that the same class of functions on the integers can be computed by any general computing device. This is the class of partial recursive functions, sometimes called the class of *computable functions*. There is a mathematical definition of this class of functions that does not refer to programming languages, a second definition that uses a kind of idealized computing device called a *Turing machine*, and a third (equivalent) definition that uses lambda calculus (see Section 4.2). As mentioned in the biographical sketch on Alan Turing, a Turing machine consists of an infinite tape, a tape read–write head, and a finite-state controller. The tape is divided into contiguous cells, each containing a single symbol. In each computation step, the machine reads a tape symbol and the finite-state controller decides whether to write a different symbol on the current tape square and then whether to move the read–write head one square left or right. Part of the evidence that Church cited in formulating this thesis was the proof that Turing machines and lambda calculus are equivalent. The fact that all standard programming languages express precisely the class of partial recursive functions is often summarized by the statement that *all programming languages are Turing complete*. Although it is comforting to know that all programming languages are universal in a mathematical sense, the fact that all programming languages are Turing complete also means that computability theory does not help us distinguish among the expressive powers of different programming languages.

Noncomputable Functions

It is useful to know that some specific functions are not computable. An important example is commonly referred to as the *halting problem*. To simplify the discussion and focus on the central ideas, the halting problem is stated for programs that require one string input. If P is such a program and x is a string input, then we write $P(x)$ for the output of program P on input x.

> *Halting Problem:* Given a program P that requires exactly one string input and a string x, determine whether P halts on input x.

We can associate the halting problem with a function f_{halt} by letting $f_{halt}(P, x) =$ "halts" if P halts on input and $f_{halt}(P, x) =$ "does not halt" otherwise. This function f_{halt} can be considered a function on strings if we write each program out as a sequence of symbols.

The *undecidability of the halting problem* is the fact that the function f_{halt} is not computable. The undecidability of the halting problem is an important fact to keep in mind in designing programming language implementations and optimizations. It implies that many useful operations on programs cannot be implemented, even in principle.

Proof of the Undecidability of the Halting Problem. Although you will not need to know this proof to understand any other topic in the book, some of you may be interested in proof that the halting function is not computable. The proof is surprisingly short, but can be difficult to understand. If you are going to be a serious computer scientist, then you will want to look at this proof several times, over the course of several days, until you understand the idea behind it.

> Step 1: Assume that there is a program Q that solves the halting problem. Specifically, assume that program Q reads two inputs, both strings, and has the

following output:

$$Q(P, x) = \begin{cases} \text{halts} & \text{if } P(x) \text{ halts} \\ \text{does not halt} & \text{if } P(x) \text{ does not} \end{cases}.$$

An important part of this specification for Q is that $Q(P, x)$ always halts for every P and x.

Step 2: Using program Q, we can build a program D that reads one string input and sometimes does not halt. Specifically, let D be a program that works as follows:

$D(P)$ = if $Q(P, P)$ = halts then *run forever* else *halt*.

Note that D has only one input, which it gives twice to Q. The program D can be written in any reasonable language, as any reasonable language should have some way of programming if-then-else and some way of writing a loop or recursive function call that runs forever. If you think about it a little bit, you can see that D has the following behavior:

$$D(P) = \begin{cases} \text{halt} & \text{if } P(P) \text{ runs forever} \\ \text{run forever} & \text{if } P(P) \text{ halts} \end{cases}.$$

In this description, the word halt means that $D(P)$ comes to a halt, and runs forever means that $D(P)$ continues to execute steps indefinitely. The program $D(P)$ halts or does not halt, but does not produce a string output in any case.

Step 3: Derive a contradiction by considering the behavior $D(D)$ of program D on input D. (If you are starting to get confused about what it means to run a program with the program itself as input, assume that we have written the program D and stored it in a file. Then we can compile D and run D with the file containing a copy of D as input.) Without thinking about how D works or what D is supposed to do, it is clear that either $D(D)$ halts or $D(D)$ does not halt. If $D(D)$ halts, though, then by the property of D given in step 2, this must be because $D(D)$ runs forever. This does not make any sense, so it must be that $D(D)$ runs forever. However, by similar reasoning, if $D(D)$ runs forever, then this must be because $D(D)$ halts. This is also contradictory. Therefore, we have reached a contradiction.

Step 4: Because the assumption in step 1 that there is a program Q solving the halting problem leads to a contradiction in step 3, it must be that the assumption is false. Therefore, there is no program that solves the halting problem.

Applications

Programming language compilers can often detect errors in programs. However, the undecidability of the halting problem implies that some properties of programs cannot be determined in advance. The simplest example is halting itself. Suppose someone writes a program like this:

```
i = 0;
while (i != f(i)) i = g(i);
printf(... i ...);
```