nothing like a PC, and would be grossly inefficient should some startup company decide to manufacture and sell them, the Turing machine long has been recognized as an accurate model for what any physical computing device is capable of doing.

In Chapter 9, we use the Turing machine to develop a theory of "undecidable" problems, that is, problems that no computer can solve. We show that a number of problems that are easy to express are in fact undecidable. An example is telling whether a given grammar is ambiguous, and we shall see many others.

8.1 Problems That Computers Cannot Solve

The purpose of this section is to provide an informal, C-programming-based introduction to the proof of a specific problem that computers cannot solve. The particular problem we discuss is whether the first thing a C program prints

is `hello, world`. Although we might imagine that simulation of the program would allow us to tell what the program does, we must in reality contend with programs that take an unimaginably long time before making any output at all. This problem — not knowing when, if ever, something will occur — is the ultimate cause of our inability to tell what a program does. However, proving formally that there is no program to do a stated task is quite tricky, and we need to develop some formal mechanics. In this section, we give the intuition behind the formal proofs.

8.1.1 Programs that Print "Hello, World"

In Fig. 8.1 is the first C program met by students who read Kernighan and Ritchie's classic book.[1] It is rather easy to discover that this program prints `hello, world` and terminates. This program is so transparent that it has become a common practice to introduce languages by showing how to write a program to print `hello, world` in those languages.

```
main()
{
    printf("hello, world\n");
}
```

Figure 8.1: Kernighan and Ritchie's hello-world program

However, there are other programs that also print `hello, world`; yet the fact that they do so is far from obvious. Figure 8.2 shows another program that might print `hello, world`. It takes an input n, and looks for positive integer solutions to the equation $x^n + y^n = z^n$. If it finds one, it prints `hello, world`. If it never finds integers x, y, and z to satisfy the equation, then it continues searching forever, and never prints `hello, world`.

To understand what this program does, first observe that `exp` is an auxiliary function to compute exponentials. The main program needs to search through triples (x, y, z) in an order such that we are sure we get to every triple of positive integers eventually. To organize the search properly, we use a fourth variable, `total`, that starts at 3 and, in the while-loop, is increased one unit at a time, eventually reaching any finite integer. Inside the while-loop, we divide `total` into three positive integers x, y, and z, by first allowing x to range from 1 to `total-2`, and within that for-loop allowing y to range from 1 up to one less than what x has not already taken from `total`. What remains, which must be between 1 and `total-2`, is given to z.

In the innermost loop, the triple (x, y, z) is tested to see if $x^n + y^n = z^n$. If so, the program prints `hello, world`, and if not, it prints nothing.

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 1978, Prentice-Hall, Englewood Cliffs, NJ.

```
int exp(int i, n)
/* computes i to the power n */
{
    int ans, j;
    ans = 1;
    for (j=1; j<=n; j++) ans *= i;
    return(ans);
}

main ()
{
    int n, total, x, y, z;
    scanf("%d", &n);
    total = 3;
    while (1) {
        for (x=1; x<=total-2; x++)
            for (y=1; y<=total-x-1; y++) {
                z = total - x - y;
                if (exp(x,n) + exp(y,n) == exp(z,n))
                    printf("hello, world\n");
            }
        total++;
    }
}
```

Figure 8.2: Fermat's last theorem expressed as a hello-world program

If the value of n that the program reads is 2, then it will eventually find combinations of integers such as `total` $= 12$, $x = 3$, $y = 4$, and $z = 5$, for which $x^n + y^n = z^n$. Thus, for input 2, the program *does* print `hello, world`.

However, for any integer $n > 2$, the program will never find a triple of positive integers to satisfy $x^n + y^n = z^n$, and thus will fail to print `hello, world`. Interestingly, until a few years ago, it was not known whether or not this program would print `hello, world` for some large integer n. The claim that it would not, i.e., that there are no integer solutions to the equation $x^n + y^n = z^n$ if $n > 2$, was made by Fermat 300 years ago, but no proof was found until quite recently. This statement is often referred to as "Fermat's last theorem."

Let us define the *hello-world problem* to be: determine whether a given C program, with a given input, prints `hello, world` as the first 12 characters that it prints. In what follows, we often use, as a shorthand, the statement about a program that it prints `hello, world` to mean that it prints `hello, world` as the first 12 characters that it prints.

It seems likely that, if it takes mathematicians 300 years to resolve a question

Why Undecidable Problems Must Exist

While it is tricky to prove that a specific problem, such as the "hello-world problem" discussed here, must be undecidable, it is quite easy to see why almost all problems must be undecidable by any system that involves programming. Recall that a "problem" is really membership of a string in a language. The number of different languages over any alphabet of more than one symbol is not countable. That is, there is no way to assign integers to the languages such that every language has an integer, and every integer is assigned to one language.

On the other hand programs, being finite strings over a finite alphabet (typically a subset of the ASCII alphabet), *are* countable. That is, we can order them by length, and for programs of the same length, order them lexicographically. Thus, we can speak of the first program, the second program, and in general, the ith program for any integer i.

As a result, we know there are infinitely fewer programs than there are problems. If we picked a language at random, almost certainly it would be an undecidable problem. The only reason that most problems *appear* to be decidable is that we rarely are interested in random problems. Rather, we tend to look at fairly simple, well-structured problems, and indeed these are often decidable. However, even among the problems we are interested in and can state clearly and succinctly, we find many that are undecidable; the hello-world problem is a case in point.

about a single, 22-line program, then the general problem of telling whether a given program, on a given input, prints `hello, world` must be hard indeed. In fact, any of the problems that mathematicians have not yet been able to resolve can be turned into a question of the form "does this program, with this input, print `hello, world`? Thus, it would be remarkable indeed if we could write a program that could examine any program P and input I for P, and tell whether P, run with I as its input, would print `hello, world`. We shall prove that no such program exists.

8.1.2 The Hypothetical "Hello, World" Tester

The proof of impossibility of making the hello-world test is a proof by contradiction. That is, we assume there is a program, call it H, that takes as input a program P and an input I, and tells whether P with input I prints `hello, world`. Figure 8.3 is a representation of what H does. In particular, the only output H makes is either to print the three characters `yes` or to print the two characters `no`. It always does one or the other.

If a problem has an algorithm like H, that always tells correctly whether an instance of the problem has answer "yes" or "no," then the problem is said to

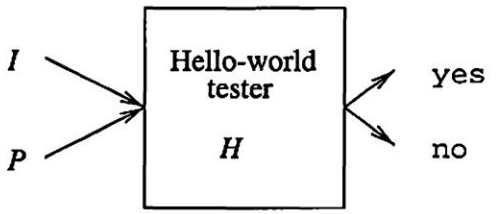

Figure 8.3: A hypothetical program H that is a hello-world detector

be "decidable." Otherwise, the problem is "undecidable." Our goal is to prove that H doesn't exist; i.e., the hello-world problem is undecidable.

In order to prove that statement by contradiction, we are going to make several changes to H, eventually constructing a related program called H_2 that we show does not exist. Since the changes to H are simple transformations that can be done to any C program, the only questionable statement is the existence of H, so it is that assumption we have contradicted.

To simplify our discussion, we shall make a few assumptions about C programs. These assumptions make H's job easier, not harder, so if we can show a "hello-world tester" for these restricted programs does not exist, then surely there is no such tester that could work for a broader class of programs. Our assumptions are:

1. All output is character-based, e.g., we are not using a graphics package or any other facility to make output that is not in the form of characters.

2. All character-based output is performed using `printf`, rather than `putchar()` or another character-based output function.

We now assume that the program H exists. Our first modification is to change the output `no`, which is the response that H makes when its input program P does not print `hello, world` as its first output in response to input I. As soon as H prints "n," we know it will eventually follow with the "o."[2] Thus, we can modify any `printf` statement in H that prints "n" to instead print `hello, world`. Another `printf` statement that prints an "o" but not the "n" is omitted. As a result, the new program, which we call H_1, behaves like H, except it prints `hello, world` exactly when H would print `no`. H_1 is suggested by Fig. 8.4.

Our next transformation on the program is a bit trickier; it is essentially the insight that allowed Alan Turing to prove his undecidability result about Turing machines. Since we are really interested in programs that take other programs as input and tell something about them, we shall restrict H_1 so it:

a) Takes only input P, not P and I.

[2]Most likely, the program would put `no` in one `printf`, but it could print the "n" in one `printf` and the "o" in another.

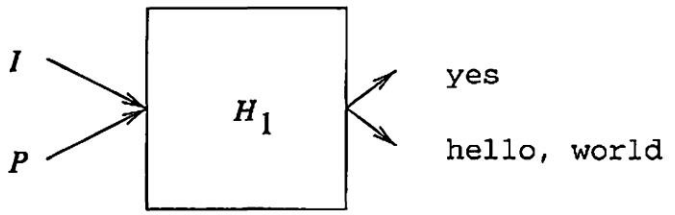

Figure 8.4: H_1 behaves like H, but it says `hello, world` instead of `no`

b) Asks what P would do if its input were its own code, i.e., what would H_1 do on inputs P as program and P as input I as well?

The modifications we must perform on H_1 to produce the program H_2 suggested in Fig. 8.5 are as follows:

1. H_2 first reads the entire input P and stores it in an array A, which it "malloc's" for the purpose.[3]

2. H_2 then simulates H_1, but whenever H_1 would read input from P or I, H_2 reads from the stored copy in A. To keep track of how much of P and I H_1 has read, H_2 can maintain two cursors that mark positions in A.

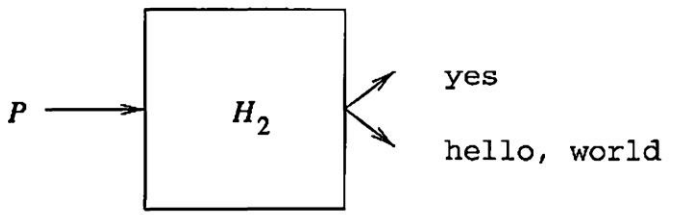

Figure 8.5: H_2 behaves like H_1, but uses its input P as both P and I

We are now ready to prove H_2 cannot exist. Thus, H_1 does not exist, and likewise, H does not exist. The heart of the argument is to envision what H_2 does when given itself as input. This situation is suggested in Fig. 8.6. Recall that H_2, given any program P as input, makes output `yes` if P prints `hello, world` when given itself as input. Also, H_2 prints `hello, world` if P, given itself as input, does not print `hello, world` as its first output.

Suppose that the H_2 represented by the box in Fig. 8.6 makes the output `yes`. Then the H_2 in the box is saying about its input H_2 that H_2, given itself

[3]The UNIX `malloc` system function allocates a block of memory of a size specified in the call to `malloc`. This function is used when the amount of storage needed cannot be determined until the program is run, as would be the case if an input of arbitrary length were read. Typically, `malloc` would be called several times, as more and more input is read and progressively more space is needed.

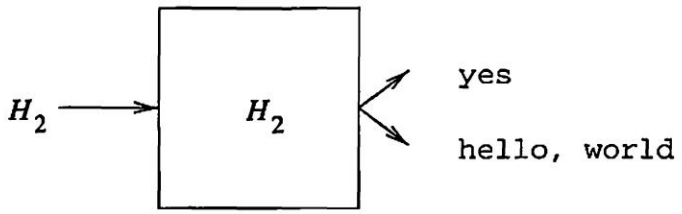

Figure 8.6: What does H_2 do when given itself as input?

as input, prints `hello, world` as its first output. But we just supposed that the first output H_2 makes in this situation is `yes` rather than `hello, world`.

Thus, it appears that in Fig. 8.6 the output of the box is `hello, world`, since it must be one or the other. But if H_2, given itself as input, prints `hello, world` first, then the output of the box in Fig. 8.6 must be `yes`. Whichever output we suppose H_2 makes, we can argue that it makes the other output.

This situation is paradoxical, and we conclude that H_2 cannot exist. As a result, we have contradicted the assumption that H exists. That is, we have proved that no program H can tell whether or not a given program P with input I prints `hello, world` as its first output.

8.1.3 Reducing One Problem to Another

Now, we have one problem — does a given program with given input print `hello, world` as the first thing it prints? — that we know no computer program can solve. A problem that cannot be solved by computer is called *undecidable*. We shall give the formal definition of "undecidable" in Section 9.3, but for the moment, let us use the term informally. Suppose we want to determine whether or not some other problem is solvable by a computer. We can try to write a program to solve it, but if we cannot figure out how to do so, then we might try a proof that there is no such program.

Perhaps we could prove this new problem undecidable by a technique similar to what we did for the hello-world problem: assume there is a program to solve it and develop a paradoxical program that must do two contradictory things, like the program H_2. However, once we have one problem that we know is undecidable, we no longer have to prove the existence of a paradoxical situation. It is sufficient to show that if we could solve the new problem, then we could use that solution to solve a problem we already know is undecidable. The strategy is suggested in Fig. 8.7; the technique is called the *reduction* of P_1 to P_2.

Suppose that we know problem P_1 is undecidable, and P_2 is a new problem that we would like to prove is undecidable as well. We suppose that there is a program represented in Fig. 8.7 by the diamond labeled "decide"; this program prints `yes` or `no`, depending on whether its input instance of problem P_2 is or is not in the language of that problem.[4]

[4]Recall that a problem is really a language. When we talked of the problem of deciding

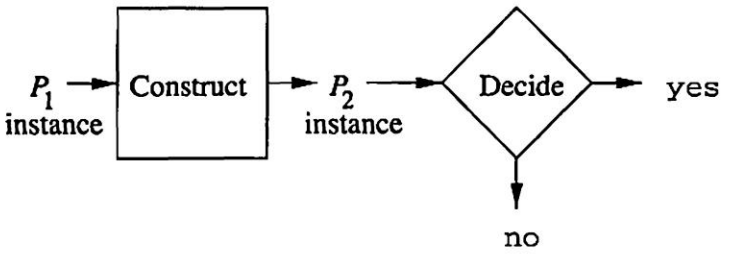

Figure 8.7: If we could solve problem P_2, then we could use its solution to solve problem P_1

In order to make a proof that problem P_2 is undecidable, we have to invent a construction, represented by the square box in Fig. 8.7, that converts instances of P_1 to instances of P_2 that have the same answer. That is, any string in the language P_1 is converted to some string in the language P_2, and any string over the alphabet of P_1 that is *not* in the language P_1 is converted to a string that is not in the language P_2. Once we have this construction, we can solve P_1 as follows:

1. Given an instance of P_1, that is, given a string w that may or may not be in the language P_1, apply the construction algorithm to produce a string x.

2. Test whether x is in P_2, and give the same answer about w and P_1.

If w is in P_1, then x is in P_2, so this algorithm says yes. If w is not in P_1, then x is not in P_2, and the algorithm says no. Either way, it says the truth about w. Since we assumed that no algorithm to decide membership of a string in P_1 exists, we have a proof by contradiction that the hypothesized decision algorithm for P_2 does not exist; i.e., P_2 is undecidable.

Example 8.1: Let us use this methodology to show that the question "does program Q, given input y, ever call function `foo`" is undecidable. Note that Q may not have a function `foo`, in which case the problem is easy, but the hard cases are when Q has a function `foo` but may or may not reach a call to `foo` with input y. Since we only know one undecidable problem, the role of P_1 in Fig. 8.7 will be played by the hello-world problem. P_2 will be the *calls-foo problem* just mentioned. We suppose there is a program that solves the calls-foo problem. Our job is to design an algorithm that converts the hello-world problem into the calls-foo problem.

That is, given program Q and its input y, we must construct a program R and an input z such that R, with input z, calls `foo` if and only if Q with input y prints `hello, world`. The construction is not hard:

whether a given program and input results in `hello, world` as the first output, we were really talking about strings consisting of a C source program followed by whatever input file(s) the program reads. This set of strings is a language over the alphabet of ASCII characters.

Can a Computer Really Do All That?

If we examine a program such as Fig. 8.2, we might ask whether it really searches for counterexamples to Fermat's last theorem. After all, integers are only 32 bits long in the typical computer, and if the smallest counterexample involved integers in the billions, there would be an overflow error before the solution was found. In fact, one could argue that a computer with 128 megabytes of main memory and a 30 gigabyte disk, has "only" $256^{30128000000}$ states, and is thus a finite automaton.

However, treating computers as finite automata (or treating brains as finite automata, which is where the FA idea originated), is unproductive. The number of states involved is so large, and the limits so unclear, that you don't draw any useful conclusions. In fact, there is every reason to believe that, if we wanted to, we could expand the set of states of a computer arbitrarily.

For instance, we can represent integers as linked lists of digits, of arbitrary length. If we run out of memory, the program can print a request for a human to dismount its disk, store it, and replace it by an empty disk. As time goes on, the computer could print requests to swap among as many disks as the computer needs. This program would be far more complex than that of Fig. 8.2, but not beyond our capabilities to write. Similar tricks would allow any other program to avoid finite limitations on the size of memory or on the size of integers or other data items.

1. If Q has a function called `foo`, rename it and all calls to that function. Clearly the new program Q_1 does exactly what Q does.

2. Add to Q_1 a function `foo`. This function does nothing, and is not called. The resulting program is Q_2.

3. Modify Q_2 to remember the first 12 characters that it prints, storing them in a global array A. Let the resulting program be Q_3.

4. Modify Q_3 so that whenever it executes any output statement, it then checks in the array A to see if it has written 12 characters or more, and if so, whether `hello, world` are the first 12 characters. In that case, call the new function `foo` that was added in item (2). The resulting program is R, and input z is the same as y.

Suppose that Q with input y prints `hello, world` as its first output. Then R as constructed will call `foo`. However, if Q with input y does not print `hello, world` as its first output, then R will never call `foo`. If we can decide whether R with input z calls `foo`, then we also know whether Q with input y (remember $y = z$) prints `hello, world`. Since we know that no algorithm to

The Direction of a Reduction Is Important

It is a common mistake to try to prove a problem P_2 undecidable by reducing P_2 to some known undecidable problem P_1; i.e., showing the statement "if P_1 is decidable, then P_2 is decidable." That statement, although surely true, is useless, since its hypothesis "P_1 is decidable" is false.

The only way to prove a new problem P_2 to be undecidable is to reduce a known undecidable problem P_1 to P_2. That way, we prove the statement "if P_2 is decidable, then P_1 is decidable." The contrapositive of that statement is "if P_1 is undecidable, then P_2 is undecidable." Since we know that P_1 undecidable, we can deduce that P_2 is undecidable.

decide the hello-world problem exists, and all four steps of the construction of R from Q could be carried out by a program that edited the code of programs, our assumption that there was a calls-foo tester is wrong. No such program exists, and the calls-foo problem is undecidable. □

8.1.4 Exercises for Section 8.1

Exercise 8.1.1: Give reductions from the hello-world problem to each of the problems below. Use the informal style of this section for describing plausible program transformations, and do not worry about the real limits such as maximum file size or memory size that real computers impose.

*! a) Given a program and an input, does the program eventually halt; i.e., does the program not loop forever on the input?

b) Given a program and an input, does the program ever produce *any* output?

! c) Given two programs and an input, do the programs produce the same output for the given input?

8.2 The Turing Machine

The purpose of the theory of undecidable problems is not only to establish the existence of such problems — an intellectually exciting idea in its own right — but to provide guidance to programmers about what they might or might not be able to accomplish through programming. The theory also has great pragmatic impact when we discuss, as we shall in Chapter 10, problems that although decidable, require large amounts of time to solve them. These problems, called "intractable problems," tend to present greater difficulty to the programmer

and system designer than do the undecidable problems. The reason is that, while undecidable problems are usually quite obviously so, and their solutions are rarely attempted in practice, the intractable problems are faced every day. Moreover, they often yield to small modifications in the requirements or to heuristic solutions. Thus, the designer is faced quite frequently with having to decide whether or not a problem is in the intractable class, and what to do about it, if so.

We need tools that will allow us to prove everyday questions undecidable or intractable. The technology introduced in Section 8.1 is useful for questions that deal with programs, but it does not translate easily to problems in unrelated domains. For example, we would have great difficulty reducing the hello-world problem to the question of whether a grammar is ambiguous.

As a result, we need to rebuild our theory of undecidability, based not on programs in C or another language, but based on a very simple model of a computer, called the Turing machine. This device is essentially a finite automaton that has a single tape of infinite length on which it may read and write data. One advantage of the Turing machine over programs as representation of what can be computed is that the Turing machine is sufficiently simple that we can represent its configuration precisely, using a simple notation much like the ID's of a PDA. In comparison, while C programs have a state, involving all the variables in whatever sequence of function calls have been made, the notation for describing these states is far too complex to allow us to make understandable, formal proofs.

Using the Turing machine notation, we shall prove undecidable certain problems that appear unrelated to programming. For instance, we shall show in Section 9.4 that "Post's Correspondence Problem," a simple question involving two lists of strings, is undecidable, and this problem makes it easy to show questions about grammars, such as ambiguity, to be undecidable. Likewise, when we introduce intractable problems we shall find that certain questions, seemingly having little to do with computation (e.g., satisfiability of boolean formulas), are intractable.

8.2.1 The Quest to Decide All Mathematical Questions

At the turn of the 20th century, the mathematician D. Hilbert asked whether it was possible to find an algorithm for determining the truth or falsehood of any mathematical proposition. In particular, he asked if there was a way to determine whether any formula in the first-order predicate calculus, applied to integers, was true. Since the first-order predicate calculus of integers is sufficiently powerful to express statements like "this grammar is ambiguous," or "this program prints `hello, world`," had Hilbert been successful, these problems would have algorithms that we now know do not exist.

However, in 1931, K. Gödel published his famous incompleteness theorem. He constructed a formula in the predicate calculus applied to integers, which asserted that the formula itself could be neither proved nor disproved within

the predicate calculus. Gödel's technique resembles the construction of the self-contradictory program H_2 in Section 8.1.2, but deals with functions on the integers, rather than with C programs.

The predicate calculus was not the only notion that mathematicians had for "any possible computation." In fact predicate calculus, being declarative rather than computational, had to compete with a variety of notations, including the "partial-recursive functions," a rather programming-language-like notation, and other similar notations. In 1936, A. M. Turing proposed the Turing machine as a model of "any possible computation." This model is computer-like, rather than program-like, even though true electronic, or even electromechanical computers were several years in the future (and Turing himself was involved in the construction of such a machine during World War II).

Interestingly, all the serious proposals for a model of computation have the same power; that is, they compute the same functions or recognize the same languages. The unprovable assumption that any general way to compute will allow us to compute only the partial-recursive functions (or equivalently, what Turing machines or modern-day computers can compute) is known as *Church's hypothesis* (after the logician A. Church) or the *Church-Turing thesis*.

8.2.2 Notation for the Turing Machine

We may visualize a Turing machine as in Fig. 8.8. The machine consists of a *finite control*, which can be in any of a finite set of states. There is a *tape* divided into squares or *cells*; each cell can hold any one of a finite number of symbols.

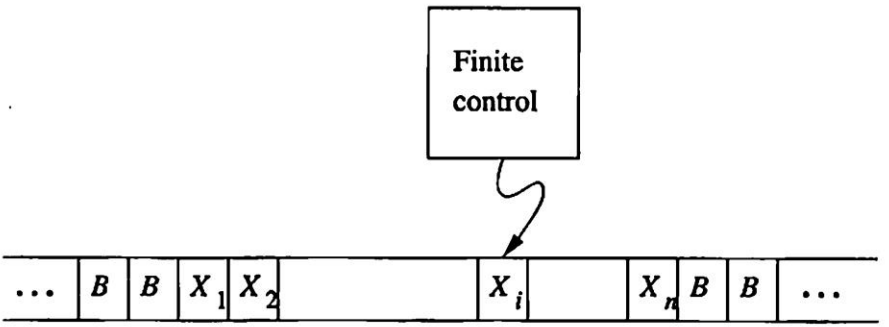

Figure 8.8: A Turing machine

Initially, the *input*, which is a finite-length string of symbols chosen from the *input alphabet*, is placed on the tape. All other tape cells, extending infinitely to the left and right, initially hold a special symbol called the *blank*. The blank is a *tape symbol*, but not an input symbol, and there may be other tape symbols besides the input symbols and the blank, as well.

There is a *tape head* that is always positioned at one of the tape cells. The Turing machine is said to be *scanning* that cell. Initially, the tape head is at

the leftmost cell that holds the input.

A *move* of the Turing machine is a function of the state of the finite control and the tape symbol scanned. In one move, the Turing machine will:

1. Change state. The next state optionally may be the same as the current state.
2. Write a tape symbol in the cell scanned. This tape symbol replaces whatever symbol was in that cell. Optionally, the symbol written may be the same as the symbol currently there.
3. Move the tape head left or right. In our formalism we require a move, and do not allow the head to remain stationary. This restriction does not constrain what a Turing machine can compute, since any sequence of moves with a stationary head could be condensed, along with the next tape-head move, into a single state change, a new tape symbol, and a move left or right.

The formal notation we shall use for a *Turing machine* (TM) is similar to that used for finite automata or PDA's. We describe a TM by the 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

whose components have the following meanings:

Q: The finite set of *states* of the finite control.

Σ: The finite set of *input symbols.*

Γ: The complete set of *tape symbols*; Σ is always a subset of Γ.

δ: The *transition function.* The arguments of $\delta(q, X)$ are a state q and a tape symbol X. The value of $\delta(q, X)$, if it is defined, is a triple (p, Y, D), where:

1. p is the next state, in Q.
2. Y is the symbol, in Γ, written in the cell being scanned, replacing whatever symbol was there.
3. D is a *direction*, either L or R, standing for "left" or "right," respectively, and telling us the direction in which the head moves.

q_0: The *start state*, a member of Q, in which the finite control is found initially.

B: The *blank* symbol. This symbol is in Γ but not in Σ; i.e., it is not an input symbol. The blank appears initially in all but the finite number of initial cells that hold input symbols.

F: The set of *final* or *accepting* states, a subset of Q.

8.2.3 Instantaneous Descriptions for Turing Machines

In order to describe formally what a Turing machine does, we need to develop a notation for configurations or *instantaneous descriptions* (ID's), like the notation we developed for PDA's. Since a TM, in principle, has an infinitely long tape, we might imagine that it is impossible to describe the configurations of a TM succinctly. However, after any finite number of moves, the TM can have visited only a finite number of cells, even though the number of cells visited can eventually grow beyond any finite limit. Thus, in every ID, there is an infinite prefix and an infinite suffix of cells that have never been visited. These cells must all hold either blanks or one of the finite number of input symbols. We thus show in an ID only the cells between the leftmost and the rightmost nonblanks. Under special conditions, when the head is scanning one of the leading or trailing blanks, a finite number of blanks to the left or right of the nonblank portion of the tape must also be included in the ID.

In addition to representing the tape, we must represent the finite control and the tape-head position. To do so, we embed the state in the tape, and place it immediately to the left of the cell scanned. To disambiguate the tape-plus-state string, we have to make sure that we do not use as a state any symbol that is also a tape symbol. However, it is easy to change the names of the states so they have nothing in common with the tape symbols, since the operation of the TM does not depend on what the states are called. Thus, we shall use the string $X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n$ to represent an ID in which

1. q is the state of the Turing machine.
2. The tape head is scanning the ith symbol from the left.
3. $X_1X_2\cdots X_n$ is the portion of the tape between the leftmost and the rightmost nonblank. As an exception, if the head is to the left of the leftmost nonblank or to the right of the rightmost nonblank, then some prefix or suffix of $X_1X_2\cdots X_n$ will be blank, and i will be 1 or n, respectively.

We describe moves of a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ by the $\vdash_M$ notation that was used for PDA's. When the TM M is understood, we shall use just $\vdash$ to reflect moves. As usual, $\vdash_M^*$, or just $\vdash^*$, will be used to indicate zero, one, or more moves of the TM M.

Suppose $\delta(q, X_i) = (p, Y, L)$; i.e., the next move is leftward. Then

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \vdash_M X_1X_2\cdots X_{i-2}pX_{i-1}YX_{i+1}\cdots X_n$$

Notice how this move reflects the change to state p and the fact that the tape head is now positioned at cell $i-1$. There are two important exceptions:

1. If $i = 1$, then M moves to the blank to the left of X_1. In that case,

$$qX_1X_2\cdots X_n \vdash_M pBYX_2\cdots X_n$$

2. If $i = n$ and $Y = B$, then the symbol B written over X_n joins the infinite sequence of trailing blanks and does not appear in the next ID. Thus,

$$X_1X_2\cdots X_{n-1}qX_n \underset{M}{\vdash} X_1X_2\cdots X_{n-2}pX_{n-1}$$

Now, suppose $\delta(q, X_i) = (p, Y, R)$; i.e., the next move is rightward. Then

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \underset{M}{\vdash} X_1X_2\cdots X_{i-1}YpX_{i+1}\cdots X_n$$

Here, the move reflects the fact that the head has moved to cell $i + 1$. Again there are two important exceptions:

1. If $i = n$, then the $i + 1$st cell holds a blank, and that cell was not part of the previous ID. Thus, we instead have

$$X_1X_2\cdots X_{n-1}qX_n \underset{M}{\vdash} X_1X_2\cdots X_{n-1}YpB$$

2. If $i = 1$ and $Y = B$, then the symbol B written over X_1 joins the infinite sequence of leading blanks and does not appear in the next ID. Thus,

$$qX_1X_2\cdots X_n \underset{M}{\vdash} pX_2\cdots X_n$$

Example 8.2: Let us design a Turing machine and see how it behaves on a typical input. The TM we construct will accept the language $\{0^n1^n \mid n \geq 1\}$. Initially, it is given a finite sequence of 0's and 1's on its tape, preceded and followed by an infinity of blanks. Alternately, the TM will change a 0 to an X and then a 1 to a Y, until all 0's and 1's have been matched.

In more detail, starting at the left end of the input, it repeatedly changes a 0 to an X and moves to the right over whatever 0's and Y's it sees, until it comes to a 1. It changes the 1 to a Y, and moves left, over Y's and 0's, until it finds an X. At that point, it looks for a 0 immediately to the right, and if it finds one, changes it to X and repeats the process, changing a matching 1 to a Y.

If the nonblank input is not in $\mathbf{0^*1^*}$, then the TM will eventually fail to have a next move and will die without accepting. However, if it finishes changing all the 0's to X's on the same round it changes the last 1 to a Y, then it has found its input to be of the form 0^n1^n and accepts. The formal specification of the TM M is

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$

where δ is given by the table in Fig. 8.9.

As M performs its computation, the portion of the tape, where M's tape head has visited, will always be a sequence of symbols described by the regular expression $\mathbf{X^*0^*Y^*1^*}$. That is, there will be some 0's that have been changed

State	Symbol 0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Figure 8.9: A Turing machine to accept $\{0^n1^n \mid n \geq 1\}$

to X's, followed by some 0's that have not yet been changed to X's. Then there are some 1's that were changed to Y's, and 1's that have not yet been changed to Y's. There may or may not be some 0's and 1's following.

State q_0 is the initial state, and M also enters state q_0 every time it returns to the leftmost remaining 0. If M is in state q_0 and scanning a 0, the rule in the upper-left corner of Fig. 8.9 tells it to go to state q_1, change the 0 to an X, and move right. Once in state q_1, M keeps moving right over all 0's and Y's that it finds on the tape, remaining in state q_1. If M sees an X or a B, it dies. However, if M sees a 1 when in state q_1, it changes that 1 to a Y, enters state q_2, and starts moving left.

In state q_2, M moves left over 0's and Y's, remaining in state q_2. When M reaches the rightmost X, which marks the right end of the block of 0's that have already been changed to X, M returns to state q_0 and moves right. There are two cases:

1. If M now sees a 0, then it repeats the matching cycle we have just described.

2. If M sees a Y, then it has changed all the 0's to X's. If all the 1's have been changed to Y's, then the input was of the form 0^n1^n, and M should accept. Thus, M enters state q_3, and starts moving right, over Y's. If the first symbol other than a Y that M sees is a blank, then indeed there were an equal number of 0's and 1's, so M enters state q_4 and accepts. On the other hand, if M encounters another 1, then there are too many 1's, so M dies without accepting. If it encounters a 0, then the input was of the wrong form, and M also dies.

Here is an example of an accepting computation by M. Its input is 0011. Initially, M is in state q_0, scanning the first 0, i.e., M's initial ID is q_00011. The entire sequence of moves of M is:

$$q_00011 \vdash Xq_1011 \vdash X0q_111 \vdash Xq_20Y1 \vdash q_2X0Y1 \vdash$$
$$Xq_00Y1 \vdash XXq_1Y1 \vdash XXYq_11 \vdash XXq_2YY \vdash Xq_2XYY \vdash$$
$$XXq_0YY \vdash XXYq_3Y \vdash XXYYq_3B \vdash XXYYBq_4B$$

For another example, consider what M does on the input 0010, which is not in the language accepted.

$$q_00010 \vdash Xq_1010 \vdash X0q_110 \vdash Xq_20Y0 \vdash q_2X0Y0 \vdash$$
$$Xq_00Y0 \vdash XXq_1Y0 \vdash XXYq_10 \vdash XXY0q_1B$$

The behavior of M on 0010 resembles the behavior on 0011, until in ID $XXYq_10$ M scans the final 0 for the first time. M must move right, staying in state q_1, which takes it to the ID $XXY0q_1B$. However, in state q_1 M has no move on tape symbol B; thus M dies and does not accept its input. □

8.2.4 Transition Diagrams for Turing Machines

We can represent the transitions of a Turing machine pictorially, much as we did for the PDA. A *transition diagram* consists of a set of nodes corresponding to the states of the TM. An arc from state q to state p is labeled by one or more items of the form X/YD, where X and Y are tape symbols, and D is a direction, either L or R. That is, whenever $\delta(q, X) = (p, Y, D)$, we find the label X/YD on the arc from q to p. However, in our diagrams, the direction D is represented pictorially by $\leftarrow$ for "left" and $\rightarrow$ for "right."

As for other kinds of transition diagrams, we represent the start state by the word "Start" and an arrow entering that state. Accepting states are indicated by double circles. Thus, the only information about the TM one cannot read directly from the diagram is the symbol used for the blank. We shall assume that symbol is B unless we state otherwise.

Example 8.3: Figure 8.10 shows the transition diagram for the Turing machine of Example 8.2, whose transition function was given in Fig. 8.9. □

Example 8.4: While today we find it most convenient to think of Turing machines as recognizers of languages, or equivalently, solvers of problems, Turing's original view of his machine was as a computer of integer-valued functions. In his scheme, integers were represented in unary, as blocks of a single character, and the machine computed by changing the lengths of the blocks or by constructing new blocks elsewhere on the tape. In this simple example, we shall show how a Turing machine might compute the function $\dot{-}$, which is called *monus* or *proper subtraction* and is defined by $m \dot{-} n = \max(m - n, 0)$. That is, $m \dot{-} n$ is $m - n$ if $m \geq n$ and 0 if $m < n$.

A TM that performs this operation is specified by

$$M = (\{q_0, q_1, \ldots, q_6\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B)$$

Note that, since this TM is not used to accept inputs, we have omitted the seventh component, which is the set of accepting states. M will start with a tape consisting of 0^m10^n surrounded by blanks. M halts with $0^{m \dot{-} n}$ on its tape, surrounded by blanks.

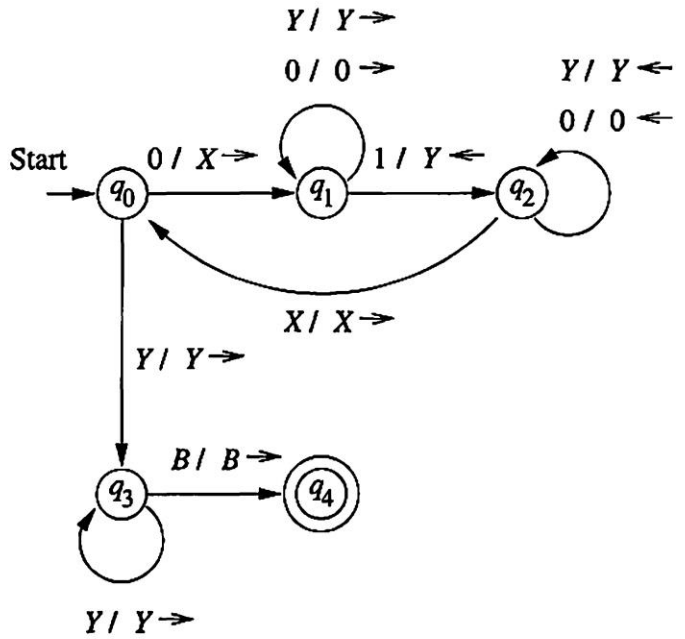

Figure 8.10: Transition diagram for a TM that accepts strings of the form 0^n1^n

M repeatedly finds its leftmost remaining 0 and replaces it by a blank. It then searches right, looking for a 1. After finding a 1, it continues right, until it comes to a 0, which it replaces by a 1. M then returns left, seeking the leftmost 0, which it identifies when it first meets a blank and then moves one cell to the right. The repetition ends if either:

1. Searching right for a 0, M encounters a blank. Then the n 0's in 0^m10^n have all been changed to 1's, and $n+1$ of the m 0's have been changed to B. M replaces the $n+1$ 1's by one 0 and n B's, leaving $m-n$ 0's on the tape. Since $m \geq n$ in this case, $m-n = m \dot{-} n$.

2. Beginning the cycle, M cannot find a 0 to change to a blank, because the first m 0's already have been changed to B. Then $n \geq m$, so $m \dot{-} n = 0$. M replaces all remaining 1's and 0's by B and ends with a completely blank tape.

Figure 8.11 gives the rules of the transition function δ, and we have also represented δ as a transition diagram in Fig. 8.12. The following is a summary of the role played by each of the seven states:

q_0: This state begins the cycle, and also breaks the cycle when appropriate. If M is scanning a 0, the cycle must repeat. The 0 is replaced by B, the head moves right, and state q_1 is entered. On the other hand, if M is

State	Symbol 0	1	B
q_0	(q_1, B, R)	(q_5, B, R)	–
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	–
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	–	–	–

Figure 8.11: A Turing machine that computes the proper-subtraction function

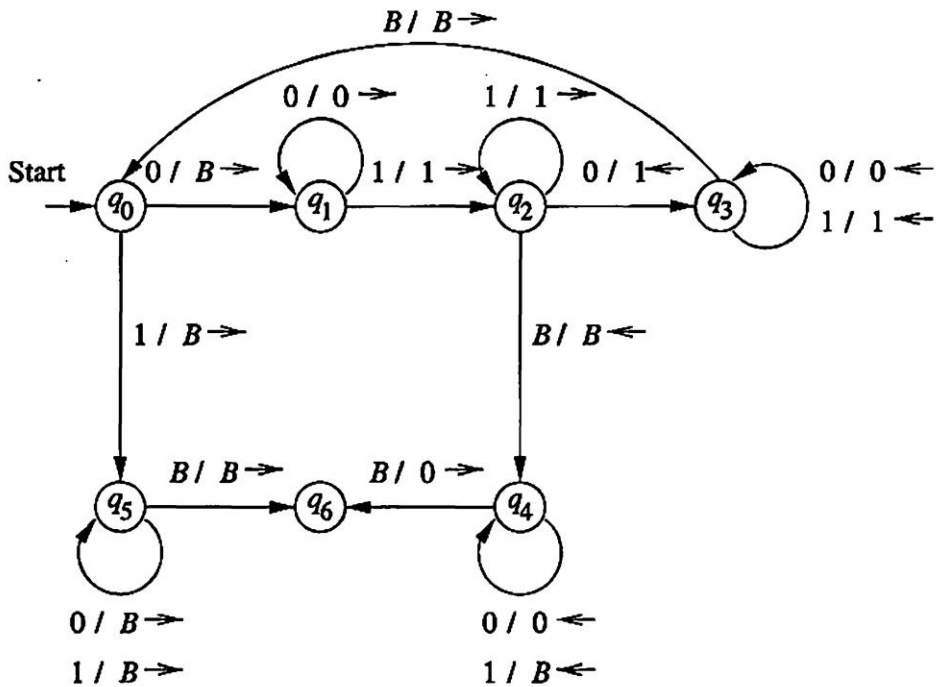

Figure 8.12: Transition diagram for the TM of Example 8.4

scanning 1, then all possible matches between the two groups of 0's on the tape have been made, and M goes to state q_5 to make the tape blank.

q_1: In this state, M searches right, through the initial block of 0's, looking for the leftmost 1. When found, M goes to state q_2.

q_2: M moves right, skipping over 1's, until it finds a 0. It changes that 0 to a 1, turns leftward, and enters state q_3. However, it is also possible that there are no more 0's left after the block of 1's. In that case, M in state q_2 encounters a blank. We have case (1) described above, where n 0's in the second block of 0's have been used to cancel n of the m 0's in the first block, and the subtraction is complete. M enters state q_4, whose purpose is to convert the 1's on the tape to blanks.

q_3: M moves left, skipping over 0's and 1's, until it finds a blank. When it finds B, it moves right and returns to state q_0, beginning the cycle again.

q_4: Here, the subtraction is complete, but one unmatched 0 in the first block was incorrectly changed to a B. M therefore moves left, changing 1's to B's, until it encounters a B on the tape. It changes that B back to 0, and enters state q_6, wherein M halts.

q_5: State q_5 is entered from q_0 when it is found that all 0's in the first block have been changed to B. In this case, described in (2) above, the result of the proper subtraction is 0. M changes all remaining 0's and 1's to B and enters state q_6.

q_6: The sole purpose of this state is to allow M to halt when it has finished its task. If the subtraction had been a subroutine of some more complex function, then q_6 would initiate the next step of that larger computation.

□

8.2.5 The Language of a Turing Machine

We have intuitively suggested the way that a Turing machine accepts a language. The input string is placed on the tape, and the tape head begins at the leftmost input symbol. If the TM eventually enters an accepting state, then the input is accepted, and otherwise not.

More formally, let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a Turing machine. Then $L(M)$ is the set of strings w in Σ^* such that $q_0 w \vdash^* \alpha p \beta$ for some state p in F and any tape strings α and β. This definition was assumed when we discussed the Turing machine of Example 8.2, which accepts strings of the form $0^n 1^n$.

The set of languages we can accept using a Turing machine is often called the *recursively enumerable languages* or RE languages, The term "recursively enumerable" comes from computational formalisms that predate the Turing machine but that define the same class of languages or arithmetic functions. We discuss the origins of the term as an aside (box) in Section 9.2.1.

Notational Conventions for Turing Machines

The symbols we normally use for Turing machines resemble those for the other kinds of automata we have seen.

1. Lower-case letters at the beginning of the alphabet stand for input symbols.
2. Capital letters, typically near the end of the alphabet, are used for tape symbols that may or may not be input symbols. However, B is generally used for the blank symbol.
3. Lower-case letters near the end of the alphabet are strings of input symbols.
4. Greek letters are strings of tape symbols.
5. Letters such as q, p, and nearby letters are states.

8.2.6 Turing Machines and Halting

There is another notion of "acceptance" that is commonly used for Turing machines: acceptance by halting. We say a TM *halts* if it enters a state q, scanning a tape symbol X, and there is no move in this situation; i.e., $\delta(q, X)$ is undefined.

Example 8.5: The Turing machine M of Example 8.4 was not designed to accept a language; rather we viewed it as computing an arithmetic function. Note, however, that M halts on all strings of 0's and 1's, since no matter what string M finds on its tape, it will eventually cancel its second group of 0's, if it can find such a group, against its first group of 0's, and thus must reach state q_6 and halt. □

We can always assume that a TM halts if it accepts. That is, without changing the language accepted, we can make $\delta(q, X)$ undefined whenever q is an accepting state. In general, without otherwise stating so:

- We assume that a TM always halts when it is in an accepting state.

Unfortunately, it is not always possible to require that a TM halts even if it does not accept. Those languages with Turing machines that do halt eventually, regardless of whether or not they accept, are called *recursive*, and we shall consider their important properties starting in Section 9.2.1. Turing machines that always halt, regardless of whether or not they accept, are a good model of an "algorithm." If an algorithm to solve a given problem exists, then

we say the problem is "decidable," so TM's that always halt figure importantly into decidability theory in Chapter 9.

8.2.7 Exercises for Section 8.2

Exercise 8.2.1: Show the ID's of the Turing machine of Fig. 8.9 if the input tape contains:

* a) 00.

b) 000111.

c) 00111.

! **Exercise 8.2.2:** Design Turing machines for the following languages:

* a) The set of strings with an equal number of 0's and 1's.

b) $\{a^n b^n c^n \mid n \geq 1\}$.

c) $\{ww^R \mid w$ is any string of 0's and 1's$\}$.

Exercise 8.2.3: Design a Turing machine that takes as input a number N and adds 1 to it in binary. To be precise, the tape initially contains a \$ followed by N in binary. The tape head is initially scanning the \$ in state q_0. Your TM should halt with N+1, in binary, on its tape, scanning the leftmost symbol of $N+1$, in state q_f. You may destroy the \$ in creating $N+1$, if necessary. For instance, $q_0\$10011 \vdash^* \$q_f10100$, and $q_0\$11111 \vdash^* q_f100000$.

a) Give the transitions of your Turing machine, and explain the purpose of each state.

b) Show the sequence of ID's of your TM when given input \$111.

*! **Exercise 8.2.4:** In this exercise we explore the equivalence between function computation and language recognition for Turing machines. For simplicity, we shall consider only functions from nonnegative integers to nonnegative integers, but the ideas of this problem apply to any computable functions. Here are the two central definitions:

- Define the *graph* of a function f to be the set of all strings of the form $[x, f(x)]$, where x is a nonnegative integer in binary, and $f(x)$ is the value of function f with argument x, also written in binary.
- A Turing machine is said to *compute* function f if, started with any nonnegative integer x on its tape, in binary, it halts (in any state) with $f(x)$, in binary, on its tape.

Answer the following, with informal, but clear constructions.

a) Show how, given a TM that computes f, you can construct a TM that accepts the graph of f as a language.

b) Show how, given a TM that accepts the graph of f, you can construct a TM that computes f.

c) A function is said to be *partial* if it may be undefined for some arguments. If we extend the ideas of this exercise to partial functions, then we do not require that the TM computing f halts if its input x is one of the integers for which $f(x)$ is not defined. Do your constructions for parts (a) and (b) work if the function f is partial? If not, explain how you could modify the construction to make it work.

Exercise 8.2.5: Consider the Turing machine

$$M = (\{q_0, q_1, q_2, q_f\}, \{0,1\}, \{0,1,B\}, \delta, q_0, B, \{q_f\})$$

Informally but clearly describe the language $L(M)$ if δ consists of the following sets of rules:

* a) $\delta(q_0,0) = (q_1,1,R)$; $\delta(q_1,1) = (q_0,0,R)$; $\delta(q_1,B) = (q_f,B,R)$.

b) $\delta(q_0,0) = (q_0,B,R)$; $\delta(q_0,1) = (q_1,B,R)$; $\delta(q_1,1) = (q_1,B,R)$; $\delta(q_1,B) = (q_f,B,R)$.

! c) $\delta(q_0,0) = (q_1,1,R)$; $\delta(q_1,1) = (q_2,0,L)$; $\delta(q_2,1) = (q_0,1,R)$; $\delta(q_1,B) = (q_f,B,R)$.

8.3 Programming Techniques for Turing Machines

Our goal is to give you a sense of how a Turing machine can be used to compute in a manner not unlike that of a conventional computer. Eventually, we want to convince you that a TM is exactly as powerful as a conventional computer. In particular, we shall learn that the Turing machine can perform the sort of calculations on other Turing machines that we saw performed in Section 8.1.2 by a program that examined other programs. This "introspective" ability of both Turing machines and computer programs is what enables us to prove problems undecidable.

To make the ability of a TM clearer, we shall present a number of examples of how we might think of the tape and finite control of the Turing machine. None of these tricks extend the basic model of the TM; they are only notational conveniences. Later, we shall use them to simulate extended Turing-machine models that have additional features — for instance, more than one tape — by the basic TM model.

8.3.1 Storage in the State

We can use the finite control not only to represent a position in the "program" of the Turing machine, but to hold a finite amount of data. Figure 8.13 suggests this technique (as well as another idea: multiple tracks). There, we see the finite control consisting of not only a "control" state q, but three data elements A, B, and C. The technique requires no extension to the TM model; we merely think of the state as a tuple. In the case of Fig. 8.13, we should think of the state as $[q, A, B, C]$. Regarding states this way allows us to describe transitions in a more systematic way, often making the strategy behind the TM program more transparent.

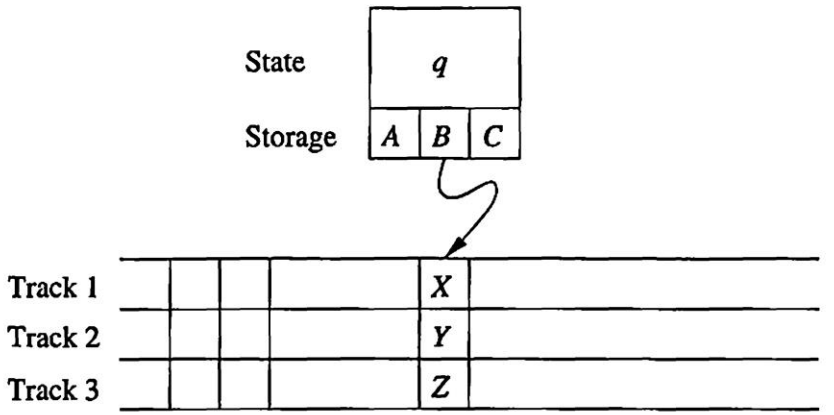

Figure 8.13: A Turing machine viewed as having finite-control storage and multiple tracks

Example 8.6: We shall design a TM

$$M = (Q, \{0,1\}, \{0,1,B\}, \delta, [q_0, B], \{[q_1, B]\})$$

that remembers in its finite control the first symbol (0 or 1) that it sees, and checks that it does not appear elsewhere on its input. Thus, M accepts the language $\mathbf{01^*+10^*}$. Accepting regular languages such as this one does not stress the ability of Turing machines, but it will serve as a simple demonstration.

The set of states Q is $\{q_0, q_1\} \times \{0,1,B\}$. That is, the states may be thought of as pairs with two components:

a) A control portion, q_0 or q_1, that remembers what the TM is doing. Control state q_0 indicates that M has not yet read its first symbol, while q_1 indicates that it *has* read the symbol, and is checking that it does not appear elsewhere, by moving right and hoping to reach a blank cell.

b) A data portion, which remembers the first symbol seen, which must be 0 or 1. The symbol B in this component means that no symbol has been read.

The transition function δ of M is as follows:

1. $\delta([q_0, B], a) = ([q_1, a], a, R)$ for $a = 0$ or $a = 1$. Initially, q_0 is the control state, and the data portion of the state is B. The symbol scanned is copied into the second component of the state, and M moves right, entering control state q_1 as it does so.

2. $\delta([q_1, a], \overline{a}) = ([q_1, a], \overline{a}, R)$ where $\overline{a}$ is the "complement" of a, that is, 0 if $a = 1$ and 1 if $a = 0$. In state q_1, M skips over each symbol 0 or 1 that is different from the one it has stored in its state, and continues moving right.

3. $\delta([q_1, a], B) = ([q_1, B], B, R)$ for $a = 0$ or $a = 1$. If M reaches the first blank, it enters the accepting state $[q_1, B]$.

Notice that M has no definition for $\delta([q_1, a], a)$ for $a = 0$ or $a = 1$. Thus, if M encounters a second occurrence of the symbol it stored initially in its finite control, it halts without having entered the accepting state. □

8.3.2 Multiple Tracks

Another useful "trick" is to think of the tape of a Turing machine as composed of several tracks. Each track can hold one symbol, and the tape alphabet of the TM consists of tuples, with one component for each "track." Thus, for instance, the cell scanned by the tape head in Fig. 8.13 contains the symbol $[X, Y, Z]$. Like the technique of storage in the finite control, using multiple tracks does not extend what the Turing machine can do. It is simply a way to view tape symbols and to imagine that they have a useful structure.

Example 8.7: A common use of multiple tracks is to treat one track as holding the data and a second track as holding a mark. We can check off each symbol as we "use" it, or we can keep track of a small number of positions within the data by marking only those positions. Examples 8.2 and 8.4 were two instances of this technique, but in neither example did we think explicitly of the tape as if it were composed of tracks. In the present example, we shall use a second track explicitly to recognize the non-context-free language

$$L_{wcw} = \{wcw \mid w \text{ is in } (0+1)^+\}$$

The Turing machine we shall design is:

$$M = (Q, \Sigma, \Gamma, \delta, [q_1, B], [B, B], \{[q_9, B]\})$$

where:

Q: The set of states is $\{q_1, q_2, \ldots, q_9\} \times \{0, 1, B\}$, that is, pairs consisting of a control state q_i and a data component: 0, 1, or blank. We again use the technique of storage in the finite control, as we allow the state to remember an input symbol 0 or 1.

Γ: The set of tape symbols is $\{B, *\} \times \{0, 1, c, B\}$. The first component, or track, can be either blank or "checked," represented by the symbols B and $*$, respectively. We use the $*$ to check off symbols of the first and second groups of 0's and 1's, eventually confirming that the string to the left of the center marker c is the same as the string to its right. The second component of the tape symbol is what we think of as the tape symbol itself. That is, we may think of the symbol $[B, X]$ as if it were the tape symbol X, for $X = 0, 1, c, B$.

Σ: The input symbols are $[B, 0]$ and $[B, 1]$, which, as just mentioned, we identify with 0 and 1, respectively.

δ: The transition function δ is defined by the following rules, in which a and b each may stand for either 0 or 1.

1. $\delta([q_1, B], [B, a]) = ([q_2, a], [*, a], R)$. In the initial state, M picks up the symbol a (which can be either 0 or 1), stores it in its finite control, goes to control state q_2, "checks off" the symbol it just scanned, and moves right. Notice that by changing the first component of the tape symbol from B to $*$, it performs the check-off.
2. $\delta([q_2, a], [B, b]) = ([q_2, a], [B, b], R)$. M moves right, looking for the symbol c. Remember that a and b can each be either 0 or 1, independently, but cannot be c.
3. $\delta([q_2, a], [B, c]) = ([q_3, a], [B, c], R)$. When M finds the c, it continues to move right, but changes to control state q_3.
4. $\delta([q_3, a], [*, b]) = ([q_3, a], [*, b], R)$. In state q_3, M continues past all checked symbols.
5. $\delta([q_3, a], [B, a]) = ([q_4, B], [*, a], L)$. If the first unchecked symbol that M finds is the same as the symbol in its finite control, it checks this symbol, because it has matched the corresponding symbol from the first block of 0's and 1's. M goes to control state q_4, dropping the symbol from its finite control, and starts moving left.
6. $\delta([q_4, B], [*, a]) = ([q_4, B], [*, a], L)$. M moves left over checked symbols.
7. $\delta([q_4, B], [B, c]) = ([q_5, B], [B, c], L)$. When M encounters the symbol c, it switches to state q_5 and continues left. In state q_5, M must make a decision, depending on whether or not the symbol immediately to the left of the c is checked or unchecked. If checked, then we have already considered the entire first block of 0's and 1's — those to the left of the c. We must make sure that all the 0's and 1's to the right of the c are also checked, and accept if no unchecked symbols remain to the right of the c. If the symbol immediately to the left of the c is unchecked, we find the leftmost unchecked symbol, pick it up, and start the cycle that began in state q_1.

8. $\delta([q_5, B], [B, a]) = ([q_6, B], [B, a], L)$. This branch covers the case where the symbol to the left of c is unchecked. M goes to state q_6 and continues left, looking for a checked symbol.

9. $\delta([q_6, B], [B, a]) = ([q_6, B], [B, a], L)$. As long as symbols are unchecked, M remains in state q_6 and proceeds left.

10. $\delta([q_6, B], [*, a]) = ([q_1, B], [*, a], R)$. When the checked symbol is found, M enters state q_1 and moves right to pick up the first unchecked symbol.

11. $\delta([q_5, B], [*, a]) = ([q_7, B], [*, a], R)$. Now, let us pick up the branch from state q_5 where we have just moved left from the c and find a checked symbol. We start moving right again, entering state q_7.

12. $\delta([q_7, B], [B, c]) = ([q_8, B], [B, c], R)$. In state q_7 we shall surely see the c. We enter state q_8 as we do so, and proceed right.

13. $\delta([q_8, B], [*, a]) = ([q_8, B], [*, a], R)$. M moves right in state q_8, skipping over any checked 0's or 1's that it finds.

14. $\delta([q_8, B], [B, B]) = ([q_9, B], [B, B], R)$. If M reaches a blank cell is state q_8 without encountering any unchecked 0 or 1, then M accepts. If M first finds an unchecked 0 or 1, then the blocks before and after the c do not match, and M halts without accepting.

□

8.3.3 Subroutines

As with programs in general, it helps to think of Turing machines as built from a collection of interacting components, or "subroutines." A Turing-machine subroutine is a set of states that perform some useful process. This set of states includes a start state and another state that temporarily has no moves, and that serves as the "return" state to pass control to whatever other set of states called the subroutine. The "call" of a subroutine occurs whenever there is a transition to its initial state. Since the TM has no mechanism for remembering a "return address," that is, a state to go to after it finishes, should our design of a TM call for one subroutine to be called from several states, we can make copies of the subroutine, using a new set of states for each copy. The "calls" are made to the start states of different copies of the subroutine, and each copy "returns" to a different state.

Example 8.8: We shall design a TM to implement the function "multiplication." That is, our TM will start with 0^m10^n1 on its tape, and will end with 0^{mn} on the tape. An outline of the strategy is:

1. The tape will, in general, have one nonblank string of the form $0^i10^n10^{kn}$ for some k.

2. In one basic step, we change a 0 in the first group to B and add n 0's to the last group, giving us a string of the form $0^{i-1}10^n10^{(k+1)n}$.

3. As a result, we copy the group of n 0's to the end m times, once each time we change a 0 in the first group to B. When the first group of 0's is completely changed to blanks, there will be mn 0's in the last group.

4. The final step is to change the leading 10^n1 to blanks, and we are done.

The heart of this algorithm is a subroutine, which we call Copy. This subroutine implements step (2) above, copying the block of n 0's to the end. More precisely, Copy converts an ID of the form $0^{m-k}1q_10^n10^{(k-1)n}$ to ID $0^{m-k}1q_50^n10^{kn}$. Figure 8.14 shows the transitions of subroutine Copy. This subroutine marks the first 0 with an X, moves right in state q_2 until it finds a blank, copies the 0 there, and moves left in state q_3 to find the marker X. It repeats this cycle until in state q_1 it finds a 1 instead of a 0. At that point, it uses state q_4 to change the X's back to 0's, and ends in state q_5.

The complete multiplication Turing machine starts in state q_0. The first thing it does is go, in several steps, from ID $q_00^m10^n$ to ID $0^{m-1}1q_10^n1$. The transitions needed are shown in the portion of Fig. 8.15 to the left of the subroutine call; these transitions involve states q_0 and q_6 only.

Then, to the right of the subroutine call in Fig. 8.15 we see states q_7 through q_{12}. The purpose of states q_7, q_8, and q_9 is to take control after Copy has just copied a block of n 0's, and is in ID $0^{m-k}1q_50^n10^{kn}$. Eventually, these states bring us to state $q_00^{m-k}10^n10^{kn}$. At that point, the cycle starts again, and Copy is called to copy the block of n 0's again.

As an exception, in state q_8 the TM may find that all m 0's have been changed to blanks (i.e., $k = m$). In that case, a transition to state q_{10} occurs. This state, with the help of state q_{11}, changes the leading 10^n1 to blanks and enters the halting state q_{12}. At this point, the TM is in ID $q_{12}0^{mn}$, and its job is done. □

8.3.4 Exercises for Section 8.3

! **Exercise 8.3.1:** Redesign your Turing machines from Exercise 8.2.2 to take advantage of the programming techniques discussed in Section 8.3.

! **Exercise 8.3.2:** A common operation in Turing-machine programs involves "shifting over." Ideally, we would like to create an extra cell at the current head position, in which we could store some character. However, we cannot edit the tape in this way. Rather, we need to move the contents of each of the cells to the right of the current head position one cell right, and then find our way back to the current head position. Show how to perform this operation. *Hint*: Leave a special symbol to mark the position to which the head must return.

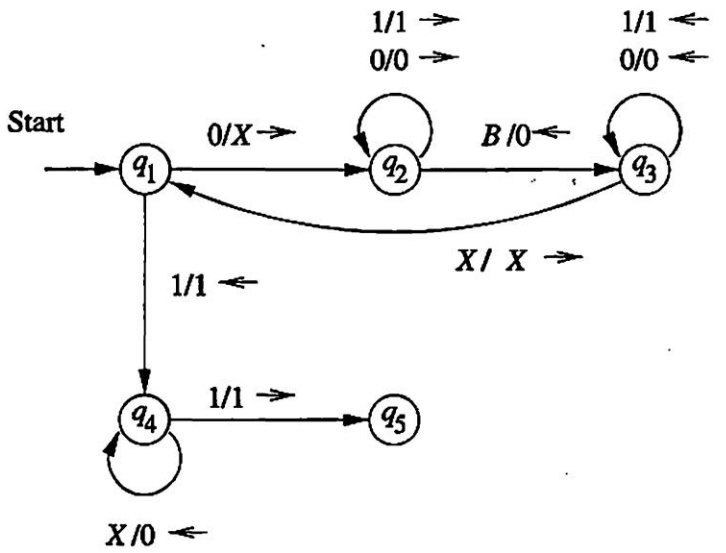

Figure 8.14: The subroutine Copy

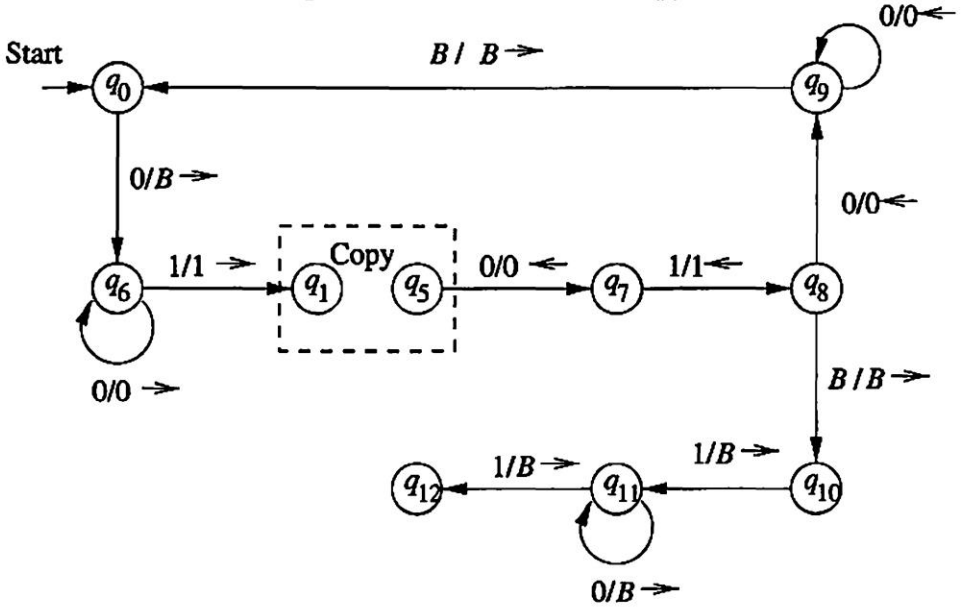

Figure 8.15: The complete multiplication program uses the subroutine Copy

* **Exercise 8.3.3:** Design a subroutine to move a TM head from its current position to the right, skipping over all 0's, until reaching a 1 or a blank. If the current position does not hold 0, then the TM should halt. You may assume that there are no tape symbols other than 0, 1, and B (blank). Then, use this subroutine to design a TM that accepts all strings of 0's and 1's that do not have two 1's in a row.

8.4 Extensions to the Basic Turing Machine

In this section we shall see certain computer models that are related to Turing machines and have the same language-recognizing power as the basic model of a TM with which we have been working. One of these, the multitape Turing machine, is important because it is much easier to see how a multitape TM can simulate real computers (or other kinds of Turing machines), compared with the single-tape model we have been studying. Yet the extra tapes add no power to the model, as far as the ability to accept languages is concerned.

We then consider the nondeterministic Turing machine, an extension of the basic model that is allowed to make any of a finite set of choices of move in a given situation. This extension also makes "programming" Turing machines easier in some circumstances, but adds no language-defining power to the basic model.

8.4.1 Multitape Turing Machines

A multitape TM is as suggested by Fig. 8.16. The device has a finite control (state), and some finite number of tapes. Each tape is divided into cells, and each cell can hold any symbol of the finite tape alphabet. As in the single-tape TM, the set of tape symbols includes a blank, and has a subset called the input symbols, of which the blank is not a member. The set of states includes an initial state and some accepting states. Initially:

1. The input, a finite sequence of input symbols, is placed on the first tape.
2. All other cells of all the tapes hold the blank.
3. The finite control is in the initial state.
4. The head of the first tape is at the left end of the input.
5. All other tape heads are at some arbitrary cell. Since tapes other than the first tape are completely blank, it does not matter where the head is placed initially; all cells of these tapes "look" the same.

A move of the multitape TM depends on the state and the symbol scanned by each of the tape heads. In one move, the multitape TM does the following:

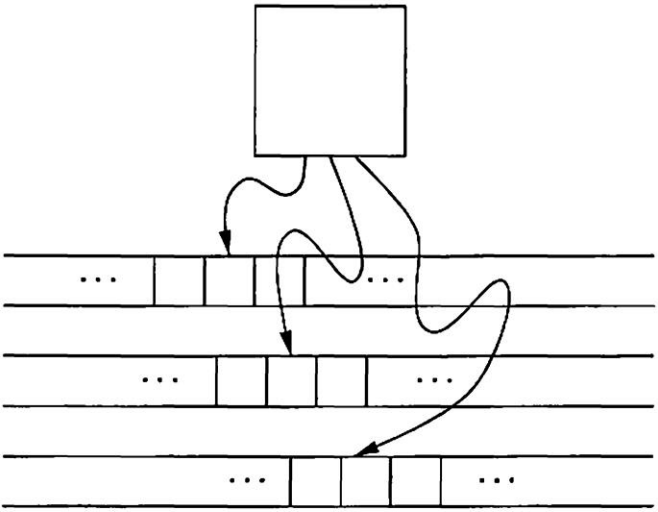

Figure 8.16: A multitape Turing machine

1. The control enters a new state, which could be the same as the previous state.

2. On each tape, a new tape symbol is written on the cell scanned. Any of these symbols may be the same as the symbol previously there.

3. Each of the tape heads makes a move, which can be either left, right, or stationary. The heads move independently, so different heads may move in different directions, and some may not move at all.

We shall not give the formal notation of transition rules, whose form is a straightforward generalization of the notation for the one-tape TM, except that directions are now indicated by a choice of L, R, or S. For the one-tape machine, we did not allow the head to remain stationary, so the S option was not present. You should be able to imagine an appropriate notation for instantaneous descriptions of the configuration of a multitape TM; we shall not give this notation formally. Multitape Turing machines, like one-tape TM's, accept by entering an accepting state.

8.4.2 Equivalence of One-Tape and Multitape TM's

Recall that the recursively enumerable languages are defined to be those accepted by a one-tape TM. Surely, multitape TM's accept all the recursively enumerable languages, since a one-tape TM *is* a multitape TM. However, are there languages that are not recursively enumerable, yet are accepted by multitape TM's? The answer is "no," and we prove this fact by showing how to simulate a multitape TM by a one-tape TM.

Theorem 8.9: Every language accepted by a multitape TM is recursively enumerable.

PROOF: The proof is suggested by Fig. 8.17. Suppose language L is accepted by a k-tape TM M. We simulate M with a one-tape TM N whose tape we think of as having $2k$ tracks. Half these tracks hold the tapes of M, and the other half of the tracks each hold only a single marker that indicates where the head for the corresponding tape of M is currently located. Figure 8.17 assumes $k = 2$. The second and fourth tracks hold the contents of the first and second tapes of M, track 1 holds the position of the head of tape 1, and track 3 holds the position of the second tape head.

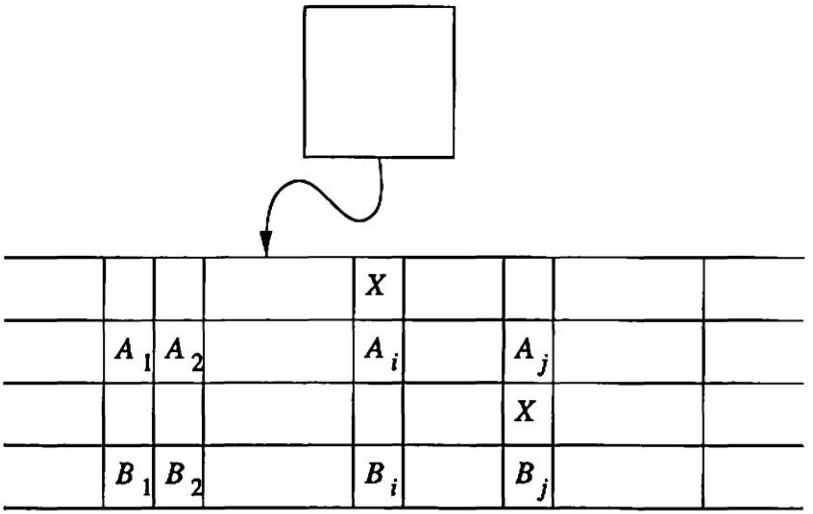

Figure 8.17: Simulation of a two-tape Turing machine by a one-tape Turing machine

To simulate a move of M, N's head must visit the k head markers. So that N not get lost, it must remember how many head markers are to its left at all times; that count is stored as a component of N's finite control. After visiting each head marker and storing the scanned symbol in a component of its finite control, N knows what tape symbols are being scanned by each of M's heads. N also knows the state of M, which it stores in N's own finite control. Thus, N knows what move M will make.

N now revisits each of the head markers on its tape, changes the symbol in the track representing the corresponding tapes of M, and moves the head markers left or right, if necessary. Finally, N changes the state of M as recorded in its own finite control. At this point, N has simulated one move of M.

We select as N's accepting states all those states that record M's state as one of the accepting states of M. Thus, whenever the simulated M accepts, N also accepts, and N does not accept otherwise. □

A Reminder About Finiteness

A common fallacy is to confuse a value that is finite at any time with a set of values that is finite. The many-tapes-to-one construction may help us appreciate the difference. In that construction, we used tracks on the tape to record the positions of the tape heads. Why could we not store these positions as integers in the finite control? Carelessly, one could argue that after n moves, the TM can have tape head positions that must be within n positions of original head positions, and so the head only has to store integers up to n.

The problem is that, while the positions are finite at any time, the complete set of positions possible at any time is infinite. If the state is to represent any head position, then there must be a data component of the state that has any integer as value. This component forces the set of states to be infinite, even if only a finite number of them can be used at any finite time. The definition of a Turing machine requires that the *set* of states be finite. Thus, it is not permissible to store a tape-head position in the finite control.

8.4.3 Running Time and the Many-Tapes-to-One Construction

Let us now introduce a concept that will become quite important later: the "time complexity" or "running time" of a Turing machine. We say the *running time* of TM M on input w is the number of steps that M makes before halting. If M doesn't halt on w, then the running time of M on w is infinite. The *time complexity* of TM M is the function $T(n)$ that is the maximum, over all inputs w of length n, of the running time of M on w. For Turing machines that do not halt on all inputs, $T(n)$ may be infinite for some or even all n. However, we shall pay special attention to TM's that do halt on all inputs, and in particular, those that have a polynomial time complexity $T(n)$; Section 10.1 initiates this study.

The construction of Theorem 8.9 seems clumsy. In fact, the constructed one-tape TM may take much more running time than the multitape TM. However, the amounts of time taken by the two Turing machines are commensurate in a weak sense: the one-tape TM takes time that is no more than the square of the time taken by the other. While "squaring" is not a very strong guarantee, it does preserve polynomial running time. We shall see in Chapter 10 that:

a) The difference between polynomial time and higher growth rates in running time is really the divide between what we can solve by computer and what is in practice not solvable.

b) Despite extensive research, the running time needed to solve many prob-

lems has not been resolved closer than to within some polynomial. Thus, the question of whether we are using a one-tape or multitape TM to solve the problem is not crucial when we examine the running time needed to solve a particular problem.

The argument that the running times of the one-tape and multitape TM's are within a square of each other is as follows.

Theorem 8.10: The time taken by the one-tape TM N of Theorem 8.9 to simulate n moves of the k-tape TM M is $O(n^2)$.

PROOF: After n moves of M, the tape head markers cannot have separated by more than $2n$ cells. Thus, if N starts at the leftmost marker, it has to move no more than $2n$ cells right, to find all the head markers. It can then make an excursion leftward, changing the contents of the simulated tapes of M, and moving head markers left or right as needed. Doing so requires no more than $2n$ moves left, plus at most $2k$ moves to reverse direction and write a marker X in the cell to the right (in the case that a tape head of M moves right).

Thus, the number of moves by N needed to simulate one of the first n moves is no more than $4n + 2k$. Since k is a constant, independent of the number of moves simulated, this number of moves is $O(n)$. To simulate n moves requires no more than n times this amount, or $O(n^2)$. □

8.4.4 Nondeterministic Turing Machines

A *nondeterministic* Turing machine (*NTM*) differs from the deterministic variety we have been studying by having a transition function δ such that for each state q and tape symbol X, $\delta(q, X)$ is a set of triples

$$\{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\}$$

where k is any finite integer. The NTM can choose, at each step, any of the triples to be the next move. It cannot, however, pick a state from one, a tape symbol from another, and the direction from yet another.

The language accepted by an NTM M is defined in the expected manner, in analogy with the other nondeterministic devices, such as NFA's and PDA's, that we have studied. That is, M accepts an input w if there is any sequence of choices of move that leads from the initial ID with w as input, to an ID with an accepting state. The existence of other choices that do *not* lead to an accepting state is irrelevant, as it is for the NFA or PDA.

The NTM's accept no languages not accepted by a deterministic TM (or *DTM* if we need to emphasize that it is deterministic). The proof involves showing that for every NTM M_N, we can construct a DTM M_D that explores the ID's that M_N can reach by any sequence of its choices. If M_D finds one that has an accepting state, then M_D enters an accepting state of its own. M_D must be systematic, putting new ID's on a queue, rather than a stack, so that after some finite time M_D has simulated all sequences of up to k moves of M_N, for $k = 1, 2, \ldots$.

Theorem 8.11: If M_N is a nondeterministic Turing machine, then there is a deterministic Turing machine M_D such that $L(M_N) = L(M_D)$.

PROOF: M_D will be designed as a multitape TM, sketched in Fig. 8.18. The first tape of M_D holds a sequence of ID's of M_N, including the state of M_N. One ID of M_N is marked as the "current" ID, whose successor ID's are in the process of being discovered. In Fig. 8.18, the third ID is marked by an x along with the inter-ID separator, which is the $*$. All ID's to the left of the current one have been explored and can be ignored subsequently.

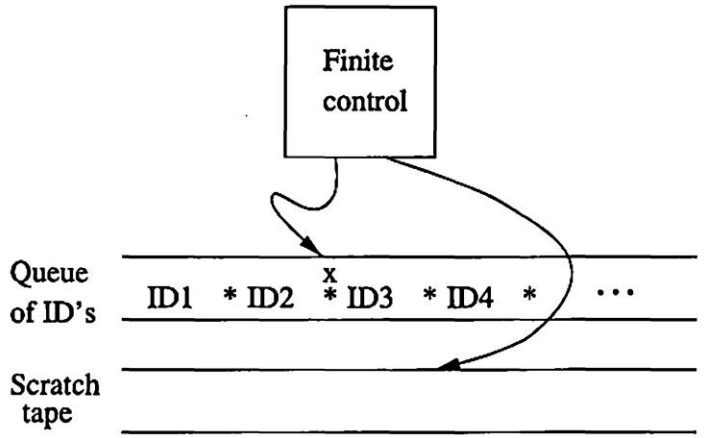

Figure 8.18: Simulation of an NTM by a DTM

To process the current ID, M_D does the following:

1. M_D examines the state and scanned symbol of the current ID. Built into the finite control of M_D is the knowledge of what choices of move M_N has for each state and symbol. If the state in the current ID is accepting, then M_D accepts and simulates M_N no further.

2. However, if the state is not accepting, and the state-symbol combination has k moves, then M_D uses its second tape to copy the ID and then make k copies of that ID at the end of the sequence of ID's on tape 1.

3. M_D modifies each of those k ID's according to a different one of the k choices of move that M_N has from its current ID.

4. M_D returns to the marked, current ID, erases the mark, and moves the mark to the next ID to the right. The cycle then repeats with step (1).

It should be clear that the simulation is accurate, in the sense that M_D will only accept if it finds that M_N can enter an accepting ID. However, we need to confirm that if M_N enters an accepting ID after a sequence of n of its own moves, then M_D will eventually make that ID the current ID and will accept.

Suppose that m is the maximum number of choices M_N has in any configuration. Then there is one initial ID of M_N, at most m ID's that M_N can reach after one move, at most m^2 ID's M_N can reach after two moves, and so on. Thus, after n moves, M_N can reach at most $1 + m + m^2 + \cdots + m^n$ ID's. This number is at most nm^n ID's.

The order in which M_D explores ID's of M_N is "breadth first"; that is, it explores all ID's reachable by 0 moves (i.e., the initial ID), then all ID's reachable by one move, then those reachable by two moves, and so on. In particular, M_D will make current, and consider the successors of, all ID's reachable by up to n moves before considering any ID's that are only reachable by more than n moves.

As a consequence, the accepting ID of M_N will be considered by M_D among the first nm^n ID's that it considers. We only care that M_D considers this ID in some finite time, and this bound is sufficient to assure us that the accepting ID is considered eventually. Thus, if M_N accepts, then so does M_D. Since we already observed that if M_D accepts it does so only because M_N accepts, we conclude that $L(M_N) = L(M_D)$. □

Notice that the constructed deterministic TM may take exponentially more time than the nondeterministic TM. It is unknown whether or not this exponential slowdown is necessary. In fact, Chapter 10 is devoted to this question and the consequences of someone discovering a better way to simulate NTM's deterministically.

8.4.5 Exercises for Section 8.4

Exercise 8.4.1: Informally but clearly describe multitape Turing machines that accept each of the languages of Exercise 8.2.2. Try to make each of your Turing machines run in time proportional to the input length.

Exercise 8.4.2: Here is the transition function of a nondeterministic TM $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_2\})$:

δ	0	1	B
q_0	$\{(q_0, 1, R)\}$	$\{(q_1, 0, R)\}$	$\emptyset$
q_1	$\{(q_1, 0, R), (q_0, 0, L)\}$	$\{(q_1, 1, R), (q_0, 1, L)\}$	$\{(q_2, B, R)\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$

Show the ID's reachable from the initial ID if the input is:

* a) 01.

b) 011.

! **Exercise 8.4.3:** Informally but clearly describe nondeterministic Turing machines — multitape if you like — that accept the following languages. Try to

take advantage of nondeterminism to avoid iteration and save time in the nondeterministic sense. That is, prefer to have your NTM branch a lot, while each branch is short.

* a) The language of all strings of 0's and 1's that have some string of length 100 that repeats, not necessarily consecutively. Formally, this language is the set of strings of 0's and 1's of the form $wxyxz$, where $|x| = 100$, and w, y, and z are of arbitrary length.

b) The language of all strings of the form $w_1\#w_2\#\cdots\#w_n$, for any n, such that each w_i is a string of 0's and 1's, and for some j, w_j is the integer j in binary.

c) The language of all strings of the same form as (b), but for at least two values of j, we have w_j equal to j in binary.

! **Exercise 8.4.4:** Consider the nondeterministic Turing machine

$$M = (\{q_0, q_1, q_2, q_f\}, \{0,1\}, \{0,1,B\}, \delta, q_0, B, \{q_f\})$$

Informally but clearly describe the language $L(M)$ if δ consists of the following sets of rules: $\delta(q_0, 0) = \{(q_0, 1, R), (q_1, 1, R)\}$; $\delta(q_1, 1) = \{(q_2, 0, L)\}$; $\delta(q_2, 1) = \{(q_0, 1, R)\}$; $\delta(q_1, B) = \{(q_f, B, R)\}$.

* **Exercise 8.4.5:** Consider a nondeterministic TM whose tape is infinite in both directions. At some time, the tape is completely blank, except for one cell, which holds the symbol \$. The head is currently at some blank cell, and the state is q.

a) Write transitions that will enable the NTM to enter state p, scanning the \$.

! b) Suppose the TM were deterministic instead. How would you enable it to find the \$ and enter state p?

Exercise 8.4.6: Design the following 2-tape TM to accept the language of all strings of 0's and 1's with an equal number of each. The first tape contains the input, and is scanned from left to right. The second tape is used to store the excess of 0's over 1's, or vice-versa, in the part of the input seen so far. Specify the states, transitions, and the intuitive purpose of each state.

Exercise 8.4.7: In this exercises, we shall implement a stack using a special 3-tape TM.

1. The first tape will be used only to hold and read the input. The input alphabet consists of the symbol $\uparrow$, which we shall interpret as "pop the stack," and the symbols a and b, which are interpreted as "push an a (respectively b) onto the stack."

2. The second tape is used to store the stack.

3. The third tape is the output tape. Every time a symbol is popped from the stack, it must be written on the output tape, following all previously written symbols.

The Turing machine is required to start with an empty stack and implement the sequence of push and pop operations, as specified on the input, reading from left to right. If the input causes the TM to try to pop and empty stack, then it must halt in a special error state q_e. If the entire input leaves the stack empty at the end, then the input is accepted by going to the final state q_f. Describe the transition function of the TM informally but clearly. Also, give a summary of the purpose of each state you use.

Exercise 8.4.8: In Fig. 8.17 we saw an example of the general simulation of a k-tape TM by a one-tape TM.

* a) Suppose this technique is used to simulate a 5-tape TM that had a tape alphabet of seven symbols. How many tape symbols would the one-tape TM have?

* b) An alternative way to simulate k tapes by one is to use a $(k+1)$st track to hold the head positions of all k tapes, while the first k tracks simulate the k tapes in the obvious manner. Note that in the $(k+1)$st track, we must be careful to distinguish among the tape heads and to allow for the possibility that two or more heads are at the same cell. Does this method reduce the number of tape symbols needed for the one-tape TM?

c) Another way to simulate k tapes by 1 is to avoid storing the head positions altogether. Rather, a $(k+1)$st track is used only to mark one cell of the tape. At all times, each simulated tape is positioned on its track so the head is at the marked cell. If the k-tape TM moves the head of tape i, then the simulating one-tape TM slides the entire nonblank contents of the ith track one cell in the opposite direction, so the marked cell continues to hold the cell scanned by the ith tape head of the k-tape TM. Does this method help reduce the number of tape symbols of the one-tape TM? Does it have any drawbacks compared with the other methods discussed?

! **Exercise 8.4.9:** A *k-head* Turing machine has k heads reading cells of one tape. A move of this TM depends on the state and on the symbol scanned by each head. In one move, the TM can change state, write a new symbol on the cell scanned by each head, and can move each head left, right, or keep it stationary. Since several heads may be scanning the same cell, we assume the heads are numbered 1 through k, and the symbol written by the highest numbered head scanning a given cell is the one that actually gets written there. Prove that the languages accepted by k-head Turing machines are the same as those accepted by ordinary TM's.

- Does this Turing machine accept this input?

Then, we exploit this undecidability result to exhibit a number of other undecidable problems. For instance, we show that all nontrivial problems about the language accepted by a Turing machine are undecidable, as are a number of problems that have nothing at all to do with Turing machines, programs, or computers.

9.1 A Language That Is Not Recursively Enumerable

Recall that a language L is *recursively enumerable* (abbreviated RE) if $L = L(M)$ for some TM M. Also, we shall in Section 9.2 introduce "recursive" or "decidable" languages that are not only recursively enumerable, but are accepted by a TM that always halts, regardless of whether or not it accepts.

Our long-range goal is to prove undecidable the language consisting of pairs (M, w) such that:

1. M is a Turing machine (suitably coded, in binary) with input alphabet $\{0, 1\}$,
2. w is a string of 0's and 1's, and
3. M accepts input w.

If this problem with inputs restricted to the binary alphabet is undecidable, then surely the more general problem, where TM's may have any alphabet, is undecidable.

Our first step is to set this question up as a true question about membership in a particular language. Thus, we must give a coding for Turing machines that uses only 0's and 1's, regardless of how many states the TM has. Once we have this coding, we can treat any binary string as if it were a Turing machine. If the string is not a well-formed representation of some TM, we may think of it as representing a TM with no moves. Thus, we may think of every binary string as some TM.

An intermediate goal, and the subject of this section, involves the language L_d, the "diagonalization language," which consists of all those strings w such that the TM represented by w does not accept the input w. We shall show that L_d has no Turing machine at all that accepts it. Remember that showing there is no Turing machine at all for a language is showing something stronger than that the language is undecidable (i.e., that it has no algorithm, or TM that always halts).

The language L_d plays a role analogous to the hypothetical program H_2 of Section 8.1.2, which prints `hello, world` whenever its input does *not* print `hello, world` when given itself as input. More precisely, just as H_2 cannot

exist because its response when given itself as input is paradoxical, L_d cannot be accepted by a Turing machine, because if it were, then that Turing machine would have to disagree with itself when given a code for itself as input.

9.1.1 Enumerating the Binary Strings

In what follows, we shall need to assign integers to all the binary strings so that each string corresponds to one integer, and each integer corresponds to one string. If w is a binary string, treat $1w$ as a binary integer i. Then we shall call w the ith string. That is, ϵ is the first string, 0 is the second, 1 the third, 00 the fourth, 01 the fifth, and so on. Equivalently, strings are ordered by length, and strings of equal length are ordered lexicographically. Hereafter, we shall refer to the ith string as w_i.

9.1.2 Codes for Turing Machines

Our next goal is to devise a binary code for Turing machines so that each TM with input alphabet $\{0,1\}$ may be thought of as a binary string. Since we just saw how to enumerate the binary strings, we shall then have an identification of the Turing machines with the integers, and we can talk about "the ith Turing machine, M_i." To represent a TM $M = (Q, \{0,1\}, \Gamma, \delta, q_1, B, F)$ as a binary string, we must first assign integers to the states, tape symbols, and directions L and R.

- We shall assume the states are $q_1, q_2, \ldots, q_r$ for some r. The start state will always be q_1, and q_2 will be the only accepting state. Note that, since we may assume the TM halts whenever it enters an accepting state, there is never any need for more than one accepting state.

- We shall assume the tape symbols are $X_1, X_2, \ldots, X_s$ for some s. X_1 always will be the symbol 0, X_2 will be 1, and X_3 will be B, the blank. However, other tape symbols can be assigned to the remaining integers arbitrarily.

- We shall refer to direction L as D_1 and direction R as D_2.

Since each TM M can have integers assigned to its states and tape symbols in many different orders, there will be more than one encoding of the typical TM. However, that fact is unimportant in what follows, since we shall show that no encoding can represent a TM M such that $L(M) = L_d$.

Once we have established an integer to represent each state, symbol, and direction, we can encode the transition function δ. Suppose one transition rule is $\delta(q_i, X_j) = (q_k, X_l, D_m)$, for some integers i, j, k, l, and m. We shall code this rule by the string $0^i10^j10^k10^l10^m$. Notice that, since all of i, j, k, l, and m are at least one, there are no occurrences of two or more consecutive 1's within the code for a single transition.

A code for the entire TM M consists of all the codes for the transitions, in some order, separated by pairs of 1's:

$$C_1 11 C_2 11 \cdots C_{n-1} 11 C_n$$

where each of the C's is the code for one transition of M.

Example 9.1: Let the TM in question be

$$M = (\{q_1, q_2, q_3\}, \{0,1\}, \{0,1,B\}, \delta, q_1, B, \{q_2\})$$

where δ consists of the rules:

$$\begin{aligned}
\delta(q_1, 1) &= (q_3, 0, R) \\
\delta(q_3, 0) &= (q_1, 1, R) \\
\delta(q_3, 1) &= (q_2, 0, R) \\
\delta(q_3, B) &= (q_3, 1, L)
\end{aligned}$$

The codes for each of these rules, respectively, are:

0100100010100
0001010100100
00010010010100
0001000100010010

For example, the first rule can be written as $\delta(q_1, X_2) = (q_3, X_1, D_2)$, since $1 = X_2$, $0 = X_1$, and $R = D_2$. Thus, its code is $0^1 10^2 10^3 10^1 10^2$, as was indicated above. A code for M is:

0100100010100110001010100100110001001001010011000100010010010

Note that there are many other possible codes for M. In particular, the codes for the four transitions may be listed in any of 4! orders, giving us 24 codes for M. □

In Section 9.2.3, we shall have need to code pairs consisting of a TM and a string, (M, w). For this pair we use the code for M followed by 111, followed by w. Note that, since no valid code for a TM contains three 1's in a row, we can be sure that the first occurrence of 111 separates the code for M from w. For instance, if M were the TM of Example 9.1, and w were 1011, then the code for (M, w) would be the string shown at the end of Example 9.1 followed by 1111011.

9.1.3 The Diagonalization Language

In Section 9.1.2 we coded Turing machines so there is now a concrete notion of M_i, the "ith Turing machine": that TM M whose code is w_i, the ith binary string. Many integers do not correspond to any TM at all. For instance, 11001

does not begin with 0, and 0010111010010100 is not valid because it has three consecutive 1's. If w_i is not a valid TM code, we shall take M_i to be the TM with one state and no transitions. That is, for these values of i, M_i is a Turing machine that immediately halts on any input. Thus, $L(M_i)$ is $\emptyset$ if w_i fails to be a valid TM code.

Now, we can make a vital definition.

- The language L_d, the *diagonalization language*, is the set of strings w_i such that w_i is not in $L(M_i)$.

That is, L_d consists of all strings w such that the TM M whose code is w does not accept when given w as input.

The reason L_d is called a "diagonalization" language can be seen if we consider Fig. 9.1. This table tells for all i and j, whether the TM M_i accepts input string w_j; 1 means "yes it does" and 0 means "no it doesn't."[1] We may think of the ith row as the *characteristic vector* for the language $L(M_i)$; that is, the 1's in this row indicate the strings that are members of this language.

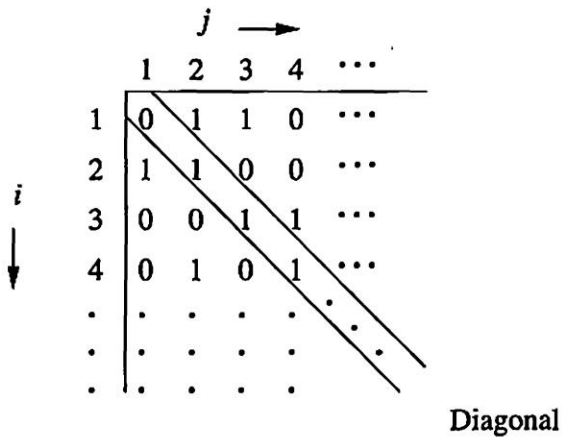

Figure 9.1: The table that represents acceptance of strings by Turing machines

The diagonal values tell whether M_i accepts w_i. To construct L_d, we complement the diagonal. For instance, if Fig. 9.1 were the correct table, then the complemented diagonal would begin $1, 0, 0, 0, \ldots$. Thus, L_d would contain $w_1 = \epsilon$, not contain w_2 through w_4, which are 0, 1, and 00, and so on.

The trick of complementing the diagonal to construct the characteristic vector of a language that cannot be the language that appears in any row, is called *diagonalization*. It works because the complement of the diagonal is itself a characteristic vector describing membership in some language, namely

[1] You should note that the actual table does not look anything like the one suggested by the figure. Since all low integers fail to represent a valid TM code, and thus represent the trivial TM that makes no moves, the top rows of the table are in fact solid 0's.

L_d. This characteristic vector disagrees in some column with every row of the table suggested by Fig. 9.1. Thus, the complement of the diagonal cannot be the characteristic vector of any Turing machine.

9.1.4 Proof that L_d is not Recursively Enumerable

Following the above intuition about characteristic vectors and the diagonal, we shall now prove formally a fundamental result about Turing machines: there is no Turing machine that accepts the language L_d.

Theorem 9.2: L_d is not a recursively enumerable language. That is, there is no Turing machine that accepts L_d.

PROOF: Suppose L_d were $L(M)$ for some TM M. Since L_d is a language over alphabet $\{0,1\}$, M would be in the list of Turing machines we have constructed, since it includes all TM's with input alphabet $\{0,1\}$. Thus, there is at least one code for M, say i; that is, $M = M_i$.

Now, ask if w_i is in L_d.

- If w_i is in L_d, then M_i accepts w_i. But then, by definition of L_d, w_i is not in L_d, because L_d contains only those w_j such that M_j does *not* accept w_j.
- Similarly, if w_i is not in L_d, then M_i does not accept w_i, Thus, by definition of L_d, w_i *is* in L_d.

Since w_i can neither be in L_d nor fail to be in L_d, we conclude that there is a contradiction of our assumption that M exists. That is, L_d is not a recursively enumerable language. □

9.1.5 Exercises for Section 9.1

Exercise 9.1.1: What strings are:

* a) w_{37}?

b) w_{100}?

Exercise 9.1.2: Write one of the possible codes for the Turing machine of Fig. 8.9.

! **Exercise 9.1.3:** Here are two definitions of languages that are similar to the definition of L_d, yet different from that language. For each, show that the language is not accepted by a Turing machine, using a diagonalization-type argument. Note that you cannot develop an argument based on the diagonal itself, but must find another infinite sequence of points in the matrix suggested by Fig. 9.1.

* a) The set of all w_i such that w_i is not accepted by M_{2i}.

b) The set of all w_i such that w_{2i} is not accepted by M_i.

! **Exercise 9.1.4:** We have considered only Turing machines that have input alphabet $\{0,1\}$. Suppose that we wanted to assign an integer to all Turing machines, regardless of their input alphabet. That is not quite possible because, while the names of the states or noninput tape symbols are arbitrary, the particular input symbols matter. For instance, the languages $\{0^n1^n \mid n \geq 1\}$ and $\{a^nb^n \mid n \geq 1\}$, while similar in some sense, are *not* the same language, and they are accepted by different TM's. However, suppose that we have an infinite set of symbols, $\{a_1, a_2, \ldots\}$ from which all TM input alphabets are chosen. Show how we could assign an integer to all TM's that had a finite subset of these symbols as its input alphabet.

9.2 An Undecidable Problem That is RE

Now, we have seen a problem — the diagonalization language L_d — that has no Turing machine to accept it. Our next goal is to refine the structure of the recursively enumerable (RE) languages (those that are accepted by TM's) into two classes. One class, which corresponds to what we commonly think of as an algorithm, has a TM that not only recognizes the language, but it tells us when it has decided the input string is not in the language. Such a Turing machine always halts eventually, regardless of whether or not it reaches an accepting state.

The second class of languages consists of those RE languages that are not accepted by any Turing machine with the guarantee of halting. These languages are accepted in an inconvenient way: if the input is in the language, we'll eventually know that, but if the input is not in the language, then the Turing machine may run forever, and we shall never be sure the input won't be accepted eventually. An example of this type of language, as we shall see, is the set of coded pairs (M, w) such that TM M accepts input w.

9.2.1 Recursive Languages

We call a language L *recursive* if $L = L(M)$ for some Turing machine M such that:

1. If w is in L, then M accepts (and therefore halts).
2. If w is not in L, then M eventually halts, although it never enters an accepting state.

A TM of this type corresponds to our informal notion of an "algorithm," a well-defined sequence of steps that always finishes and produces an answer. If we think of the language L as a "problem," as will be the case frequently, then problem L is called *decidable* if it is a recursive language, and it is called *undecidable* if it is not a recursive language.

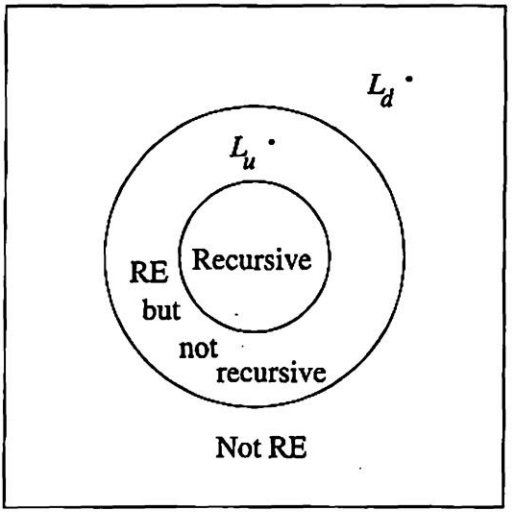

Figure 9.2: Relationship between the recursive, RE, and non-RE languages

The existence or nonexistence of an algorithm to solve a problem is often of more importance than the existence of some TM to solve the problem. As mentioned above, the Turing machines that are not guaranteed to halt may not give us enough information ever to conclude that a string is not in the language, so there is a sense in which they have not "solved the problem." Thus, dividing problems or languages between the decidable — those that are solved by an algorithm — and those that are undecidable is often more important than the division between the recursively enumerable languages (those that have TM's of some sort) and the non-recursively-enumerable languages (which have no TM at all). Figure 9.2 suggests the relationship among three classes of languages:

1. The recursive languages.
2. The languages that are recursively enumerable but not recursive.
3. The non-recursively-enumerable (*non-RE*) languages.

We have positioned the non-RE language L_d properly, and we also show the language L_u, or "universal language," that we shall prove shortly not to be recursive, although it is RE.

9.2.2 Complements of Recursive and RE languages

A powerful tool in proving languages to belong in the second ring of Fig. 9.2 (i.e., to be RE, but not recursive) is consideration of the complement of the language. We shall show that the recursive languages are closed under complementation. Thus, if a language L is RE, but $\overline{L}$, the complement of L, is not RE, then we

Why "Recursive"?

Programmers today are familiar with recursive functions. Yet these recursive functions don't seem to have anything to do with Turing machines that always halt. Worse, the opposite — nonrecursive or undecidable — refers to languages that cannot be recognized by any algorithm, yet we are accustomed to thinking of "nonrecursive" as referring to computations that are so simple there is no need for recursive function calls.

The term "recursive," as a synonym for "decidable," goes back to Mathematics as it existed prior to computers. Then, formalisms for computation based on recursion (but not iteration or loops) were commonly used as a notion of computation. These notations, which we shall not cover here, had some of the flavor of computation in functional programming languages such as LISP or ML. In that sense, to say a problem was "recursive" had the positive sense of "it is sufficiently simple that I can write a recursive function to solve it, and the function always finishes." That is exactly the meaning carried by the term today, in connection with Turing machines.

The term "recursively enumerable" harks back to the same family of concepts. A function could list all the members of a language, in some order; that is, it could "enumerate" them. The languages that can have their members listed in some order are the same as the languages that are accepted by some TM, although that TM might run forever on inputs that it does not accept.

know L cannot be recursive. For if L were recursive, then $\overline{L}$ would also be recursive and thus surely RE. We now prove this important closure property of the recursive languages.

Theorem 9.3: If L is a recursive language, so is $\overline{L}$.

PROOF: Let $L = L(M)$ for some TM M that always halts. We construct a TM $\overline{M}$ such that $\overline{L} = L(\overline{M})$ by the construction suggested in Fig. 9.3. That is, $\overline{M}$ behaves just like M. However, M is modified as follows to create $\overline{M}$:

1. The accepting states of M are made nonaccepting states of $\overline{M}$ with no transitions; i.e., in these states $\overline{M}$ will halt without accepting.

2. $\overline{M}$ has a new accepting state r; there are no transitions from r.

3. For each combination of a nonaccepting state of M and a tape symbol of M such that M has no transition (i.e., M halts without accepting), add a transition to the accepting state r.

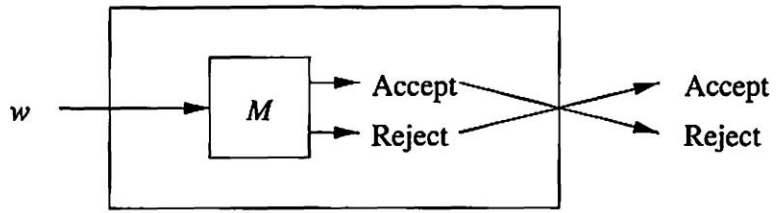

Figure 9.3: Construction of a TM accepting the complement of a recursive language

Since M is guaranteed to halt, we know that $\overline{M}$ is also guaranteed to halt. Moreover, $\overline{M}$ accepts exactly those strings that M does not accept. Thus $\overline{M}$ accepts $\overline{L}$. □

There is another important fact about complements of languages that further restricts where in the diagram of Fig. 9.2 a language and its complement can fall. We state this restriction in the next theorem.

Theorem 9.4: If both a language L and its complement are RE, then L is recursive. Note that then by Theorem 9.3, $\overline{L}$ is recursive as well.

PROOF: The proof is suggested by Fig. 9.4. Let $L = L(M_1)$ and $\overline{L} = L(M_2)$. Both M_1 and M_2 are simulated in parallel by a TM M. We can make M a two-tape TM, and then convert it to a one-tape TM, to make the simulation easy and obvious. One tape of M simulates the tape of M_1, while the other tape of M simulates the tape of M_2. The states of M_1 and M_2 are each components of the state of M.

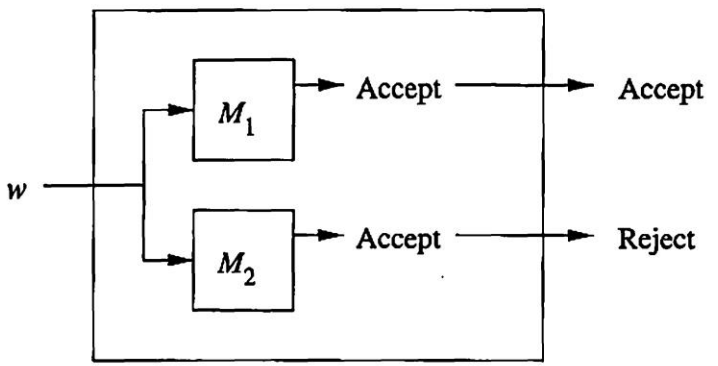

Figure 9.4: Simulation of two TM's accepting a language and its complement

If input w to M is in L, then M_1 will eventually accept. If so, M accepts and halts. If w is not in L, then it is in $\overline{L}$, so M_2 will eventually accept. When M_2 accepts, M halts without accepting. Thus, on all inputs, M halts, and

$L(M)$ is exactly L. Since M always halts, and $L(M) = L$, we conclude that L is recursive. □

We may summarize Theorems 9.3 and 9.4 as follows. Of the nine possible ways to place a language L and its complement $\overline{L}$ in the diagram of Fig. 9.2, only the following four are possible:

1. Both L and $\overline{L}$ are recursive; i.e., both are in the inner ring.
2. Neither L nor $\overline{L}$ is RE; i.e., both are in the outer ring.
3. L is RE but not recursive, and $\overline{L}$ is not RE; i.e., one is in the middle ring and the other is in the outer ring.
4. $\overline{L}$ is RE but not recursive, and L is not RE; i.e., the same as (3), but with L and $\overline{L}$ swapped.

In proof of the above, Theorem 9.3 eliminates the possibility that one language (L or $\overline{L}$) is recursive and the other is in either of the other two classes. Theorem 9.4 eliminates the possibility that both are RE but not recursive.

Example 9.5: As an example, consider the language L_d, which we know is not RE. Thus, $\overline{L_d}$ could not be recursive. It is, however, possible that $\overline{L_d}$ could be either non-RE or RE-but-not-recursive. It is in fact the latter.

$\overline{L_d}$ is the set of strings w_i such that M_i accepts w_i. This language is similar to the universal language L_u consisting of all pairs (M, w) such that M accepts w, which we shall show in Section 9.2.3 is RE. The same argument can be used to show $\overline{L_d}$ is RE. □

9.2.3 The Universal Language

We already discussed informally in Section 8.6.2 how a Turing machine could be used to simulate a computer that had been loaded with an arbitrary program. That is to say, a single TM can be used as a "stored program computer," taking its program as well as its data from one or more tapes on which input is placed. In this section, we shall repeat the idea with the additional formality that comes with talking about the Turing machine as our representation of a stored program.

We define L_u, the *universal language*, to be the set of binary strings that encode, in the notation of Section 9.1.2, a pair (M, w), where M is a TM with the binary input alphabet, and w is a string in $(0+1)^*$, such that w is in $L(M)$. That is, L_u is the set of strings representing a TM and an input accepted by that TM. We shall show that there is a TM U, often called the *universal Turing machine*, such that $L_u = L(U)$. Since the input to U is a binary string, U is in fact some M_j in the list of binary-input Turing machines we developed in Section 9.1.2.

It is easiest to describe U as a multitape Turing machine, in the spirit of Fig. 8.22. In the case of U, the transitions of M are stored initially on the first tape, along with the string w. A second tape will be used to hold the simulated tape of M, using the same format as for the code of M. That is, tape symbol X_i of M will be represented by 0^i, and tape symbols will be separated by single 1's. The third tape of U holds the state of M, with state q_i represented by i 0's. A sketch of U is in Fig. 9.5.

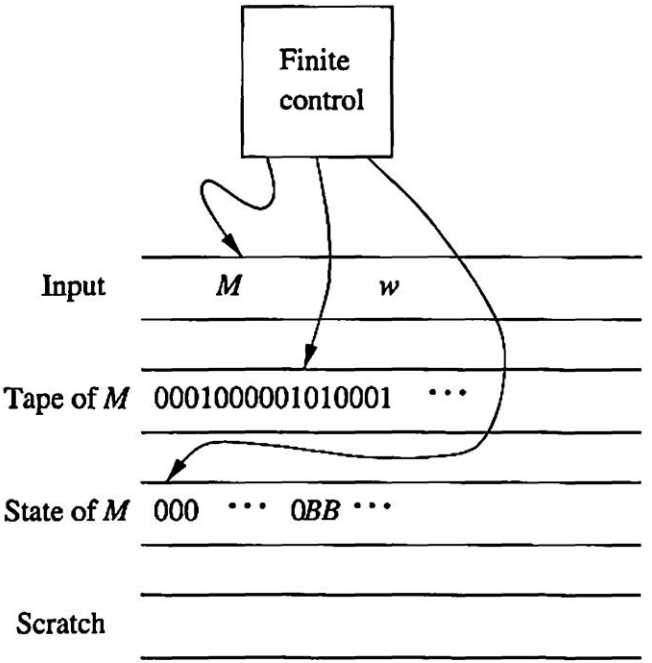

Figure 9.5: Organization of a universal Turing machine

The operation of U can be summarized as follows:

1. Examine the input to make sure that the code for M is a legitimate code for some TM. If not, U halts without accepting. Since invalid codes are assumed to represent the TM with no moves, and such a TM accepts no inputs, this action is correct.

2. Initialize the second tape to contain the input w, in its encoded form. That is, for each 0 of w, place 10 on the second tape, and for each 1 of w, place 100 there. Note that the blanks on the simulated tape of M, which are represented by 1000, will not actually appear on that tape; all cells beyond those used for w will hold the blank of U. However, U knows that, should it look for a simulated symbol of M and find its own blank, it must replace that blank by the sequence 1000 to simulate the blank of M.

A More Efficient Universal TM

A efficient simulation of M by U, one that would not require us to shift symbols on the tape, would have U first determine the number of tape symbols M used. If there are between $2^{k-1}+1$ and 2^k symbols, U could use a k-bit binary code to represent the different tape symbols uniquely. Tape cells of M could be simulated by k of U's tape cells. To make things even easier, the given transitions of M could be rewritten by U to use the fixed-length binary code instead of the variable-length unary code we introduced.

3. Place 0, the start state of M, on the third tape, and move the head of U's second tape to the first simulated cell.

4. To simulate a move of M, U searches on its first tape for a transition $0^i10^j10^k10^l10^m$, such that 0^i is the state on tape 3, and 0^j is the tape symbol of M that begins at the position on tape 2 scanned by U. This transition is the one M would next make. U should:

   (a) Change the contents of tape 3 to 0^k; that is, simulate the state change of M. To do so, U first changes all the 0's on tape 3 to blanks, and then copies 0^k from tape 1 to tape 3.

   (b) Replace 0^j on tape 2 by 0^l; that is, change the tape symbol of M. If more or less space is needed (i.e., $i \neq l$), use the scratch tape and the shifting-over technique of Section 8.6.2 to manage the spacing.

   (c) Move the head on tape 2 to the position of the next 1 to the left or right, respectively, depending on whether $m = 1$ (move left) or $m = 2$ (move right). Thus, U simulates the move of M to the left or to the right.

5. If M has no transition that matches the simulated state and tape symbol, then in (4), no transition will be found. Thus, M halts in the simulated configuration, and U must do likewise.

6. If M enters its accepting state, then U accepts.

In this manner, U simulates M on w. U accepts the coded pair (M, w) if and only if M accepts w.

9.2.4 Undecidability of the Universal Language

We can now exhibit a problem that is RE but not recursive; it is the language L_u. Knowing that L_u is undecidable (i.e., not a recursive language) is in many ways more valuable than our previous discovery that L_d is not RE. The reason

The Halting Problem

One often hears of the *halting problem* for Turing machines as a problem similar to L_u — one that is RE but not recursive. In fact, the original Turing machine of A. M. Turing accepted by halting, not by final state. We could define $H(M)$ for TM M to be the set of inputs w such that M halts given input w, regardless of whether or not M accepts w. Then, the *halting problem* is the set of pairs (M, w) such that w is in $H(M)$. This problem/language is another example of one that is RE but not recursive.

is that the reduction of L_u to another problem P can be used to show there is no algorithm to solve P, regardless of whether or not P is RE. However, reduction of L_d to P is only possible if P is not RE, so L_d cannot be used to show undecidability for those problems that are RE but not recursive. On the other hand, if we want to show a problem not to be RE, then only L_d can be used; L_u is useless since it *is* RE.

Theorem 9.6: L_u is RE but not recursive.

PROOF: We just proved in Section 9.2.3 that L_u is RE. Suppose L_u were recursive. Then by Theorem 9.3, $\overline{L_u}$, the complement of L_u, would also be recursive. However, if we have a TM M to accept $\overline{L_u}$, then we can construct a TM to accept L_d (by a method explained below). Since we already know that L_d is not RE, we have a contradiction of our assumption that L_u is recursive.

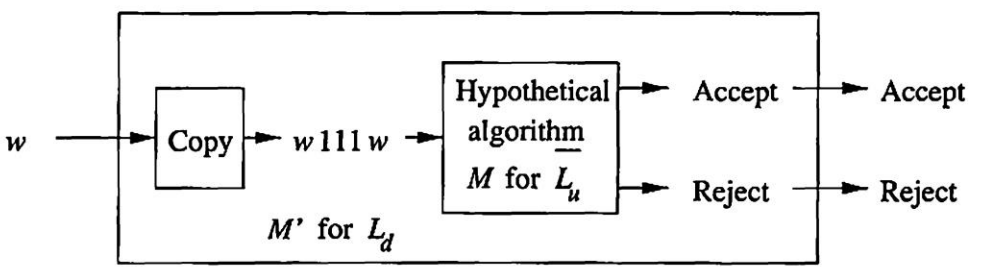

Figure 9.6: Reduction of L_d to $\overline{L_u}$

Suppose $L(M) = \overline{L_u}$. As suggested by Fig. 9.6, we can modify TM M into a TM M' that accepts L_d as follows.

1. Given string w on its input, M' changes the input to $w111w$. You may, as an exercise, write a TM program to do this step on a single tape. However, an easy argument that it can be done is to use a second tape to copy w, and then convert the two-tape TM to a one-tape TM.

2. M' simulates M on the new input. If w is w_i in our enumeration, then M' determines whether M_i accepts w_i. Since M accepts $\overline{L_u}$, it will accept if and only if M_i does not accept w_i; i.e., w_i is in L_d.

Thus, M' accepts w if and only if w is in L_d. Since we know M' cannot exist by Theorem 9.2, we conclude that L_u is not recursive. □

9.2.5 Exercises for Section 9.2

Exercise 9.2.1: Show that the halting problem, the set of (M, w) pairs such that M halts (with or without accepting) when given input w is RE but not recursive. (See the box on "The Halting Problem" in Section 9.2.4.)

Exercise 9.2.2: In the box "Why 'Recursive'?" in Section 9.2.1 we suggested that there was a notion of "recursive function" that competed with the Turing machine as a model for what can be computed. In this exercise, we shall explore an example of the recursive-function notation. A *recursive function* is a function F defined by a finite set of rules. Each rule specifies the value of the function F for certain arguments; the specification can use variables, nonnegative-integer constants, the successor (add one) function, the function F itself, and expressions built from these by composition of functions. For example, *Ackermann's function* is defined by the rules:

1. $A(0, y) = 1$ for any $y \geq 0$.
2. $A(1, 0) = 2$.
3. $A(x, 0) = x + 2$ for $x \geq 2$.
4. $A(x + 1, y + 1) = A\big(A(x, y + 1), y\big)$ for any $x \geq 0$ and $y \geq 0$.

Answer the following:

* a) Evaluate $A(2, 1)$.

! b) What function of x is $A(x, 2)$?

! c) Evaluate $A(4, 3)$.

Exercise 9.2.3: Informally describe multitape Turing machines that *enumerate* the following sets of integers, in the sense that started with blank tapes, it prints on one of its tapes $10^{i_1}10^{i_2}1\cdots$ to represent the set $\{i_1, i_2, \ldots\}$.

* a) The set of all perfect squares $\{1, 4, 9, \ldots\}$.

b) The set of all primes $\{2, 3, 5, 7, 11, \ldots\}$.

!! c) The set of all i such that M_i accepts w_i. *Hint*: It is not possible to generate all these i's in numerical order. The reason is that this language, which is $\overline{L_d}$, is RE but not recursive. In fact, a definition of the RE-but-not-recursive languages is that they can be enumerated, but not in numerical order. The "trick" to enumerating them at all is that we have to simulate all M_i's on w_i, but we cannot allow any M_i to run forever, since it would preclude trying any other M_j for $j \neq i$ as soon as we encountered some M_i that does not halt on w_i. Thus, we need to operate in rounds, where in the kth round we try only a limited set of M_i's, and we do so for only a limited number of steps. Thus, each round can be completed in finite time. As long as for each TM M_i and for each number of steps s there is some round such that M_i will be simulated for at least s steps, then we shall eventually discover each M_i that accepts w_i and enumerate i.

* **Exercise 9.2.4:** Let $L_1, L_2, \ldots, L_k$ be a collection of languages over alphabet Σ such that:

1. For all $i \neq j$, $L_i \cap L_j = \emptyset$; i.e., no string is in two of the languages.
2. $L_1 \cup L_2 \cup \cdots \cup L_k = \Sigma^*$; i.e., every string is in one of the languages.
3. Each of the languages L_i, for $i = 1, 2, \ldots, k$ is recursively enumerable.

Prove that each of the languages is therefore recursive.

*! **Exercise 9.2.5:** Let L be recursively enumerable and let $\overline{L}$ be non-RE. Consider the language

$$L' = \{0w \mid w \text{ is in } L\} \cup \{1w \mid w \text{ is not in } L\}$$

Can you say for certain whether L' or its complement are recursive, RE, or non-RE? Justify your answer.

! **Exercise 9.2.6:** We have not discussed closure properties of the recursive languages or the RE languages, other than our discussion of complementation in Section 9.2.2. Tell whether the recursive languages and/or the RE languages are closed under the following operations. You may give informal, but clear, constructions to show closure.

* a) Union.

b) Intersection.

c) Concatenation.

d) Kleene closure (star).

* e) Homomorphism.

f) Inverse homomorphism.