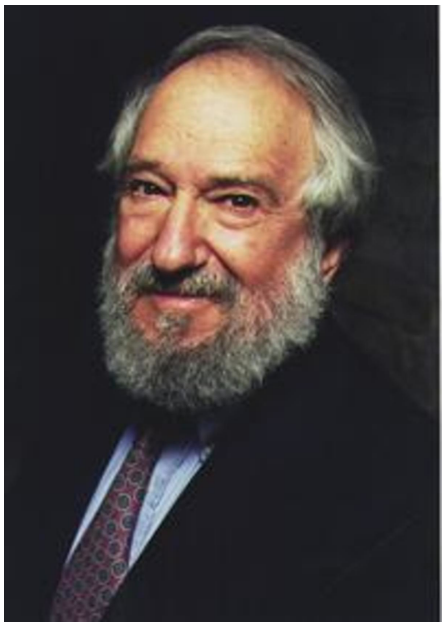
CHAPTER 1
A Gentle Introduction

> In many schools today . . . the computer is being used to program the child. In my vision, the child programs the computer and, in doing so, both acquires a sense of mastery over a piece of the most modern and powerful technology and establishes an intimate contact with some of the deepest ideas from science, from mathematics, and from the art of intellectual model-building.
>
> — Seymour Papert, *Mindstorms,* 1980



Seymour Papert (1928–2016)

In the 1960s, Professor Seymour Papert at MIT used a language called LOGO to teach programming to schoolchildren in the Boston area, who wrote programs to control a robotic turtle. The turtle could move forward or backward, rotate a specified number of degrees around its center, and draw pictures on large sheets of paper with a pen mounted on its underside. The LOGO turtle thereby became the first programming microworld, designed to teach the basics of computation in a simplified environment.

This book is about the ideas that underlie the science of computing. You won't, however, get much out of it through reading alone. Computing, after all, is an activity. As with most activities, one learns computing best through practice. To get you started, this book provides you with the tools you need to solve simple computational problems on your own. That process of necessity involves ***programming,*** which is the process of transforming a strategy for solving a problem into a precise formulation that can be executed by a computer.

At the same time, it is hard to learn programming through the metaphorical equivalent of jumping in at the deep end of the pool. You have to approach the subject more gradually. Modern programming languages involve so many details that their complexity gets in the way of understanding the bigger picture.

To avoid overwhelming beginners with the intricacies inherent in those languages, computer science courses often introduce programming in the context of a simplified environment called a ***microworld.*** By design, microworlds are easy to learn and enable students to start programming right away. In the process, those students become familiar with the fundamental concepts of programming without having to master a lot of extraneous details.

Many different microworlds have flourished over the years, including the Project LOGO Turtle described briefly on the title page of this chapter. This book uses a microworld called Karel that we have used with great success in our introductory courses here at Stanford for more than 30 years. Using Karel enables you to solve challenging problems from the very beginning. And because the Karel environment encourages imagination and creativity, you can have a lot of fun along the way.

1.1 Introducing Karel


Richard Pattis

In the 1970s, a Stanford graduate student named Rich Pattis decided that it would be easier to teach the fundamentals of programming if students could learn those ideas in an environment free from the complexities that characterize most programming languages. Drawing inspiration from the success of the LOGO project, Pattis designed a microworld in which students teach a virtual robot to solve simple problems. Pattis called his robot ***Karel*** after the Czech playwright Karel Čapek whose 1923 play *R.U.R.* (*Rossum's Universal Robots*) gave the word *robot* to the English language. Karel was an immediate success and soon spread to universities all over the world.

Programming in Karel

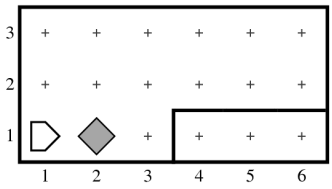
Karel is a very simple robot living in a very simple world. By giving Karel a set of instructions, you can direct it to perform certain tasks within its world. Those

instructions constitute a ***program***. Generically, the text that makes up a program is called ***code.*** When you write a Karel program, you must do so in a precise way so that Karel can correctly interpret it. Every program you write must obey a set of ***syntactic rules*** that define whether that program is legal.

In many respects, the rules of the Karel programming language are similar to those you will see in more sophisticated languages. The most important difference is that Karel's programming language is tiny—so small, in fact, that it is easy to learn everything there is to know about the Karel language in less than an hour. The details are easy to master. Even so, you will discover that solving a problem in Karel's world can be extremely challenging. Solving problems is the essence of programming. You learn the rules to unlock the problem-solving power.

Karel's world

Karel's world is defined by ***streets*** running from west to east and ***avenues*** running from south to north. The intersection of a street and an avenue is called a ***corner***. Karel can only be positioned on corners and must be facing in one of the four standard compass directions (north, east, south, and west). In the following sample world, Karel is facing east at the corner of 1st Street and 1st Avenue:



Several other components of Karel's world can be seen in this example. The object in front of Karel is a ***beeper***. According to Rich Pattis, beepers are "plastic cones which emit a quiet beeping noise." Karel can only detect a beeper if it is on the same corner. The solid lines in the diagram are ***walls***. Karel's world always has walls along the edges, but the world may also contain internal walls that serve as barriers.

Karel's built-in functions

The operations that Karel performs as it executes a program are called ***functions.*** When Karel is shipped from the factory, it knows how to execute only the four functions shown in Figure 1-1. The parentheses that appear in each of these examples are part of Karel's syntax and specify that you want to perform that operation, which in programming terminology is known as ***calling the function.***

Several of the built-in functions place specific restrictions on Karel's activities. If Karel tries to do something illegal, such as moving through a wall or picking up a

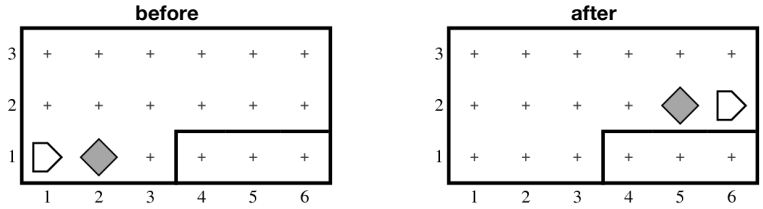
FIGURE 1-1 The built-in Karel Instructions

`move()`	Karel moves forward one block. Karel cannot move forward if there is a wall blocking the way.
`turnLeft()`	Karel rotates 90 degrees to the left (counterclockwise).
`pickBeeper()`	Karel picks up one beeper from the current corner and stores that beeper in its beeper bag, which can hold an infinite number of beepers. Karel can execute the `pickBeeper` function only if there is a beeper on that corner.
`putBeeper()`	Karel puts a beeper from its bag down on the current corner. Karel can execute the `putBeeper` function only if there are beepers in its beeper bag.

nonexistent beeper, an ***error condition*** occurs. Whenever an error arises, Karel displays a message describing what went wrong and stops executing the program.

1.2 Teaching Karel to solve problems

For the most part, learning to program in Karel is a matter of figuring out how to use Karel's limited set of operations to solve a specified problem. As a simple example, suppose that you want Karel to move the beeper from its initial position on 2nd Avenue and 1st Street to the center of the ledge at 5th Avenue and 2nd Street. Thus, your goal is to write a Karel program that accomplishes the task illustrated in the following before-and-after diagram:

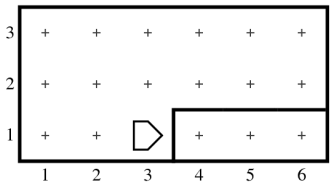


Getting started

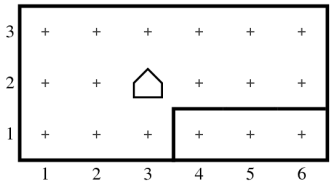
The first few steps in solving this problem are simple enough. You need to tell Karel to move forward, pick up the beeper, and then move forward again to reach the base of the ledge. The Karel simulator allows you to execute instructions by typing them into an interactive window called the ***Karel console.*** The first three steps in the program therefore look like this:

```
Karel Console
> move()
> pickBeeper()
> move()
>
```

Executing these function calls leaves Karel in the following position:



From here, Karel's next step is to turn left to begin climbing the ledge. That operation is also easy, because Karel's set of built-in functions includes **`turnLeft`**. Calling **`turnLeft`** at the end of the preceding program leaves Karel facing north on the corner of 3rd Avenue and 1st Street. If you then call the **`move`** instruction, Karel will move north to reach the following position:



The next thing you need to do is get Karel to turn right so that it is again facing east. While this operation is conceptually as easy as getting Karel to turn left, there is a slight problem: Karel's language includes a **`turnLeft`** instruction, but no **`turnRight`** instruction. It's as if you bought the economy model only to discover that it is missing an important feature.

At this point, you have your first opportunity to begin thinking like a programmer. You have access to a set of Karel functions, but not exactly the set you need. What can you do? Can you accomplish the effect of a **`turnRight`** function using only the capabilities you have? The answer, of course, is yes. You can turn right by turning left three times. After three left turns, Karel will be facing in the desired direction. The next three steps in the program might therefore be

```
turnLeft()
turnLeft()
turnLeft()
```

Although turning left three times has the desired effect, it is hardly an elegant solution. What you as the programmer want to say is

```
turnRight()
```

The only difficulty is that Karel doesn't yet have a definition for the **`turnRight`** function. To use this more expressive operation in your program, you first have to teach Karel what **`turnRight`** means.

Defining functions

One of the most powerful features of the Karel programming language is the ability to define new functions. Whenever you have a sequence of Karel operations that performs some useful task—such as turning right—you can give that sequence a name. The operation of encapsulating a sequence of instructions under a new name is called ***defining a function.*** The format for defining a function looks like this:

```
function name() {
   statements that make up the body of the function
}
```

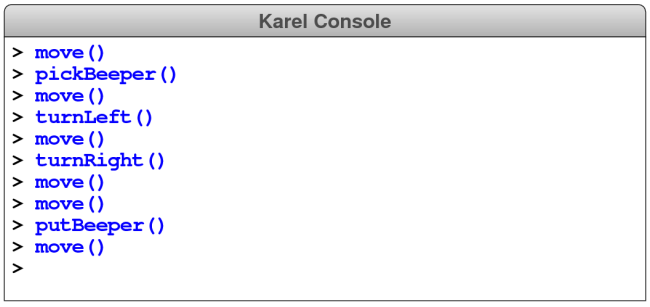
In this pattern, *name* represents the name you have chosen for the new function. To complete the definition, all you have to do is specify the statements between the curly braces. The only difference between the statements in a function and those you enter on the console is that each statement in a Karel function must end with a semicolon. For example, you can use the editor window to define the **turnRight** function as follows, usually as part of a larger program:

```
function turnRight() {
   turnLeft();
   turnLeft();
   turnLeft();
}
```

Once you've defined a function like **turnRight**, you can think of it as a new primitive operation, just like **move** or **turnLeft**. In a sense, defining a new function is like buying an upgrade for your robot that includes the missing operations.

Completing the program

After turning right to face the top of the ledge, the rest of the program is easy. All you need to do is move forward twice, put down the beeper, and then move forward to reach the desired final state. The complete sequence of Karel operations you need to solve the program from beginning to end looks like this:



Instead of typing each instruction into the console, it makes sense to define a new function that contains this sequence of instructions. You can then call that function with a single name. That function, which appears in Figure 1-2 together with the definition of **turnRight**, constitutes a complete Karel program.

In addition to the definitions of the functions **moveBeeperToLedge** and **turnRight**, Figure 1-2 also includes two examples of an important programming feature called a ***comment,*** which consists of text designed to explain the operation of the program to human readers. In Karel, comments begin with the characters **/*** and end with the characters ***/**. The first comment describes the operation of the program as a whole; the second describes the **turnRight** function. In a program this short, such comments may seem unnecessary. As programs become more complicated, however, comments quickly become essential tools to document the program design and make it easier for other programmers to understand.

Using library functions

Although the code in Figure 1-2 explicitly includes the definition of **turnRight**, it is tedious to have to copy that code into every program that needs that function. For

FIGURE 1-2 **Program to move the beeper up to the ledge**

```
/*
 * File: MoveBeeperToLedge.k
 * -------------------------
 * This program solves the problem of moving a beeper to a ledge.
 */

function moveBeeperToLedge() {
   move();
   pickBeeper();
   move();
   turnLeft();
   move();
   turnRight();
   move();
   putBeeper();
   move();
}

/*
 * Turns Karel right 90 degrees.
 */

function turnRight() {
   turnLeft();
   turnLeft();
   turnLeft();
}
```

the most common operations, it makes sense to store them in a way that makes it easy to reuse them in other programs. In computer science, collections of useful functions and other program components are called ***libraries.*** For example, the **turnRight** function and the equally useful **turnAround** function are both included in a library called **turns**, which you can use simply by including the following line at the beginning of your program:

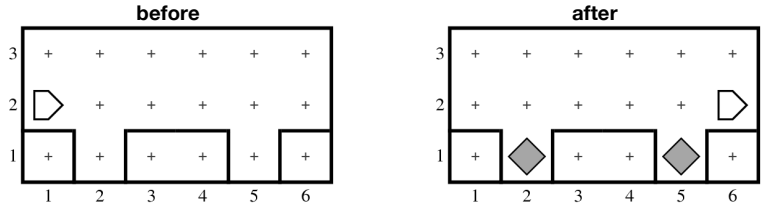
```
import "turns";
```

This statement imports the **turns** library, which includes definitions for the functions **turnRight** and **turnAround**. All the Karel examples in the rest of this chapter make use of this feature.

Decomposition

Whenever you begin the solution of a programming problem—no matter whether that program is written in Karel or a more advanced programming language—your first task is to figure out how to divide the complete problem into smaller pieces called ***subproblems,*** each of which can be implemented as a separate function. That process is called ***decomposition.*** Decomposition is one of the most powerful strategies that programmers use to manage complexity, and you will see it again and again throughout this book.

To get a sense of how decomposition works in the context of a very simple problem, imagine that Karel is standing on a "road" as shown on the left side of the following before-and-after diagram:



Karel's job is to fill each of the two potholes—the one on 2nd Avenue and the one on 5th Avenue—with a beeper and then continue on to the next corner, ending up in the position shown on the right.

Although you could solve this problem using the four predefined instructions, you can use functions to improve the structure of your program. If nothing else, you can use **turnAround** and **turnRight** to shorten the program and make its intent clearer. More importantly, you can use decomposition to break the problem down into subproblems and then solve those problems independently. You can, for example, break the problem of filling the pothole into the following subproblems:

1. Move one block forward to reach the first pothole on 2nd Avenue.
2. Fill the pothole by dropping a beeper into it.
3. Move three blocks forward to reach the second pothole on 5th Avenue.
4. Fill the pothole by dropping a beeper into it.
5. Move one block forward to reach the desired final position.

If you think about the problem in this way, you can use functions to ensure that the program reflects your conception of the problem structure, as shown in Figure 1-3.

FIGURE 1-3 Karel program to fill two potholes

```
/*
 * File: FillTwoPotholes.k
 * -----------------------
 * This program instructs Karel to fill two potholes.
 */

import "turns";

/*
 * This function fills two potholes, which must be on 2nd and 5th Avenues.
 */

function fillTwoPotholes() {
   move();
   fillPothole();
   move();
   move();
   move();
   fillPothole();
   move();
}

/*
 * Fills a pothole immediately underneath Karel.  When you call
 * this function, Karel must be standing just above the pothole,
 * facing east.  When the function returns, Karel will be in its
 * original position above the repaired pothole.
 */

function fillPothole() {
   turnRight();
   move();
   putBeeper();
   turnAround();
   move();
   turnRight();
}
```

As with any programming problem, there are other decomposition strategies you might have tried. Some strategies make the program easier to read, while others only make the meaning more opaque. As programming problems become more complex, decomposition will turn out to be one of the most important aspects of the design process.

Choosing an effective decomposition is much more of an art than a science, although you will find that you get better with practice. Section 1.4 presents some general guidelines that will help you in that process.

1.3 Control statements

As useful as it is, the ability to define new functions does not actually enable Karel to solve any new problems. Because each function name is merely shorthand for a specific set of instructions, it is always possible to expand a program written as a series of function calls into a single function that accomplishes the same task, although the resulting code is likely to be long and difficult to read. The instructions are still executed in a fixed order that does not depend on the state of Karel's world. Before you can solve more interesting problems, you need to learn how to write programs in which this strictly linear, step-by-step order of operations does not apply. To unlock the extraordinary power that this ability provides, you need to learn several new statements in Karel's programming language that enable Karel to examine its world and change its execution pattern accordingly.

Statements that affect the order in which a program executes instructions are called ***control statements.*** Control statements fall into the following two classes:

1. ***Conditional statements.*** Conditional statements specify that certain statements in a program should be executed only if a particular condition holds. In Karel, you specify conditional execution using an **`if`** statement.
2. ***Iterative statements.*** Iterative statements specify that certain statements in a program should be executed repeatedly, forming what programmers call a ***loop.*** Karel supports two iterative statements: a **`repeat`** statement that allows you to repeat a set of instructions a fixed number of times, and a **`while`** statement that allows you to repeat a set of instructions as long as some condition holds.

Conditional statements

To get a sense of where conditional statements might come in handy, let's go back to the pothole-filling program presented at the end of section 1.2. Before filling the pothole in the **`fillPothole`** function, Karel might want to check to see if some other repair crew has already filled the hole, which means that there is already a beeper on that corner. If so, Karel does not need to put down a second one. To

represent such checks in the context of a program, you need to use the **if** statement, which has one of the following two forms:

```
if (conditional test) {
   statements to be executed only if the condition is true
}
```

or

```
if (conditional test) {
   statements to be executed if the condition is true
} else {
   statements to be executed if the condition is false
}
```

The first form of the **if** statement is useful when you want to perform an action only under certain conditions. The second is appropriate when you need to choose between two alternative courses of action.

The conditional test shown in the first line of these patterns must be replaced by one of the tests Karel can perform on its environment, as listed in Figure 1-4. Like function calls, tests include an empty set of parentheses, which are part of the Karel syntax. Every test in the list is paired with a second test that checks the opposite condition. For example, you can use the **frontIsClear** condition to check whether the path ahead of Karel is clear or the **frontIsBlocked** condition to see if there is a wall blocking the way. Choosing the right condition requires you to think about the logic of the problem and see which condition is easiest to test.

FIGURE 1-4 Conditions that Karel can test

`frontIsClear()`	`frontIsBlocked()`	Is there a wall in front of Karel?
`leftIsClear()`	`leftIsBlocked()`	Is there a wall to Karel's left?
`rightIsClear()`	`rightIsBlocked()`	Is there a wall to Karel's right?
`beepersPresent()`	`noBeepersPresent()`	Are there beepers on this corner?
`beepersInBag()`	`noBeepersInBag()`	Any there beepers in Karel's bag?
`facingNorth()`	`notFacingNorth()`	Is Karel facing north?
`facingEast()`	`notFacingEast()`	Is Karel facing east?
`facingSouth()`	`notFacingSouth()`	Is Karel facing south?
`facingWest()`	`notFacingWest()`	Is Karel facing west?

You can use the **if** statement to modify the definition of the **fillPothole** function so that Karel puts down a beeper only if there is not already a beeper on that corner. The new definition of **fillPothole** looks like this:

```
function fillPothole() {
   turnRight();
   move();
   if (noBeepersPresent()) {
      putBeeper();
   }
   turnAround();
   move();
   turnRight();
}
```

The **if** statement in this example illustrates several features common to all control statements in Karel. The control statement begins with a ***header,*** which indicates the type of control statement along with any additional information to control the program flow. In this case, the header is

```
if (noBeepersPresent())
```

which shows that the statements enclosed within the braces should be executed only if the **noBeepersPresent** test is true. The statements enclosed in braces represent the ***body*** of the control statement.

It often makes sense to include **if** statements in a function that check whether it makes sense to apply that function in the current state of the world. For example, calling the **fillPothole** function makes sense only if Karel is facing east directly above a hole. You can use the **rightIsClear** test to determine if there is a hole to the south, which is the direction to the right of the one that Karel is facing. The following implementation of **fillPothole** includes this test along with the **noBeepersPresent** test you have already seen:

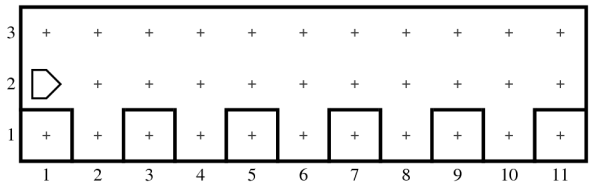
```
function fillPothole() {
   if (rightIsClear()) {
      turnRight();
      move();
      if (noBeepersPresent()) {
         putBeeper();
      }
      turnAround();
      move();
      turnRight();
   }
}
```

As you can see from the spacing used in this example, the body of each control statement is indented with respect to the statements that enclose it. The indentation makes it much easier to see exactly which statements will be affected by the control statement. Such indentation is particularly important when the body of a control statement contains other control statements. Control statements that occur inside other control statements are said to be ***nested.***

Iterative statements

In solving Karel problems, you will often find that repetition is a necessary part of your solution. If you were really going to program a robot to fill potholes, it would hardly be worthwhile to have it fill just one. The value of having a robot perform such a task comes from the fact that the robot could repeatedly execute its program to fill one pothole after another.

To see how repetition can be used in the context of a programming problem, consider the following stylized roadway in which the potholes are evenly spaced along 1st Street at every even-numbered avenue:



Your mission is to write a program that instructs Karel to fill all the holes in this road. Note that the road reaches a dead end after 11th Avenue, which means that you have exactly five holes to fill.

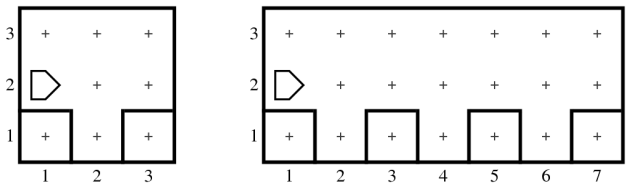
Since you know from this example that there are exactly five holes to fill, the control statement that you need is a **`repeat`** statement, which specifies that you want to repeat some operation a predetermined number of times. The **`repeat`** statement looks like this:

```
repeat (number of repetitions) {
   statements to be repeated
}
```

For example, if you want to change the **`fillTwoPotholes.k`** program so that it solves the more complex problem of filling five evenly-spaced holes, all you have to do is write the following code:

```
function fillFivePotholes() {
   repeat (5) {
      move();
      fillPothole();
      move();
   }
}
```

The **repeat** statement is useful only when you know in advance the number of repetitions you need to perform. In most applications, the number of repetitions is controlled by the specific nature of the problem. For example, it seems unlikely that a pothole-filling robot could always count on there being exactly five potholes. It would be much better if Karel could continue to fill holes until it encountered some condition that caused it to stop, such as reaching the end of the street. Such a program would be more general in its application and would work correctly in either of the following worlds as well as any other world in which the potholes were spaced exactly two corners apart:



To write a general program that works with any of these worlds, you need to use a **while** statement. In Karel, a **while** statement has the following general form:

```
while (conditional test) {
   statements to be repeated
}
```

The conditional test in the header is chosen from the set of conditions listed in Figure 1-4.

To solve the pothole-filling problem, Karel needs to check whether the path in front is clear by invoking the condition **frontIsClear**. If you use the **frontIsClear** condition in a **while** loop, Karel will repeatedly execute the loop until it hits a wall. The **while** statement therefore makes it possible to solve the somewhat more general problem of repairing a roadway, as long as the potholes appear at every even-numbered corner and the end of the roadway is marked by a wall. The following definition of the function **fillRegularPotholes** accomplishes this task:

```
function fillRegularPotholes() {
   while (frontIsClear()) {
      move();
      fillPothole();
      move();
   }
}
```

Solving general problems

So far, the various pothole-filling programs have not been very realistic, because they rely on specific conditions—such as evenly spaced potholes—that are unlikely to be true in the real world. If you want to write a more general program to fill potholes, it should be able to work with fewer constraints. In particular, it does not really make sense to assume that the potholes occur on every other corner. Ideally, there should be no limits on the number of potholes or any restrictions on their spacing. A pothole is simply an opening in the wall representing the road surface.

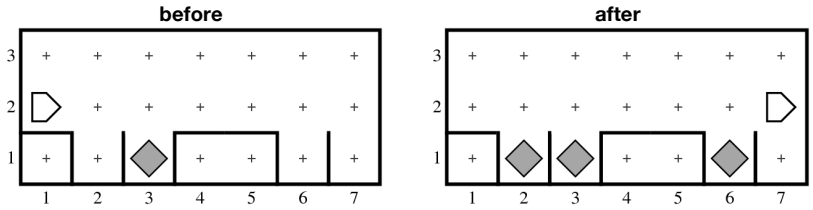
To change the program so that it solves this more general problem requires you to think about the overall strategy in a different way. Instead of having a loop that cycles through each pothole, you need to have it call **`fillPothole`** at every intersection along the roadway.

This strategic analysis suggests that the solution to the general problem might be as simple as the following definition:

```
function fillAllPotholes() {
   while (frontIsClear()) {
      fillPothole();
      move();
   }
}
```

Unfortunately, the solution is not quite so easy. The program as written contains a logical flaw—the sort of error that programmers call a ***bug.*** This book uses the bug symbol on the right to mark functions that contain errors to ensure that you don't accidentally use those examples as models for your own code.

The bug in this example turns out to be relatively subtle. It would be easy to miss, even if you thought you had tested the program thoroughly. In particular, the program works correctly on all the pothole-filling worlds you've seen so far and on many which you haven't. It only fails if there is a pothole in the very last avenue on the street, as illustrated by the following before-and-after diagram:



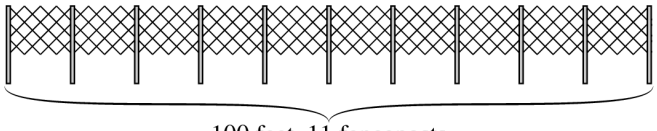
In this example, Karel stops without filling the last pothole. In fact, if you watch the execution carefully, Karel never even goes down into that last pothole to check whether it needs filling. What's the problem here?

If you follow through the logic of the program carefully, you'll discover that the bug lies in the structure of the loop in **`fillAllPotholes`**, which looks like this:

```
while (frontIsClear()) {
   fillPothole();
   move();
}
```

As soon as Karel finishes filling the pothole on 6th Avenue, it executes the **`move`** instruction and returns to the top of the **`while`** loop. At that point, Karel is standing at the corner of 7th Avenue and 2nd street, where it is up against the boundary wall. Because the **`frontIsClear`** test now fails, the **`while`** loop exits without checking the last segment of the roadway.

The bug in this program is an example of a programming problem called a ***fencepost error.*** The name comes from the fact that it takes one more fence post than you might think to fence off a particular distance. How many fence posts, for example, do you need to build a 100-foot fence if the posts are always positioned 10 feet apart? The answer is 11, as illustrated by the following diagram:



The situation in Karel's world has much the same structure. In order to fill potholes in a street that is seven corners long, Karel has to check for seven potholes but only has to move six times. Because Karel starts and finishes at an end of the roadway, it needs to execute one fewer **`move`** instruction than the number of corners it checks.

Once you discover it, fixing this bug is actually quite easy. Before Karel stops at the end of the roadway, all that the program has to do is to make a special-case check for a pothole at the final intersection, as follows:

```
function fillAllPotholes() {
   while (frontIsClear()) {
      fillPothole();
      move();
   }
   fillPothole();
}
```

The complete program appears in Figure 1-5.

FIGURE 1-5 Karel program to fill any number of potholes

```
/*
 * File: FillAllPotholes.k
 * -----------------------
 * This program fills an arbitrary number of potholes in a road.
 */

import "turns";

/*
 * Fills all the potholes up to the end of the road.
 */

function fillAllPotholes() {
   while (frontIsClear()) {
      fillPothole();
      move();
   }
   fillPothole();
}

/*
 * Fills a pothole immediately underneath Karel, if one exists.
 * When you call this function, Karel must be standing just above
 * the pothole, facing east.  When the function returns, Karel
 * will be in its original position above the repaired pothole.
 */

function fillPothole() {
   if (rightIsClear()) {
      turnRight();
      move();
      if (noBeepersPresent()) {
         putBeeper();
      }
      turnAround();
      move();
      turnRight();
   }
}
```

1.4 Stepwise refinement

When you are faced with a complex programming problem, figuring out how to decompose the problem into pieces is usually one of your most important tasks. One of the most productive strategies is called ***stepwise refinement,*** which consists of solving problems by starting with the problem as a whole. You break the whole problem down into pieces, and then solve each piece, breaking those down further if necessary.

An exercise in stepwise refinement

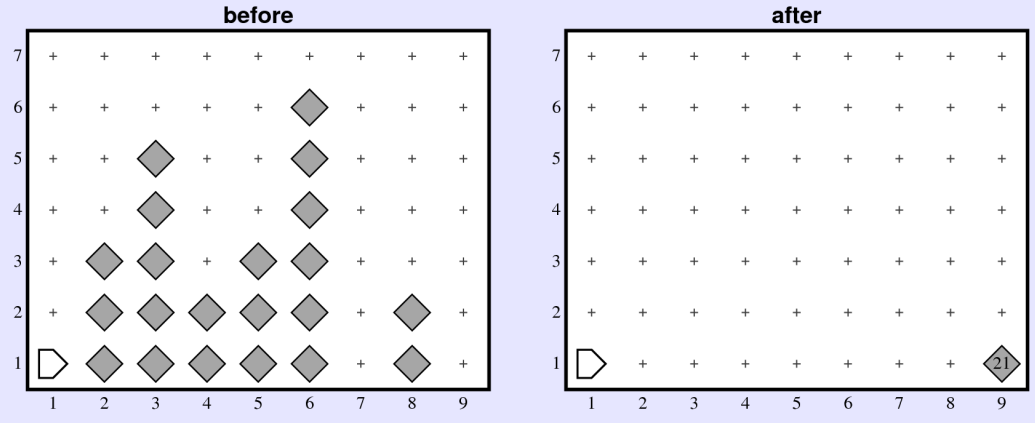
Suppose that Karel is initially facing east at the corner of 1st Street and 1st Avenue in a world in which each avenue may contain a vertical tower of beepers of an unknown height, although some avenues may also be empty. Karel's job is to collect the beepers in each of these towers, put them all back down on the easternmost corner of 1st Street, and then return to its starting position. Figure 1-6 illustrates the operation of this program for one possible world.

The key to solving this problem is to decompose the program in the right way. This task is more complex than the others you have seen, which makes choosing appropriate subproblems more important to obtaining a successful solution.

The principle of top-down design

The central idea in stepwise refinement is that you should start the design of your program from the top, which refers to the level of the program that is conceptually highest and most abstract. At this level, the beeper tower problem is clearly divided into three independent phases. First, Karel has to collect all the beepers. Second,

FIGURE 1-6 Before and after diagrams for the CollectBeeperTowers problem



Karel has to deposit them on the last intersection. Third, Karel has to return to its home position. This outline suggests the following decomposition of the problem:

```
function collectBeeperTowers() {
   collectAllBeepers();
   dropAllBeepers();
   returnHome();
}
```

At this level, the problem is easy to understand. Even though you have not written the code for the functions in the body of **collectBeeperTowers**, it is important to convince yourself that, as long as you believe that the functions you are about to write will solve the subproblems correctly, you will have a solution to the problem as a whole.

Refining the first subproblem

Now that you have defined the structure for the program as a whole, it is time to move on to the first subproblem, which consists of collecting all the beepers. This task is itself more complicated than the problems you have seen so far. Collecting all the beepers means that you have to pick up the beepers in every tower until you get to the final corner. The fact that you need to repeat an operation for each tower suggests that you need to use a **while** loop.

But what does this **while** loop look like? First of all, you should think about the conditional test. You want Karel to stop when it hits the wall at the end of the row, which means that you want Karel to keep going as long as the space in front is clear. The **collectAllBeepers** function will therefore include a **while** loop that uses the **frontIsClear** test. At each position, you want Karel to collect all the beepers in the tower beginning on that corner. If you give that operation a name like **collectOneTower**, you can then write a definition for the **collectAllBeepers** function even though you haven't yet filled in the details. You do, however, have to be careful. To avoid the fencepost problem described on page 17, the code must call **collectOneTower** after the last cycle of the loop, as follows:

```
function collectAllBeepers {
   while (frontIsClear()) {
      collectOneTower();
      move();
   }
   collectOneTower();
}
```

As you can see, this function has the same structure as the **fillAllPotholes** function in Figure 1-5. The only difference is that **collectAllBeepers** calls

collectOneTower where the earlier one called **fillPothole**. These two programs are each examples of a general strategy that looks like this:

```
while (frontIsClear()) {
   Perform some operation.
   move();
}
Perform the same operation for the final corner.
```

You can use this strategy whenever you need to perform an operation on every corner as you move along a path that ends at a wall. If you remember the general strategy, you can quickly write the code whenever you encounter a problem of a similar form. Reusable strategies of this sort come up frequently in programming and are referred to as ***programming idioms*** or ***patterns***. The more patterns you know, the easier it will be for you to find one that fits a particular type of problem.

Coding the next level

Even though the code for **collectAllBeepers** is complete, you can't run the program until you implement **collectOneTower**. When **collectOneTower** is called, Karel is standing either at the base of a tower or on an empty corner. In the former case, you need to collect the beepers in the tower. In the latter case, you can simply move on. This situation at first suggests that you need an **if** statement in which you call **beepersPresent** to see whether a tower exists.

Before you add such a statement to the code, it is worth giving some thought to whether you need to make this test. Often, programs can be made much simpler by observing that cases that at first seem to be special can be treated in precisely the same way as the more general situation. In the current problem, what happens if you decide that there is a tower of beepers on *every* avenue but that some of those towers are zero beepers high? Making use of this insight simplifies the program because you no longer have to test whether there is a tower on a particular avenue.

The **collectOneTower** function is still complex enough that an additional level of decomposition makes sense. To collect all the beepers in a tower, Karel has to climb the tower to collect each beeper, turn around, and then return to the wall that marks the southern boundary of the world. These steps suggest the following code:

```
function collectOneTower() {
   turnLeft();
   collectLineOfBeepers();
   turnAround();
   moveToWall();
   turnLeft();
}
```

The **turnLeft** instructions at the beginning and end of the **collectOneTower** function are critical to the correctness of this program. When **collectOneTower** is called, Karel is always somewhere on 1st Street facing east. When it completes its operation, the program works correctly only if Karel is once again facing east. Conditions that must be true before a function is called are called ***preconditions;*** conditions that must apply after the function finishes are called ***postconditions.***

Finishing up

Although the hard work has been done, a few loose ends still need to be resolved. Four functions—**collectLineOfBeepers**, **dropAllBeepers**, **moveToWall**, and **returnHome**—are as yet unwritten. Fortunately, each of these functions is simple enough to code without any further decomposition. A complete implementation of the **CollectBeeperTowers** program appears in Figure 1-7.

1.5 Algorithms in Karel's world

Although top-down design is a critical strategy for programming, you can't apply it mechanically without thinking about problem-solving strategies. Figuring out how to solve a particular problem generally requires considerable creativity. The process of designing a solution strategy is traditionally called ***algorithmic design.***

FIGURE 1-7 Karel program to collect all the beepers in a set of towers

```
/*
 * File: CollectBeeperTowers.k
 * ---------------------------
 * This program collects all the beepers in a series of towers, deposits
 * them at the easternmost corner on 1st Street, and then returns home.
 */

function collectBeeperTowers() {
   collectAllBeepers();
   dropAllBeepers();
   returnHome();
}

/*
 * Collects the beepers from every tower along 1st Street.
 */

function collectAllBeepers() {
   while (frontIsClear()) {
      collectOneTower();
      move();
   }
   collectOneTower();
}
```

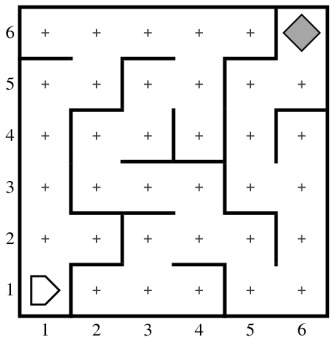
☞

FIGURE 1-7 Karel program to collect all the beepers in a set of towers (continued)

```
/*
 * Collects the beepers in a single tower.
 */

function collectOneTower() {
   turnLeft();
   collectLineOfBeepers();
   turnAround();
   moveToWall();
   turnLeft();
}

/*
 * Collects a consecutive line of beepers.
 */

function collectLineOfBeepers() {
   while (beepersPresent()) {
      pickBeeper();
      if (frontIsClear()) {
         move();
      }
   }
}

/*
 * Drops all the beepers from Karel's bag on the current corner.
 */

function dropAllBeepers() {
   while (beepersInBag()) {
      putBeeper();
   }
}

/*
 * Returns Karel to the corner of 1st Avenue and 1st Street, facing east.
 */

function returnHome() {
   turnAround();
   moveToWall();
   turnAround();
}

/*
 * Moves Karel forward until it is blocked by a wall.
 */

function moveToWall() {
   while (frontIsClear()) {
      move();
   }
}
```

The word *algorithm* comes from the name of a ninth-century Persian mathematician, Muhammad ibn Mūsā al-Khwārizmī, who developed the first systematic treatment of algebra. You will have more of a chance to learn about algorithms and al-Khwārizmī in Chapter 4.

Even before you have a chance to study algorithms in more detail, it is useful to consider a simple algorithm in Karel's domain. Suppose, for example, that you want to teach Karel to escape from a maze. In Karel's world, a maze might look like this:



Karel's job is to navigate the corridors of the maze until it finds the beeper marking the exit. The program, however, must be general enough to solve any maze, and not just the one pictured here.

For most mazes, you can use a simple strategy called the ***right-hand rule,*** in which you start by putting your right hand on the wall and then go through the maze without ever taking your hand off the wall. Another way to express this strategy is to proceed through the maze one step at a time, always taking the rightmost available path. The program that implements the right-hand rule turns out to be easy to implement in Karel and fits in a single function:

```
function solveMazeUsingRightHandRule() {
   while (noBeepersPresent()) {
      turnRight();
      while (frontIsBlocked()) {
         turnLeft();
      }
      move();
   }
}
```

At the beginning of the outer **`while`** loop, Karel turns right to check whether that path is available. The inner **`while`** loop then turns left until an opening appears.

When that happens, Karel moves forward, and the entire process continues until Karel reaches the beeper marking the end of the maze.

Summary

In this chapter, you had a chance to meet Karel, a very simple robot living in a very simple world. Starting off with Karel makes it possible to learn the fundamentals of programming without having to master the many complexities that come with a full-scale programming language. The important points in this chapter include the following:

- Karel the Robot is a *programming microworld* developed in the 1970s by Rich Pattis who was then a computer science graduate student at Stanford. Ever since that time, Karel has welcomed each new generation of Stanford students to the wonders of programming.
- Karel lives in a rectangular world defined by *streets* running from west to east and *avenues* running from south to north. Karel is always positioned on a *corner* marking the intersection of a street and an avenue and must be facing in one of the four standard compass directions (north, east, south, and west).
- Karel's world is surrounded by *walls* around the border and may also contain additional interior walls that block Karel's passage between two corners.
- Karel's world can also contain *beepers,* which Rich Pattis describes as "plastic cones which emit a quiet beeping noise." Beepers exist either on corners or in Karel's beeper bag, both of which can contain an arbitrarily large number of beepers.
- When Karel is shipped from the factory, it knows how to execute only four operations—**move**, **turnLeft**, **putBeeper**, and **pickBeeper**—which are defined in detail in Figure 1-1 on page 4.
- You can extend Karel's repertoire of operations by defining *functions,* which are sequences of operations that have been collected together and given a name. For example, the following function definition gives Karel the power to turn right by executing three consecutive left turns:

```
function turnRight() {
   turnLeft();
   turnLeft();
   turnLeft();
}
```

- The functions **turnRight** and **turnAround** are included in a *library* called **turns**, which you can *import* by including the following line in your program:

```
import "turns";
```

- The best strategy for solving a large problem is to divide it into successively smaller subproblems, each of which is implemented as a separate function. This process is called *decomposition* or *stepwise refinement.*
- The Karel programming language includes *control statements* that fall into two classes. *Conditional statements* allow you to execute other statements only if a particular condition holds. *Iterative statements* allow you to repeat a sequence of statements, either a specified number of times or as long as a condition holds.
- The rules for Karel's control statements appear in the syntax boxes to the right.
- The conditions that Karel can test appear in Figure 1-4 on page 11.
- When you are using iterative statements, it is important to avoid the *fencepost error,* which occurs when you fail to recognize that the number of **`move`** instructions necessary to cover a distance is one less than the number of corners.
- In computer science, an *algorithm* is a solution strategy. If you study computer science, algorithms will be one of the most important topics.

The `if` statement

```
if (condition) {
   statements
}
```

The `if-else` statement

```
if (condition) {
   statements
} else {
   statements
}
```

The `repeat` statement

```
repeat (count) {
   statements
}
```

The `while` statement

```
while (condition) {
   statements
}
```

Review questions

1. In your own words, explain the meaning and purpose of a *programming microworld.*
2. Who created the Karel microworld?
3. What is the etymology of the name *Karel?*
4. Define each of the following aspects of Karel's world: *street, avenue, corner, wall,* and *beeper*.
5. What are the four predefined Karel functions?
6. What are the two functions included in the Karel library named **`turns`**?
7. What is the meant by the strategy of *stepwise refinement?*
8. What statement does Karel offer to execute statements only if some condition applies? What are the two forms of this statement?
9. What two statements does Karel offer for repeating a group of statements?
10. What condition would you use to test whether Karel can move forward from its current position? What condition would you use to test whether there are any beepers on the current corner?
11. What is a *fencepost error?*

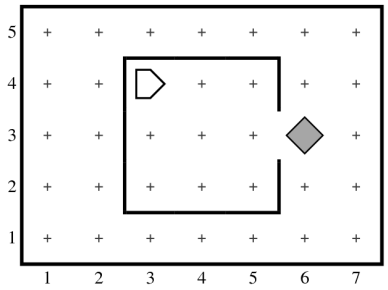
12. What are *preconditions* and *postconditions?*

13. The **`collectLineOfBeepers`** function in Figure 1-7 includes an **`if`** statement that checks the **`frontIsClear`** condition before moving. Why is it important to make this test?

Exercises

1. Only one of the two functions in the **`turns`** library is defined explicitly in this chapter. Write a Karel function that implements the other.

2. Suppose that Karel has settled into its house, which is the square area in the center of the following diagram:



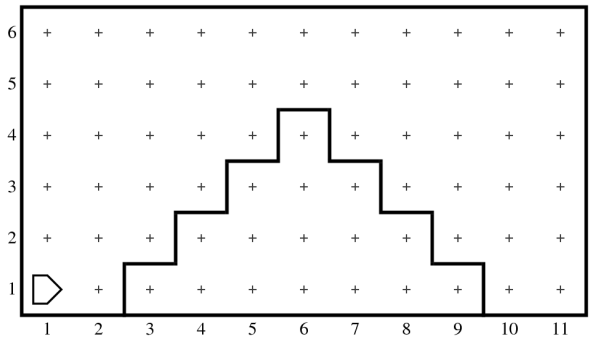
Karel starts off in the northwest corner of its house as shown in the diagram. The problem is to program Karel to collect the newspaper—represented (as all objects in Karel's world are) by a beeper—from outside the doorway and then to return to its initial position.

This exercise is extremely simple and is intended mostly to get you started. You can assume that every part of the world looks just as it does in the diagram. The house is exactly this size, the door is always in the position shown, and the beeper is just outside the door. Thus, all you have to do is write the sequence of statements necessary to have Karel perform the following tasks:

1. Move to the newspaper.
2. Pick it up.
3. Return to its original starting point.

Even though the program requires just a few lines, it is still worth getting at least a little practice in decomposition. In your solution, decompose the program so that it includes a function for each step shown in the outline.

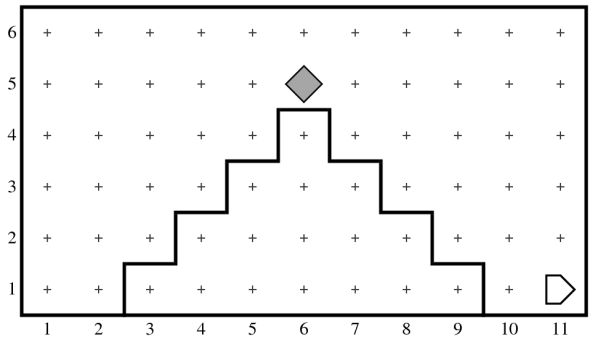
3. Write a program that teaches Karel to climb a mountain exactly like this:



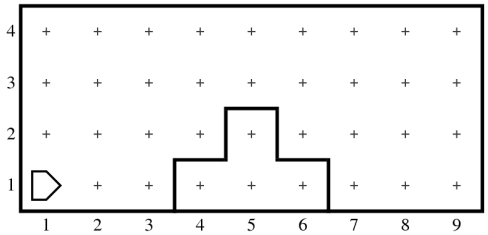
The steps involved are

1. Move up to the mountain.
2. Climb each of the four stair steps to reach the summit.
3. Plant a flag (represented by a beeper, of course) at the top of the mountain.
4. Climb down each of the four stair steps on the opposite side.
5. Move forward to the east end of the world.

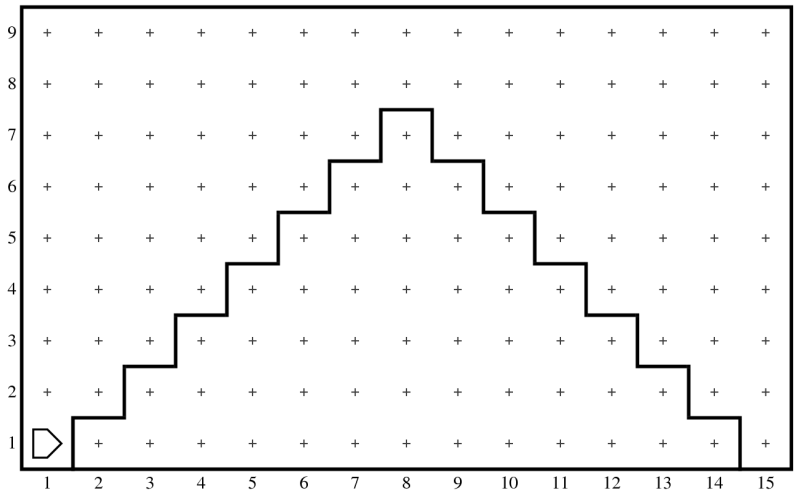
The final state of the world should look like this:



4. Generalize the program you wrote in exercise 3 so that Karel is able to climb a stair-step mountain of any height. Thus, in addition to climbing the mountain in that exercise, it should be able to scale a molehill like

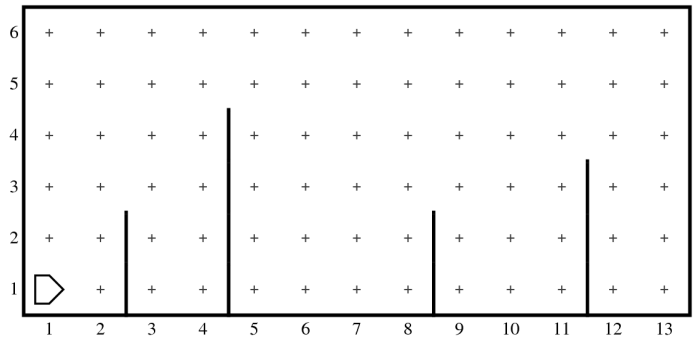


or an Everest-sized peak like

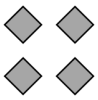


5. “sweet spring is your
time is my time is our
time for springtime is lovetime
and viva sweet love”
—e. e. cummings

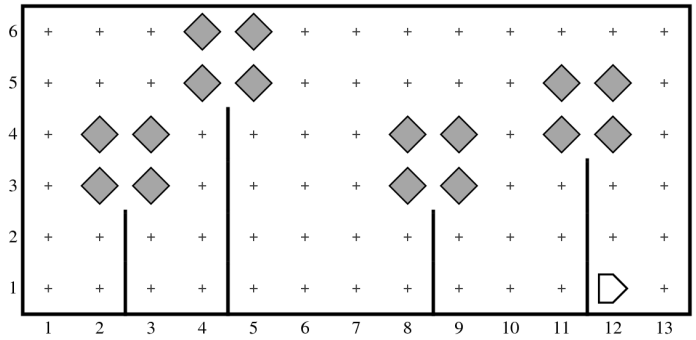
For those who live in colder climates, winter can be a bitter time. The trees have lost their leaves and stand as empty monuments to the ravages of the season, as shown in the following sample world:



In this sample world, the vertical wall sections represent barren tree trunks. Karel’s job is to climb each of the trees and adorn the top of each tree with a cluster of four leaves arranged in a square like this:



Thus, when Karel is done, the scene will look like this:



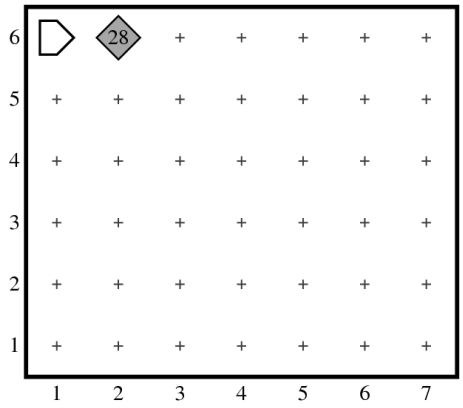
The situation that Karel faces need not match exactly the one shown in the diagram. There may be more trees; Karel simply continues the process until there are no beepers left in the beeper bag. The trees may also be of different heights or spaced differently than the ones shown in the diagram. Your task is to design a program that is general enough to solve any such problem, subject to the following assumptions:

- Karel starts at the origin facing east, somewhere west of the first tree.
- The trees are always separated by at least two corners, so that the leaves at the top don't interfere with one another.
- The trees always end at least two corners below the top, so that the leaf cluster will not run into the top wall.
- Karel has just enough beepers to outfit all the trees. The original number of beepers must therefore be four times the number of trees.
- Karel should finish facing east at the bottom of the last tree.

Think hard about what the parts of this program are and how you could break it down into simpler subproblems. What if there were only one tree? How does that simplify the problem, and how can you use the one-tree solution to help solve the more general case?

6. In this problem, you job is to program Karel to create a calendar by putting down beepers in a pattern that corresponds to the days in a particular month. Karel begins the task in the upper left row of a 7×6 world. On one of the intersections on that row—corresponding to the first day of the month in

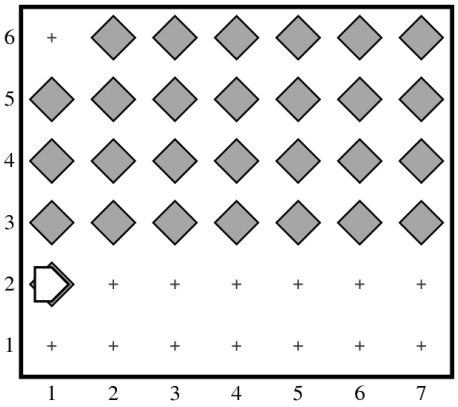
question—there is a beeper pile containing exactly the same number of beepers as there are days in the month. For example, for a February that begins on a Monday and has 28 days, the initial state of the world would look like this:



What Karel needs to do is

1. Walk across the top row to find the beeper pile.
2. Pick up all the beepers.
3. Put the beepers down, one at a time, starting at the intersection on which it found the pile and then continuing across each row in turn until it runs out of beepers.

Thus, given the starting configuration for February, Karel should finish with a world diagram that looks like this:

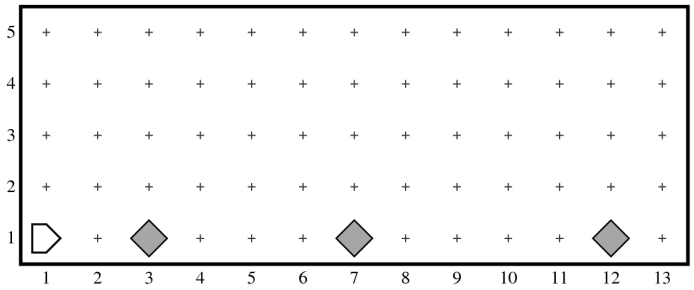


Karel may count on the following facts about the world:

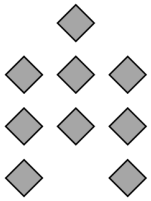
- Karel's world always has seven avenues and six streets.
- Karel begins at the corner of 6th Street and 1st Avenue, facing east, with an empty beeper bag.
- There is a pile with the correct number of beepers somewhere on 6th Street.
- At the end of execution, Karel should be positioned on top of the beeper representing the last day of the month, facing east.

7. More than a decade after Hurricane Katrina, considerable damage remains along the Gulf Coast, and some communities have yet to be rebuilt. As part of its plans to improve the nation's infrastructure, the government has established a new program named Katrina Automated RELief (or KAREL) whose mission is to dispatch house-building robots to repair the damaged area. Your job is to program those robots.

   Each robot begins at the west end of a street that might look like this:

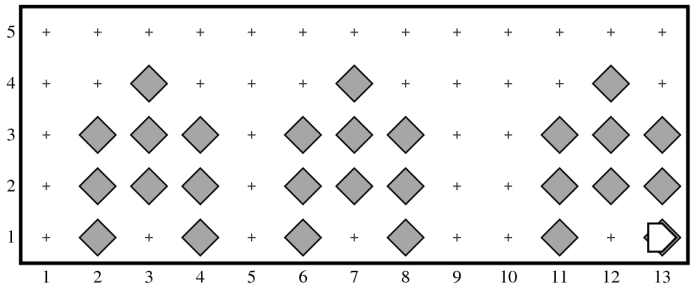


   Each beeper in the figure represents a pile of debris where a house once stood. Karel's job is to walk along the street and build a new house in the place marked by each beeper. Those houses, moreover, need to be raised on stilts to avoid damage from the next storm. Each house, in fact, should look exactly like this:



   The new house should be centered at the point at which the bit of debris was left, which means that the first house in the diagram above will be constructed with its left edge along 2nd Avenue.

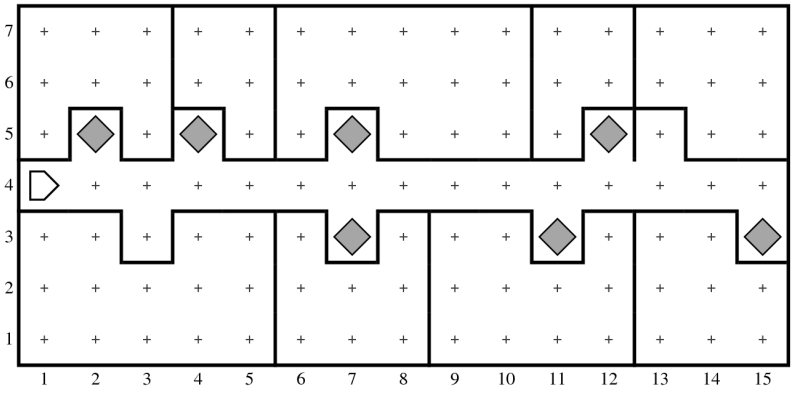
At the end of the run, Karel should be at the east end of the street having created a set of houses that look like this for the initial conditions shown:



In solving this problem, you can count on the following facts about the world:

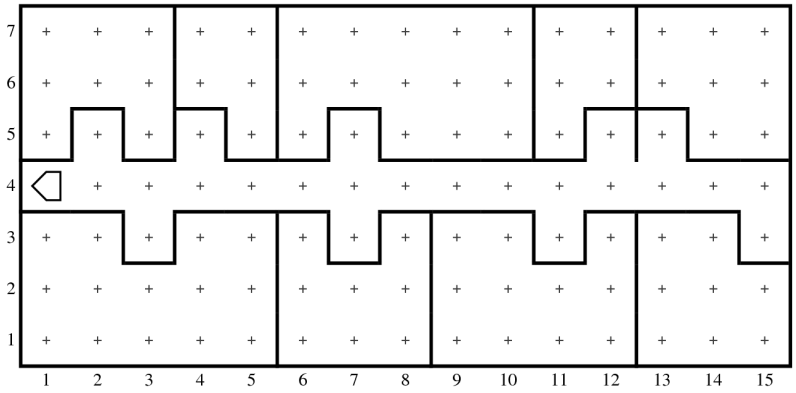
- Karel starts off facing east at the corner of 1st Street and 1st Avenue with an infinite number of beepers in its beeper bag.
- The beepers indicating the positions at which houses should be built will be spaced so that there is room to build the houses without overlapping or hitting walls.
- Karel must end up facing east at the southeast corner of the world. Moreover, Karel should not run into a wall if it builds a house that extends into that final corner.

8. Suppose that it's Halloween, and Karel is going Trick-or-Treating. Karel starts off at the west end of a dead-end street that contains houses on both sides of street, such as the one pictured in the following diagram:



Each house has a front porch at some point along its front side. Karel's mission is to go to each house, step into the porch area, and see if the porch contains a treat, represented by a beeper. If there is, Karel should pick it up. If not, Karel

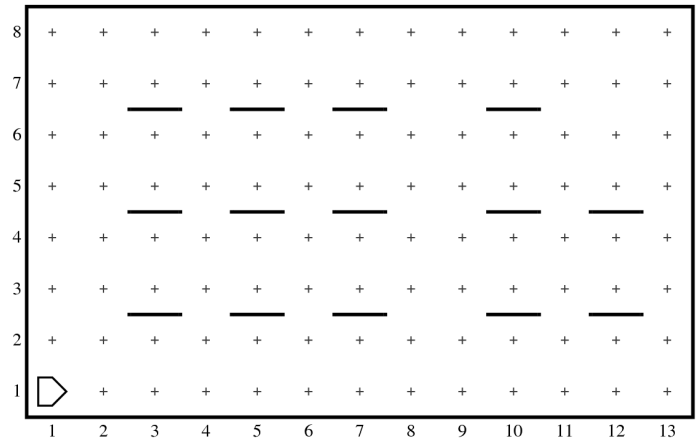
should move on to the next house. Karel must check every porch on both sides of the street and should end up at the original intersection facing in the opposite direction. Thus, after executing your program in the world shown above, Karel should end up in the following position:



Karel may count on the following facts about the world:

- Karel starts out at the west end of a street, facing east, with an empty beeper bag. You do not know how long the street is, but you do know that there are walls closing off each end of the street.
- The houses on the street are packed closely together, with no space between adjacent houses.
- There may be any number of houses on the street. Moreover, the number of houses on one side is not necessarily equal to the number of houses on the other side. The individual houses typically vary in size.
- The side of the house facing the street is a solid wall except for a small porch, which is always one intersection wide. The porch can appear at any point in the front wall.
- Each porch is either empty or marked with a single beeper, representing a Halloween treat.
- At the end of execution, Karel should return to its original intersection at the west end of the street, but should now be facing west.

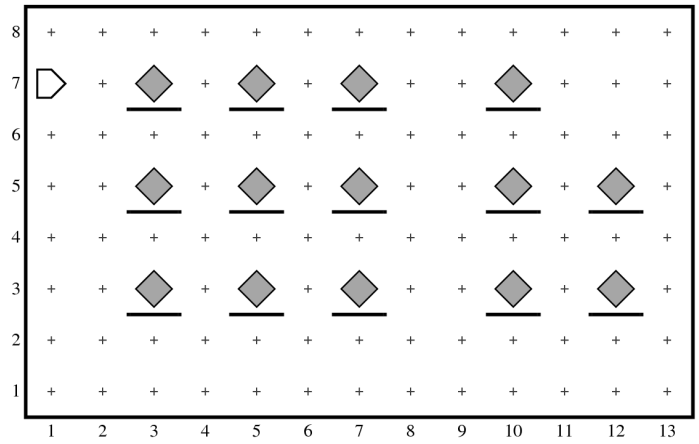
9. Now that Karel has mastered Halloween, it's time to celebrate a different holiday. Karel has decided to deliver beeper valentines to every student in an elementary school class that is using Karel to learn about programming. Karel does not remember exactly how many desks there are in each horizontal row but does remember that there are precisely three rows of desks and that the classroom looks something like the one shown in the following diagram:



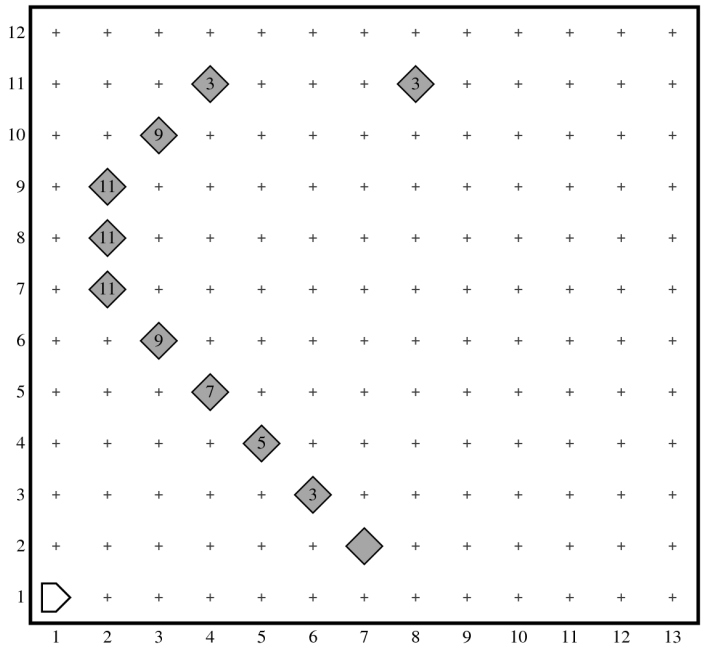
Karel may count on the following facts:

- Karel starts at 1st Avenue and 1st Street, facing east, with an infinite number of beepers in its bag.
- There are exactly three rows of student desks, positioned as shown in the diagram, just to the south of 3rd, 5th, and 7th Streets.
- Karel does not know how many desks there are in each row (which may not all be the same), or how many blank spaces there are between the desks, or how many spaces exist between the desks at the ends of each row and the walls of the classroom. What Karel does know is that each of the desks is exactly one unit wide and that there are no desks right up against the wall.

When Karel is done, all of the desks in the room should have a valentine, as shown in the following diagram:



10. Having heard that programming is at least as much an art as a science, Karel has decided to enroll in a paint-by-numbers class. In this class, Karel is presented with a "canvas" containing piles of beepers, such as those shown in the following diagram:

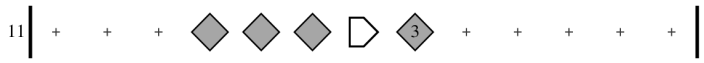


To complete the paint-by-numbers task, all Karel has to do is walk from left to right across each street, pick up each pile of beepers, and then redistribute the beepers from that pile, one at a time, on each successive corner.

To get a sense of how this process works, consider what happens when Karel gets to 11th Street. At the beginning of the row, Karel is standing on the first corner with an empty beeper bag, as follows:

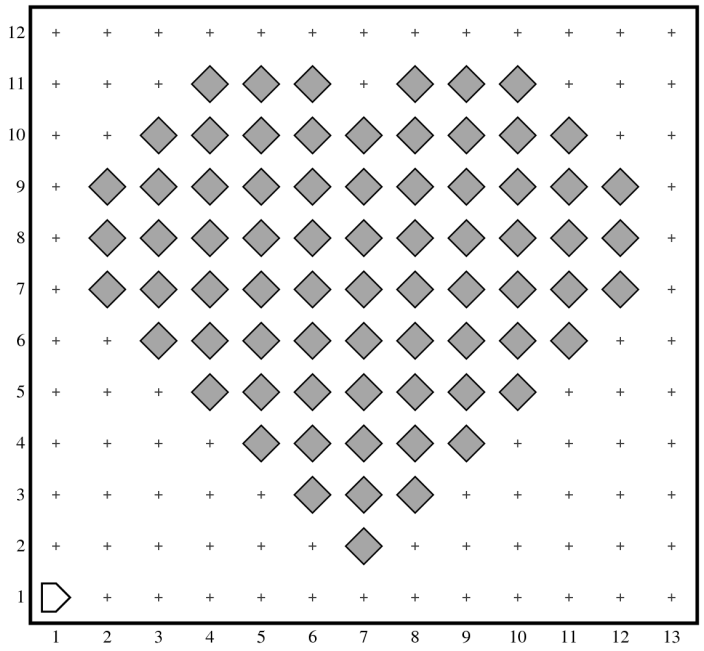
Karel begins by walking down the street looking for a beeper pile. The first one it finds is the pile of three beepers on 4th Avenue. When Karel gets to that corner, it picks up the beepers. This step leaves Karel in the following position with three beepers in its bag:

From here, the next step is to put the beepers down, one at a time, starting with the corner in which the pile was found. Executing this step leads to the following configuration, where Karel again has an empty beeper bag:



Karel then repeats this process for the second beeper pile, ending up in the following position at the end of the row:

The final state of the world should look like this:



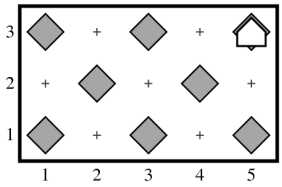
Your job is to write the program that converts a paint-by-number picture of this sort into the corresponding completed masterpiece. In writing the program, Karel can count on the following facts about the world:

- The world contains an arbitrary number of beeper piles but no interior walls.
- The beeper piles never have so many beepers that they cause Karel to run into a wall or another beeper pile.

- Karel always starts facing east in the southwest corner (1st Street and 1st Avenue) with an empty beeper bag.
- Karel must finish execution facing east at the northeast corner of the world.

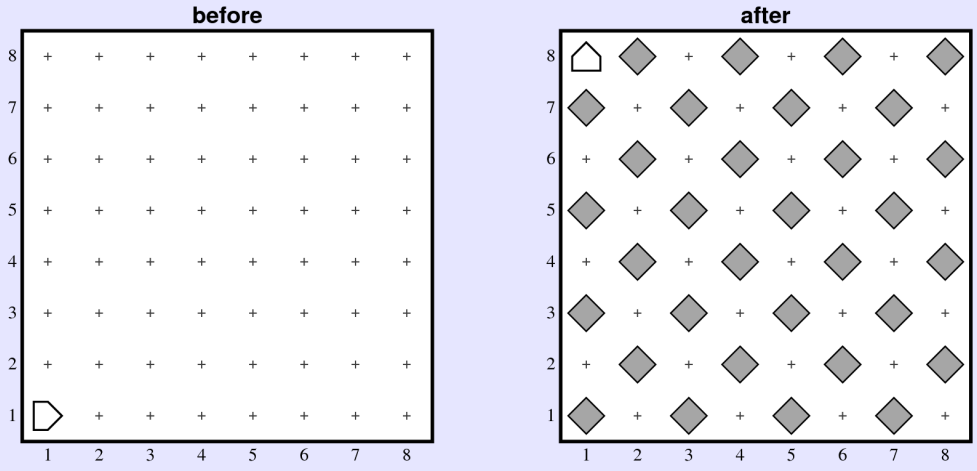
11. In this exercise, your job is to get Karel to create a checkerboard pattern of beepers inside an empty rectangular world, as illustrated in the before-and-after diagram in Figure 1-8.

    This problem has a nice decomposition structure along with some interesting algorithmic issues. As you think about how you will solve the problem, you should make sure that your solution works with checkerboards that are different in size from the standard 8×8 checkerboard shown in the example. Odd-sized checkerboards are tricky, and you should make sure that your program generates the following pattern in a 5×3 world:

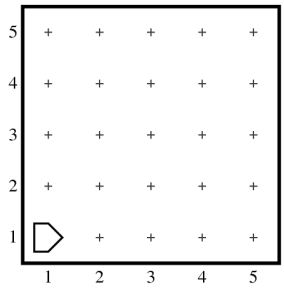


    Another special case you need to consider is that of a world which is only one column wide or one row high.

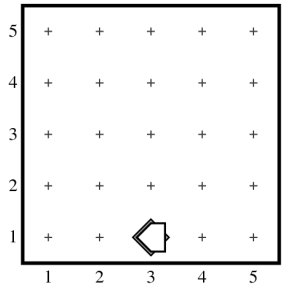
FIGURE 1-8 Before and after diagram for the checkerboard problem



12. Program Karel to place a single beeper at the center of 1st Street. For example, if Karel starts in the world



it should end with Karel standing on a beeper in the following position:



Note that the final configuration of the world should have only a single beeper at the midpoint of 1st Street. Along the way, Karel is allowed to place additional beepers wherever it wants to, but must pick them all up again before it finishes.

In solving this problem, you may count on the following:

- Karel starts at 1st Avenue and 1st Street, facing east, with an infinite number of beepers in its bag.
- The initial state of the world includes no interior walls or beepers.
- The world need not be square, but you may assume that it is at least as tall as it is wide.
- If the width of the world is odd, Karel must put the beeper in the center square. If the width is even, Karel may drop the beeper on either of the two center squares.
- It does not matter which direction Karel is facing at the end of the run.

There are many different algorithms you can use to solve this problem. The interesting part of this problem is to come up with a strategy that works.