CHAPTER 17
Theory of Computation

Objectives

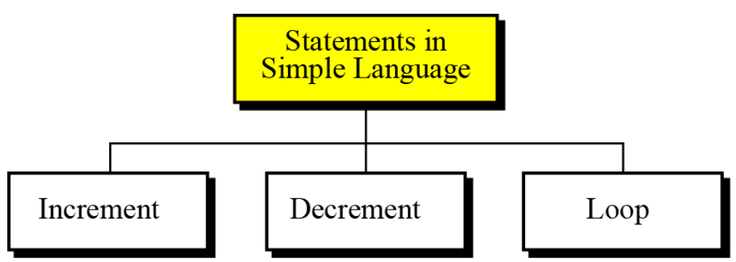
After studying this chapter, the student should be able to:

- Describe a programming language we call Simple Language and define its basic statements.
- Write macros in Simple Language using the combination of simple statements.
- Describe the components of a Turing machine as a computation model.
- Show how simple statements in Simple Language can be simulated using a Turing machine.
- Understand the Church-Turing thesis and its implication.
- Define the Gödel number and its application.
- Understand the concept of the halting problem and how it can be proved that this problem is unsolvable.
- Distinguish between solvable and unsolvable problems.
- Distinguish between polynomial and non-polynomial solvable problems.

17-1 SIMPLE LANGUAGE

We can define a computer language with only three statements: the increment statement, the decrement statement, and the loop statement (Figure 17.1).

Figure 17.1 Statements in Simple Language



17.1.1 Increment statement

The increment statement adds 1 to a variable. The format is shown in Algorithm 17.1.

Algorithm 17.1 The increment statement

```
incr (X)
```

17.1.2 Decrement statement

The decrement statement subtracts 1 from a variable. The format is shown in Algorithm 17.2.

Algorithm 17.2 The decrement statement

```
decr (X)
```

The loop statement repeats an action (or a series of actions) while the value of the variable is not 0. The format is shown in Algorithm 17.3.

Algorithm 17.3 Loop statement

```
while (X)
{
        decr (X)
        Body of the loop
}
```

It can be shown that this simple programming language with only three statements is as powerful—although not necessarily as efficient—as any sophisticated language in use today, such as C. To do so, we show how we can simulate several statements found in some popular languages.

Macros in Simple Language

We call each simulation a macro and use it in other simulations without the need to repeat code. A macro (short for macroinstruction) is an instruction in a high-level language that is equivalent to a specific set of one or more ordinary instructions in the same language.

Algorithm 17.4 shows how to use the statements in Simple Language to assign 0 to a variable X. It is sometimes called clearing a variable.

Algorithm 17.4 Macro X ← 0

```
while (X)
{
        decr (X)
}
```

Algorithm 17.5 shows how to use the statements in Simple Language to assign a positive integer n to a variable X. First clear the variable X, then increment X *n* times.

Algorithm 17.5 Macro X ← n

```
X ← 0
incr (X)
incr (X)
...
incr (X)                    // The statement incr (X) is repeated n times.
```

Algorithm 17.6 simulates the macro Y ← X in Simple Language. Note that we can use an extra line of code to restore the value of X.

Algorithm 17.6 Macro Y ← X

```
Y ← 0
while (X)
{
        decr (X)
        incr (Y)
}
```

Algorithm 17.7 simulates the macro Y ← Y + X in Simple Language. Again, we can use more code lines to restore the value of X to its original value.

Algorithm 17.7 Macro Y ← Y + X

```
while (X)
{
        decr (X)
        incr (Y)
}
```

Algorithm 17.8 simulates the macro Y ← Y × X in Simple Language. We can use the addition macro because integer multiplication can be simulated by repeated addition. Note that we need to preserve the value of X in a temporary variable, because in each addition we need the original value of X to be added to Y.

Algorithm 17.8 Macro Y ← Y × X

```
TEMP ← Y
Y ← 0
while (X)
{
        decr (X)
        Y ← Y + TEMP
}
```

Algorithm 17.9 simulates the macro Y ← YX in Simple Language. We do this using the multiplication macro because integer exponentiation can be simulated by repeated multiplication.

Algorithm 17.9 Macro $Y \leftarrow Y^X$

```
TEMP ← Y
Y ← 1
while (X)
{
        decr (X)
        Y ← Y × TEMP
}
```

Algorithm 17.10 simulates the seventh macro in Simple Language. This macro simulates the decision-making (if) statement of modern languages. In this macro, the variable X has only one of the two values 0 or 1. If the value of X is not 0, A is executed in the loop.

Algorithm 17.10 Macro if X then A

```
while (X)
{
        decr (X)
        A
}
```

It is obvious that we need more macros to make Simple Language compatible with contemporary languages. Creating other macros is possible, although not trivial.

Input and output

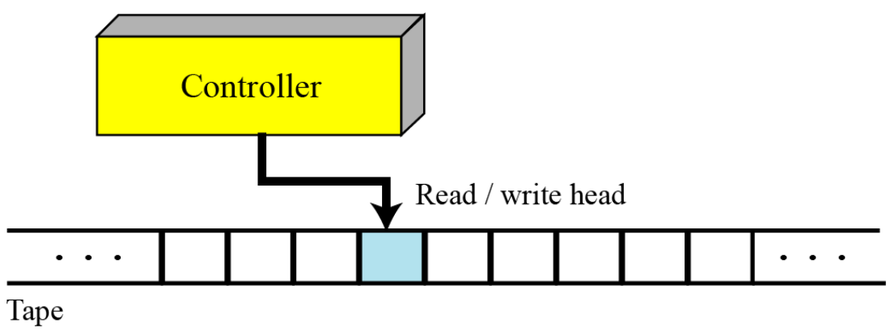
In this simple language the statement read X can be simulated using (X ← *n*). We also simulate the output by assuming that the last variable used in a program holds what should be printed. Remember that this is not a practical language, it is merely designed to prove some theorems in computer science.

17-2 THE TURING MACHINE

The Turing machine was introduced in 1936 by Alan M. Turing to solve computable problems, and is the foundation of modern computers. In this section we introduce a very simplified version of the machine to show how it works.

A Turing machine is made of three components: a tape, a controller, and a read/write head (Figure 17.2).

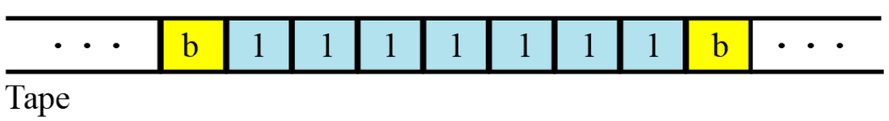
Figure 17.2 The Turing machine



Tape

Although modern computers use a random-access storage device with finite capacity, we assume that the Turing machine's memory is infinite. The tape, at any one time, holds a sequence of characters from the set of characters accepted by the machine. For our purpose, we assume that the machine can accept only two symbols: a blank (b) and digit 1.

Figure 17.3 The tape in the Turing machine



Read/write head

The read/write head at any moment points to one symbol on the tape. We call this symbol the current symbol. The read/write head reads and writes one symbol at a time from the tape. After reading and writing, it moves to the left or to the right. Reading, writing, and moving are all done under instructions from the controller.

Controller

The controller is the theoretical counterpart of the central processing unit (CPU) in modern computers. It is a finite state automaton, a machine that has a predetermined finite number of states and moves from one state to another based on the input.

Figure 17.4 Transition state diagram for the Turing machine

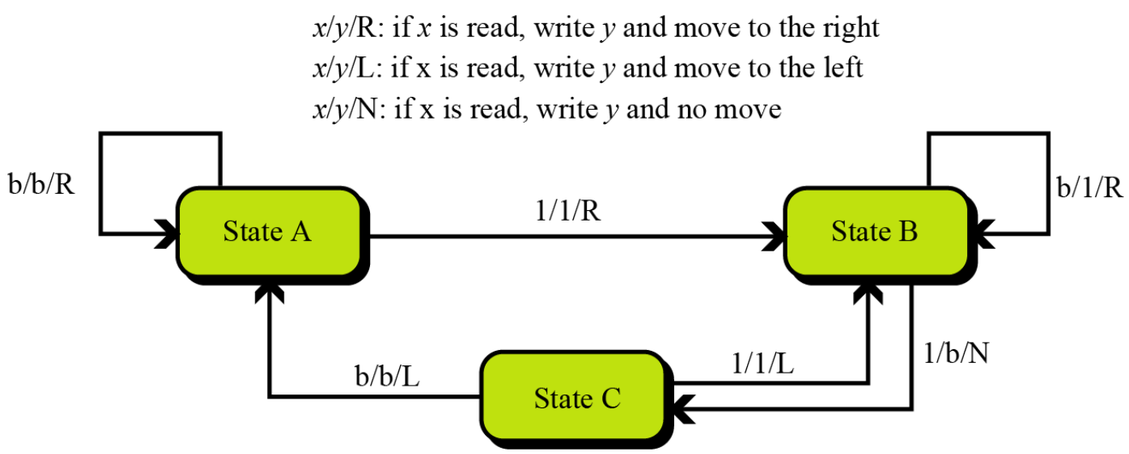


Table 17.1 Transition table

Current state	*Read*	*Write*	*Move*	*New state*
A	b	b	R	A
A	1	1	R	B
B	b	1	R	B
B	1	b	N	C
C	b	b	L	A
C	1	1	L	B

Example 17.1

A Turing machine has only two states and the following four instructions:

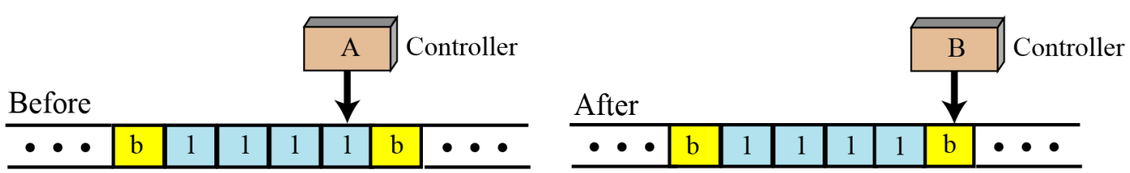
1. (A, b, b, L, A) 2. (A, 1, 1, R, B) 3. (B, b, b, L, A) 4. (B, 1, b, R, A)

If the machine starts with the configuration shown in Figure 17.5, what is the configuration of the machine after executing one of the above instructions?

Solution

The machine is in state A and the current symbol is 1, which means that only the second instruction, (A, 1, 1, R, B) can be executed. The new configuration is also shown in Figure 17.5. Note that the state of the controller has been changed to B and the read/write head has moved one symbol to the right.

Figure 17.5 Example 17.1

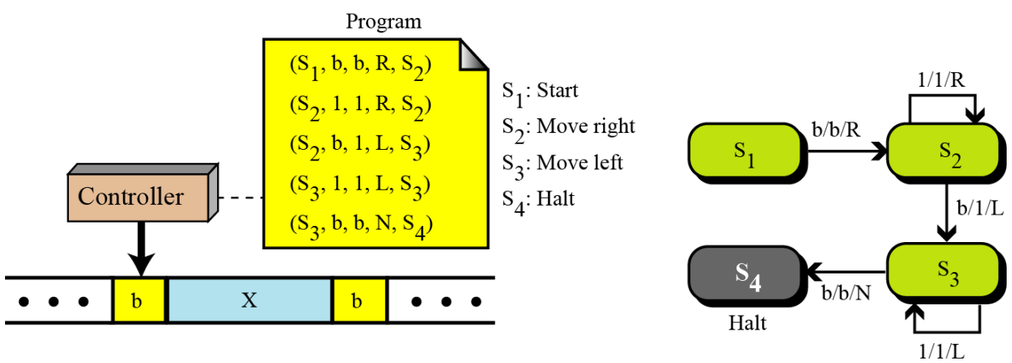


We can now write programs that implement the statements of Simple Language. Note that these statements can be written in many different ways: we have chosen the simplest or most convenient for our educational purpose, but they are not necessarily the best ones.

Increment statement

Figure 17.6 shows the Turing machine for the incr(X) statement. The controller has four states, S1 through S4. State S1 is the starting state, state S2 is the moving-right state, state S3 is the moving-left state, and state 4 is the halting state.

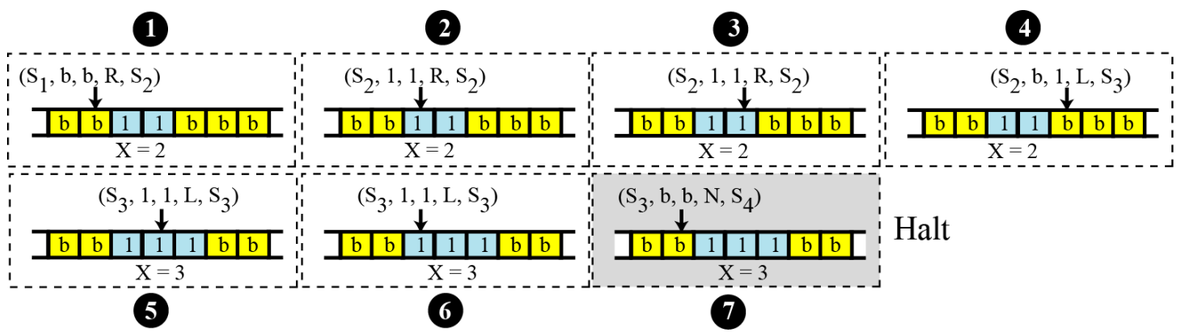
Figure 17.6 The Turing machine for the incr (X) statement



Example 17.2

It shows how the Turing machine can increment X when X = 2.

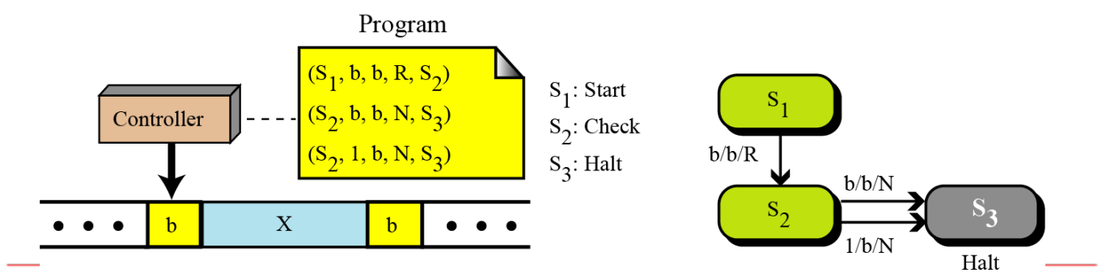
Figure 17.7 Example 17.2



Decrement statement

We implement the decr(X) statement using the minimum number of instructions. The reason is that we need to use this statement in the next statement, the while loop, which will also be used to implement all macros.

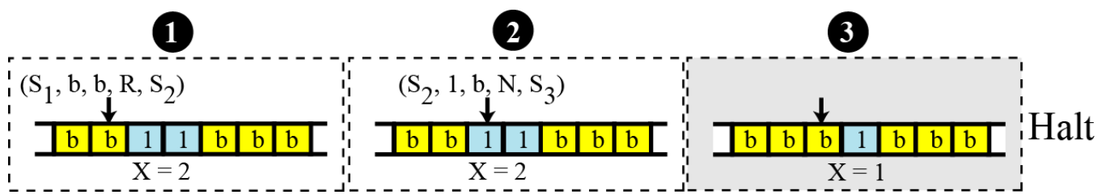
Figure 17.8 The Turing machine for the decr (X) statement



Example 17.3

It shows how the Turing machine can decrement X when X = 2.

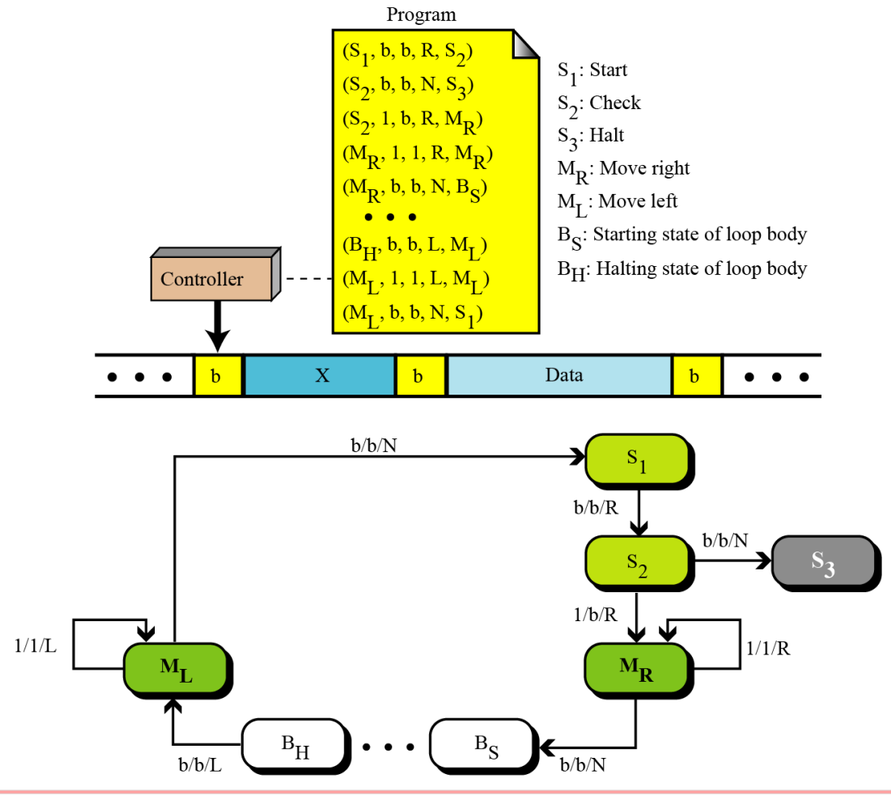
Figure 17.9 Example 17.3



Loop statement

To simulate the loop, we assume that X and the data to be processed by the body of the loop are stored on the tape separated by a single blank symbol. Figure 17.10 shows the table, the program, and the state transition diagram for a general loop statement. The three states S1, S2, and S3 control the loops by determining X and exiting the loop if X = 0. Compare these three statements to the three statements used in the decrement statement in Figure 17.8.

Figure 17.10 The Turing machine for the while loop statement



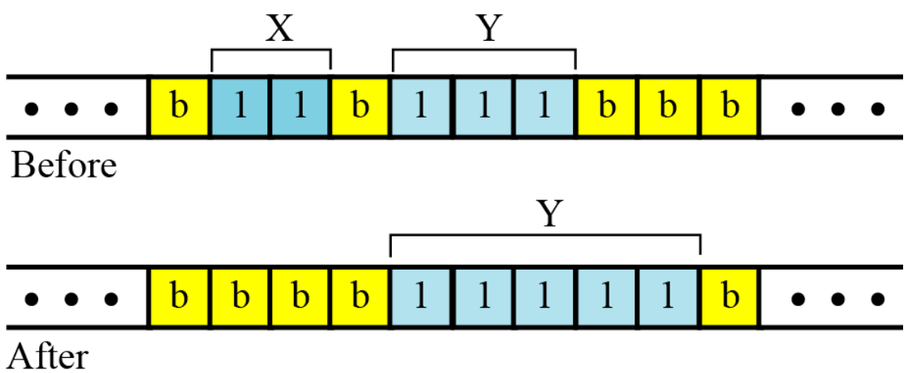
Example 17.4

Let us show a very simple example. Suppose we want to simulate the fourth macro, Y← Y + X (page 444). As we discussed before, this macro can be simulated using the while statement in Simple Language:

```
while (X)
{
        decr (X)
        incr (Y)
}
```

To make the procedure shorter, we assume that X = 2 and Y = 3, so the result is Y = 5. Figure 17.11 shows the state of the tape before and after applying the macro.

Figure 17.11 Configuration of the tapes for Example 17.4



We have shown that a Turing machine can simulate the three basic statements in Simple Language. This means that the Turing machine can also simulate all the macros we defined for Simple Language. Can the Turing machine therefore solve any problem that can be solved by a computer? The answer to this question can be found in the Church–Turing thesis.

The Church–Turing Thesis
If an algorithm exists to do a symbol manipulation task, then a Turing machine exists to do that task.

17-3 GÖDEL NUMBERS

In theoretical computer science, an unsigned number is assigned to every program that can be written in a specific language. This is usually referred to as the Gödel number, named after the Austrian mathematician Kurt Gödel. This assignment has many advantages. First, programs can be used as a single data item as input to other programs. Second, programs can be referred to by just their integer representations. Third, the numbering can be used to prove that some problems cannot be solved by a computer.

Different methods have been devised for numbering programs. We use a very simple transformation to number programs written in our Simple Language. Simple Language uses only fifteen symbols (Table 17.2).

Table 17.2 Code for symbols used in Simple Language

Symbol	*Hex code*	*Symbol*	*Hex code*
1	1	9	9
2	2	incr	A
3	3	decr	B
4	4	while	C
5	5	{	D
6	6	}	E
7	7	X	F
8	8		

Using the table, we can represent any program written in Simple Language by a unique positive integer by following these steps:

1. Replace each symbol with the corresponding hexadecimal code from the table.
2. Interpret the resulting hexadecimal number as an unsigned integer.

Example 17.5

What is the Gödel number for the program incr (X)?

Solution

Replace each symbol by its hexadecimal code

$$\text{incr X} \rightarrow (\text{AF})_{16} \rightarrow 175$$

So this program can be represented by the number 175.

To show that the numbering system is unique, use the following steps to interpret a Gödel number:

. Convert the number to hexadecimal.
. Interpret each hexadecimal digit as a symbol using Table 17.2 (ignore a 0).

Note that while any program written in Simple Language can be represented by a number, not every number can be interpreted as a valid program. After conversion, if the symbols do not follow the syntax of the language, the number is not a valid program.

Example 17.6

Interpret 3058 as a program.

Solution

Change the number to hexadecimal and replace each digit with the corresponding symbol:

$$3058 \rightarrow (BF2)_{16} \rightarrow \text{decr X 2} \rightarrow \textbf{decr} (X_2)$$

This means that the equivalent code in Simple Language is decr (X2). Note that in Simple Language, each program includes input and output. This means that the combination of a program and its inputs defines the Gödel number.

16-4 THE HALTING PROBLEM

Almost every program written in a programming language involves some form of repetition—loops or recursive functions. A repetition construct may never terminate (halt): that is, a program can run forever if it has an infinite loop. For example, the following program in Simple Language never terminates:

```
X ← 1
while (X)
{
}
```

A classical programming question is:

Can we write a program that tests whether or not any program, represented by its Gödel number, will terminate?

The existence of this program would save programmers a lot of time. Running a program without knowing if it halts or not is a tedious job. Unfortunately, it has now been proven that such a program cannot exist—much to the disappointment of programmers!

Instead of saying that the testing program does not exist and can never exist, the computer scientist says “The halting problem is not solvable”.

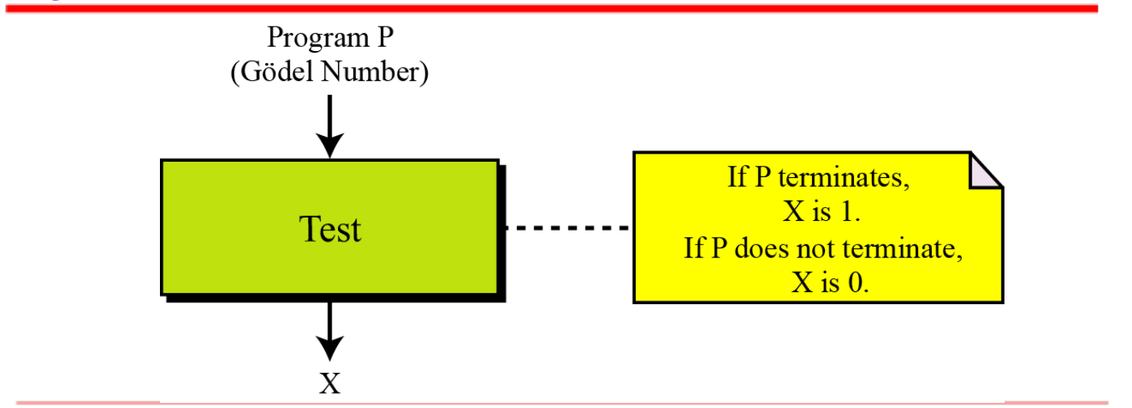
Proof

Let us give an informal proof about the nonexistence of this testing program. Our method, called *proof by contradiction*, is often used in mathematics: we assume that the program does exist, then show that its existence creates a contradiction—therefore, it cannot exist. We use three steps to show the proof in this approach.

Step 1

In this step, we assume that a program, called Test, exists. It can accept any program such as P, represented by its Gödel number, as input, and outputs either 1 or 0. If P terminates, the output of Test is 1: if P does not terminate, the output of Test is 0 (Figure 17.14).

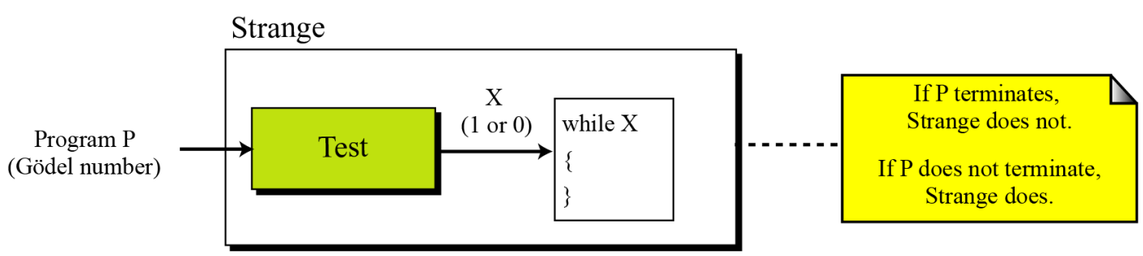
Figure 17.12 Step 1 in the proof



Step 2

In this step, we create another program called Strange that is made of two parts: a copy of Test at the beginning and an empty loop—a loop with an empty body—at the end. The loop uses X as the testing variable, which is actually the output of the Test program. This program also uses P as the input.

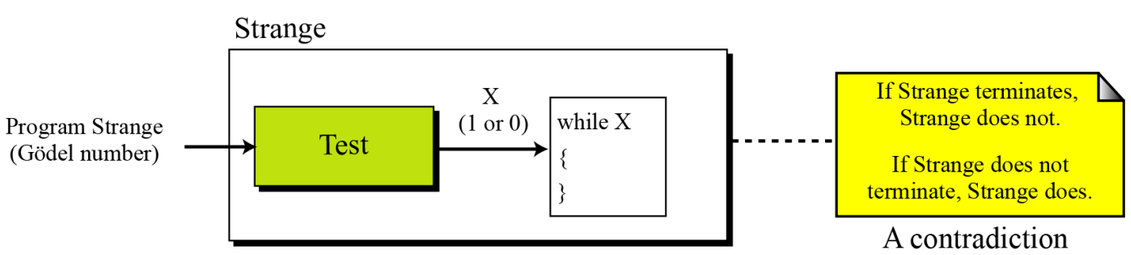
Figure 17.13 Step 2 in the proof



Step 3

Having written the program Strange, we test it with itself (its Gödel number) as input. This is legitimate because we did not put any restrictions on P. Figure 17.16 shows the situation.

Figure 17.14 Step 3 in the proof



Contradiction

If we assume that Test exists, we have the following contradictions:
Strange does not terminate if Strange terminates.
Strange terminates if Strange does not terminate.

This proves that the Test program cannot exist and that we should stop looking for it, so…

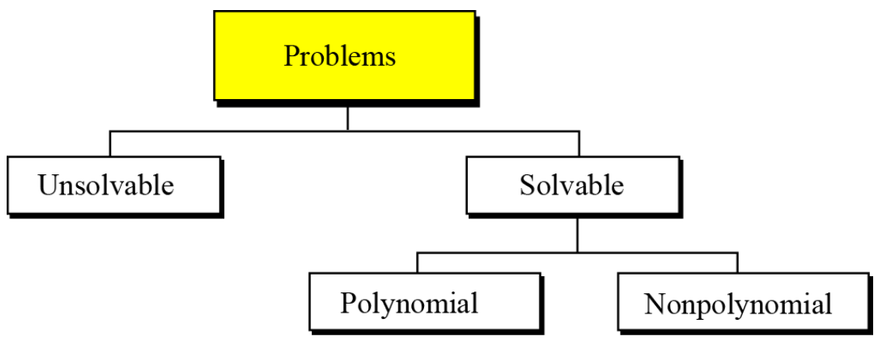
The halting problem is unsolvable.

The un-solvability of the halting program has proved that many other programs are also unsolvable, because if they are solvable, then the halting problem is solvable— which is not.

17-5 THE COMPLEXITY OF PROBLEMS

Now that we have shown that at least one problem is unsolvable by a computer, we'll touch on this important issue a bit more. In computer science, we can say that, in general, problems can be divided into two categories: solvable problems and unsolvable problems. The solvable problems can themselves be divided into two categories: polynomial and non-polynomial problems (Figure 17.17).

Figure 17.15 Taxonomy of problems



There are an infinite number of problems that cannot be solved by a computer: one is the halting problem. One method to prove that a problem is not solvable is to show that if that problem is solvable, the halting problem is solvable too. In other words, prove that the solvability of a problem results in the solvability of the halting problem.

There are many problems that can be solved by a computer. However, we often want to know how long it takes for the computer to solve that problem. In other words, how complex is the program? The complexity of a program can be measured in several different ways, such as its run time, the memory it needs, and so on. One approach is the program's run time—how long does the program take to run?

Complexity of solvable problems

One way to measure the complexity of a solvable problem is to find the number of operations executed by the computer when it runs the program.

Big-O notation

With the speed of computers today, we are not as concerned with exact numbers as with general orders of magnitude. This simplification of efficiency is known as big-O notation. We present the idea of this notation without delving into its formal definition and calculation. In big-O notation, the number of operations given as a function of the number of inputs. The notation O(n) means a program does n operations for n inputs, while the notation O(n2) means a program does n2 operations for n inputs.

Example 17.7

Imagine we have written three different programs to solve the same problem. The first one has a complexity of O(log10 *n*), the second O(*n*), and the third O(*n*2). Assuming 1 million inputs, how long does it take to execute each of these programs on a computer that executes one instruction in 1 microsecond, that is, 1 million instructions per second?

Solution

1st program:	n = 1,000,000	$O(\log_{10} n) \rightarrow 6$	Time → 6 μs
2nd program:	n = 1,000,000	$O(n) \rightarrow 1{,}000{,}000$	Time → 1 sec
3rd program:	n = 1,000,000	$O(n^2) \rightarrow 10^{12}$	Time → 277 h

Polynomial problems

If a program has a complexity of O(log$n$), O(n), O(n2), O(n3), O(n4), or O(nk), where k is a constant, it is called polynomial. With the speed of computers today, we can get solutions to polynomial problems with a reasonable number of inputs, for example 1000 to 1 million.

Non-polynomial problems

If a program has a complexity that is greater than a polynomial—for example, O(10n) or O(n!)—it can be solved if the number of inputs is very small, such as fewer than 100. If the number of inputs is large, one could sit in front of the computer for months to see the result of a non-polynomial problem.