2

# Computability

Some mathematical functions are computable and some are not. In all general-purpose programming languages, it is possible to write a program for each function that is computable in principle. However, the limits of computability also limit the kinds of things that programming language implementations can do. This chapter contains a brief overview of computability so that we can discuss limitations that involve computability in other chapters of the book.

## 2.1 PARTIAL FUNCTIONS AND COMPUTABILITY

From a mathematical point of view, a program defines a function. The output of a program is computed as a function of the program inputs and the state of the machine before the program starts. In practice, there is a lot more to a program than the function it computes. However, as a starting point in the study of programming languages, it is useful to understand some basic facts about computable functions.

The fact that not all functions are computable has important ramifications for programming language tools and implementations. Some kinds of programming constructs, however useful they might be, cannot be added to real programming languages because they cannot be implemented on real computers.

### 2.1.1 Expressions, Errors, and Nontermination

In mathematics, an expression may have a defined value or it may not. For example, the expression $3 + 2$ has a defined value, but the expression $3/0$ does not. The reason that $3/0$ does not have a value is that division by zero is not defined: division is defined to be the inverse of multiplication, but multiplication by zero cannot be inverted. There is nothing to try to do when we see the expression $3/0$; a mathematician would just say that this operation is undefined, and that would be the end of the discussion.

In computation, there are two different reasons why an expression might not have a value:

https://doi.org/10.1017/CBO9780511804175.003 Published online by Cambridge University Press

**ALAN TURING**

Alan Turing was a British mathematician. He is known for his early work on computability and his work for British Intelligence on code breaking during the Second World War. Among computer scientists, he is best known for the invention of the Turing machine. This is not a piece of hardware, but an idealized computing device. A Turing machine consists of an infinite tape, a tape read–write head, and a finite-state controller. 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. The importance of this idealized computer is that it is both very simple and very powerful.

Turing was a broad-minded individual with interests ranging from relativity theory and mathematical logic to number theory and the engineering design of mechanical computers. There are numerous published biographies of Alan Turing, some emphasizing his wartime work and others calling attention to his sexuality and its impact on his professional career.

The ACM Turing Award is the highest scientific honor in computer science, equivalent to a Nobel Prize in other fields.

- *Error termination:* Evaluation of the expression cannot proceed because of a conflict between operator and operand.
- *Nontermination:* Evaluation of the expression proceeds indefinitely.

An example of the first kind is division by zero. There is nothing to compute in this case, except possibly to stop the computation in a way that indicates that it could not proceed any further. This may halt execution of the entire program, abort one thread of a concurrent program, or raise an exception if the programming language provides exceptions.

The second case is different: There is a specific computation to perform, but the computation may not terminate and therefore may not yield a value. For example, consider the recursive function defined by

```
f(x:int) = if x=0 then 0 else x + f(x-2)
```

This is a perfectly meaningful definition of a *partial* function, a function that has a value on some arguments but not on all arguments. The expression f(4) calling the function f above has value $4 + 2 + 0 = 6$, but the expression f(5) does not have a value because the computation specified by this expression does not terminate.

### 2.1.2 Partial Functions

A partial function is a function that is defined on some arguments and undefined on others. This is ordinarily what is meant by function in programming, as a function declared in a program may return a result or may not if some loop or sequence of recursive calls does not terminate. However, this is not what a mathematician ordinarily means by the word function.

The distinction can be made clearer by a look at the mathematical definitions. A reasonable definition of the word function is this: A function $f\colon A \to B$ from set $A$ to set $B$ is a rule associating a unique value $y = f(x)$ in $B$ with every $x$ in $A$. This is almost a mathematical definition, except that the word rule does not have a precise mathematical meaning. The notation $f\colon A \to B$ means that, given arguments in the set $A$, the function $f$ produces values from set $B$. The set $A$ is called the *domain* of $f$, and the set $B$ is called the *range* or the *codomain* of $f$.

The usual mathematical definition of function replaces the idea of rule with a set of argument–result pairs called the graph of a function. This is the mathematical definition:

A *function* $f\colon A \to B$ is a set of ordered pairs $f \subseteq A \times B$ that satisfies the following conditions:

1. If $\langle x, y\rangle \in f$ and $\langle x, z\rangle \in f$, then $y = z$.
2. For every $x \in A$, there exists $y \in B$ with $\langle x, y\rangle \in f$.

When we associate a set of ordered pairs with a function, the ordered pair $\langle x, y\rangle$ is used to indicate that $y$ is the value of the function on argument $x$. In words, the preceding two conditions can be stated as (1) a function has at most one value for every argument in its domain, and (2) a function has at least one value for every argument in its domain.

A partial function is similar, except that a partial function may not have a value for every argument in its domain. This is the mathematical definition:

A *partial function* $f\colon A \to B$ is a set of ordered pairs $f \subseteq A \times B$ satisfying the preceding condition

1. If $\langle x, y\rangle \in f$ and $\langle x, z\rangle \in f$, then $y = z$.

In words, a partial function is single valued, but need not be defined on all elements of its domain.

### Programs Define Partial Functions

In most programming languages, it is possible to define functions recursively. For example, here is a function f defined in terms of itself:

```
f(x:int) = if x = 0 then 0 else x + f(x-2);
```

If this were written as a program in some programming language, the declaration would associate the function name f with an algorithm that terminates on every even x≥0, but diverges (does not halt and return a value) if x is odd or negative. The algorithm for f defines the following mathematical function $f$, expressed here as a set of ordered pairs:

$$f = \{\langle x, y\rangle \mid x \text{ is positive and even, } y = 0 + 2 + 4 + \cdots + x\}.$$

This is a partial function on the integers. For every integer $x$, there is at most one $y$ with $f(x) = y$. However, if $x$ is an odd number, then there is no $y$ with $f(x) = y$. Where the algorithm does not terminate, the value of the function is undefined. Because a function call may not terminate, this program defines a partial function.

## 2.1.3 Computability

Computability theory gives us a precise characterization of the functions that are computable in principle. The class of functions on the natural numbers that are computable in principle is often called the class of *partial recursive functions*, as recursion is an essential part of computation and computable functions are, in general, partial rather than total. The reason why we say "computable in principle" instead of "computable in practice" is that some computable functions might take an extremely long time to compute. If a function call will not return for an amount of time equal to the length of the entire history of the universe, then in practice we will not be able to wait for the computation to finish. Nonetheless, computability in principle is an important benchmark for programming languages.

### Computable Functions

Intuitively, a function is *computable* if there is some program that computes it. More specifically, a function $f: A \to B$ is computable if there is an algorithm that, given any $x \in A$ as input, halts with $y = f(x)$ as output.

One problem with this intuitive definition of computable is that a program has to be written out in some programming language, and we need to have some implementation to execute the program. It might very well be that, in one programming language, there is a program to compute some mathematical function and in another language there is not.

In the 1930s, Alonzo Church of Princeton University proposed an important principle, called Church's thesis. Church's thesis, which is a widely held belief about the relation between mathematical definitions and the real world of computing, states 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:

$$D(P) = \text{if } Q(P, P) = \text{halts then } \textit{run forever} \text{ else } \textit{halt}.$$

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 . . .);
```

It seems very likely that the programmer wants the while loop to halt. Otherwise, why would the programmer have written a statement to print the value of i after the loop halts? Therefore, it would be helpful for the compiler to print a warning message if the loop will not halt. However useful this might be, though, it is not possible for a compiler to determine whether the loop will halt, as this would involve solving the halting problem.

## 2.2 CHAPTER SUMMARY

Computability theory establishes some important ground rules for programming language design and implementation. The following main concepts from this short overview should be remembered:

- *Partiality:* Recursively defined functions may be partial functions. They are not always total functions. A function may be partial because a basic operation is not defined on some argument or because a computation does not terminate.
- *Computability:* Some functions are computable and others are not. Programming languages can be used to define computable functions; we cannot write programs for functions that are not computable in principle.
- *Turing completeness:* All standard general-purpose programming languages give us the same class of computable functions.
- *Undecidability:* Many important properties of programs cannot be determined by any computable function. In particular, the halting problem is undecidable.

When the value of a function or the value of an expression is undefined because a basic operation such as division by zero does not make sense, a compiler or interpreter can cause the program to halt and report the error. However, the undecidability of the halting problem implies that there is no way to detect and report an error whenever a program is not going to halt.

There is a lot more to computability and complexity theory than is summarized in the few pages here. For more information, see one of the many books on computability and complexity theory such as *Introduction to Automata Theory, Languages, and Computation* by Hopcroft, Motwani, and Ullman (Addison Wesley, 2001) or *Introduction to the Theory of Computation* by Sipser (PWS, 1997).

## EXERCISES

### 2.1 Partial and Total Functions

For each of the following function definitions, give the graph of the function. Say whether this is a partial function or a total function on the integers. If the function is partial, say where the function is defined and undefined.

For example, the graph of `f(x) = if x>0 then x+2 else x/0` is the set of ordered pairs $\{\langle x, x+2\rangle \mid x > 0\}$. This is a partial function. It is defined on all integers greater than 0 and undefined on integers less than or equal to 0.

Functions:

(a) `f(x) = if x+2>3 then x * 5 else x/0`
(b) `f(x) = if x<0 then 1 else f(x-2)`
(c) `f(x) = if x=0 then 1 else f(x-2)`

## 2.2 Halting Problem on No Input

Suppose you are given a function $\texttt{Halt}_\emptyset$ that can be used to determine whether a program that requires no input halts. To make this concrete, assume that you are writing a C or Pascal program that reads in another program as a string. Your program is allowed to call $\texttt{Halt}_\emptyset$ with a string input. Assume that the call to $\texttt{Halt}_\emptyset$ returns true if the argument is a program that halts and does not read any input and returns false if the argument is a program that runs forever and does not read any input. You should not make any assumptions about the behavior of $\texttt{Halt}_\emptyset$ on an argument that is not a syntactically correct program.

Can you solve the halting problem by using $\texttt{Halt}_\emptyset$? More specifically, can you write a program that reads a program text $P$ as input, reads an integer $n$ as input, and then decides whether $P$ halts when it reads $n$ as input? You may assume that any program $P$ you are given begins with a read statement that reads a single integer from standard input. This problem does not ask you to write the program to solve the halting problem. It just asks whether it is possible to do so.

If you believe that the halting problem can be solved if you are given $\texttt{Halt}_\emptyset$, then explain your answer by describing how a program solving the halting problem would work. If you believe that the halting problem cannot be solved by using $\texttt{Halt}_\emptyset$, then explain briefly why you think not.

## 2.3 Halting Problem on All Input

Suppose you are given a function $\texttt{Halt}_\forall$ that can be used to determine whether a program halts on all input. Under the same conditions as those of problem 2.2, can you solve the halting problem by using $\texttt{Halt}_\forall$?