CPSC 231 - FALL 2018

KAREL THE ROBOT COMES TO CALGARY

SONNY CHAN, UNIVERSITY OF CALGARY

This handout is an abridged adaptation of

Karel the Robot Learns Java by Eric Roberts

(Stanford University, 2005).

CHAPTER 1

Introducing Karel the Robot

What is Karel?

Karel is a very simple, fictitious robot living in a very simple, fictitious world. Karel the Robot was invented by Richard E. Pattis in 1981,[1] and joins us in this course to help you learn the fundamentals of computer science.

Karel can understand and execute a very small set of predefined commands, but by giving these commands to Karel in various combinations, you can get him to perform some very interesting tasks in his world. The process of specifying commands to Karel is called *programming*. As simple as the commands and the world are, you will soon appreciate that there is an intricate art to doing this.

When you program Karel to perform a task, you must write out the commands in a very precise manner so that the Karel can correctly interpret what you have instructed him to do. The format and particular layout with which you must specify these commands are called *syntactic rules*. The set of commands and the syntactic rules together define the Karel *programming language*.

You may already know that Python is also a computer programming language. Indeed, Python is the language that we will be spending the majority of CPSC 231 working in. We start you off with Karel the Robot rather than with Python for one key reason: Karel's programming language is extremely small in the sense that it has very few commands and rules. We can learn the entirety of the language in just a few hours, leaving us plenty of time to focus on what we would argue is the most important aspect of computer science: *computational problem solving*.

Python is a very intricate real-world programming language with many nuances that can take a lifetime to learn and master. Working with Karel allows us to concentrate on solving problems and to get to the essence of programming without getting caught up in the details of the language. Our version of Karel's programming language is designed to have the same structure and fundamental elements that a Python program has, so that you will have a graceful transition to writing programs in Python later on.

[1] Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, 2nd edition, 1995

Karel's World

Karel the Robot lives in a rectangular world filled with horizontal *streets* running east-west and vertical *avenues* running north-south. The streets and avenues are evenly spaced and form a grid. At any moment in time, Karel stands at the intersection of a street and an avenue, which we naturally call a street corner, and faces one of the four cardinal directions: north, east, south, or west. The streets and avenues are numbered and the intersections are marked by small + symbols as shown in the sample Karel world of Figure 1.

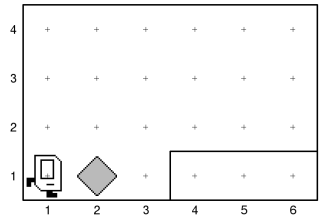

Figure 1: A sample world that contains all elements that Karel will encounter. Here, Karel is standing in his favourite starting location, the intersection of 1st street and 1st avenue, facing east.

You can see several other elements of Karel's world in this example. The grey diamond-shaped object in front of Karel is called a *beeper*. As Richard Pattis describes them, beepers are "plastic cones that emit a quiet beeping noise." Karel can detect the presence or absence of a beeper located at the intersection on which he is standing, and beepers are the only objects in Karel's world that he can manipulate.

The solid lines within the outer rectangle of Figure 1 are *walls*. Walls block Karel's passage and create obstacles that Karel must walk around. As you can see, Karel's world is always surrounded by an outer boundary of walls. The world itself can have different dimensions (number of streets/avenues) depending on the requirements of the task or problem that Karel is working on.

Karel's Commands

The "out-of-the-box" set of instructions that Karel can understand and execute is extremely small. He only knows how to do four things! The commands are as follows:

`move()` instructs Karel to move forward one block. Karel cannot move if there is a wall in his way and will execute an error shutoff in such a situation instead.

`turn_left()` instructs Karel to turn 90 degrees to the left (counter-clockwise when observing from overhead).

`pick_beeper()` instructs Karel to pick up one beeper from the location he is standing at and place it in his beeper bag. If no beeper is present at his location, he executes an error shutoff instead.

`put_beeper()` instructs Karel to take a beeper from his bag and place it at his current location. If he has no beepers left in his bag, he will execute an error shutoff.

You will notice that Karel's two-word commands have the words connected by an underscore (_) character. This is part of the syntactic rules of the language, where each instruction must be a continuous

string of characters, not separated by a space. The empty parentheses that follow each instruction are also part of the syntax of the instruction. For now, you can safely think of it as punctuation that must follow each instruction, much like how we punctuate sentences with periods.

Some of these commands require the world or Karel to be in a certain state before they can complete. For example, Karel must have at least one beeper in his bag before he can put a beeper down. If Karel attempts to execute a command that cannot complete correctly, such as moving through a wall or picking up a non-existent beeper, he will stop and shut himself off for safety reasons. No further commands will execute.

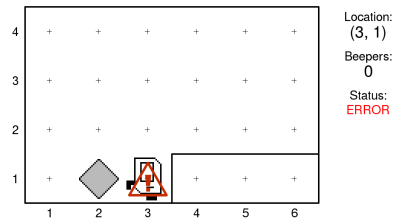

Figure 2: Karel has executed an error shutdown after running into a wall.

It may appear that, with just these four primitive instructions, Karel's capabilities would be rather limited. This couldn't be further from the truth because, as you'll learn later in your studies of computer science, Karel is actually capable of solving incredibly complex and intricate problems. A hallmark of the Karel programming language is the ability to create new instructions by combining existing ones, and teaching Karel new commands is an important part of the programming process that you will learn.

CHAPTER 2

Programming Karel

THERE IS AN AGE-OLD ADAGE in computer science that says computer programming is a skill best learned through experience and practice. Like many other disciplines (art and drawing come to mind), a concept may seem very clear and simple to you when you read about it, or when an instructor shows it to you, but turns out to be very challenging for you to put into practice. As we work through examples of programming Karel here, we encourage you to actively explore, investigate, and tinker with the programs yourself to understand how the system works. Let's get our hands dirty!

Hello Karel

Karel programs are created by writing instructions in a plain text file (named with a `.py` extension), carefully following the syntactic rules of the Karel programming language. We often use the colloquial term "code" to refer to such program text. A listing of a very simple, but complete Karel program is shown below.

beeper-picking-karel.py

```
"""
This simple Karel the Robot example contains instructions that
cause Karel to move forward one block, pick up a beeper, then
move ahead to the next corner.
"""
from karel import *

begin_karel_program()

# instructions for Karel start here
move()
pick_beeper()
move()

end_karel_program()
```

The first several lines of text in the file are surrounded by lines containing exactly three quotes each (`"""`), and are highlighted in

blue in the listing. These lines of text together constitute a *comment* in the code. Any text following a # on a line, such as the one in the middle of the listing, is also a comment.

Comments in the code text are completely ignored by Karel, and have no bearing on what he does or how he interprets the instructions in the program. However, they play a very important role in the practice of programming in that they communicate important information to the readers of the code. Remember that when you write a program, you should not write it for a computer to understand, but write it for *people* to understand. The person reading your code my be a friend, colleague, boss, stranger, or even yourself a few years down the road!

Our version of Karel the Robot actually runs inside the Python environment (which probably comes as no surprise to you). The first statement in the program,

```
from karel import *
```

tells Python to load all of the Karel commands from the `karel` module. Every Karel program you write should start with this statement.

The next instruction, `begin_karel_program()`, tells Karel to start interpreting and executing commands. This is paired with the `end_karel_program()` instruction at the end of the program which tells Karel to stop executing commands. Thus, whenever you write a Karel program, every instruction you want Karel to execute should be sandwiched between the instruction pair `begin_karel_program()` and `end_karel_program()`. We don't actually know what Karel will do if you forget these, or put instructions outside this sandwich!

Finally, after the one-line comment, we see the instructions that we actually want Karel to execute:

```
move()
pick_beeper()
move()
```

Karel with carry out these commands as faithfully as he can, in order from top to bottom. If at any time his present state, or the state of the world, does not allow him to carry out your command, Karel will execute an error shutoff.

If you ran this sample program with Karel in the sample world previously shown, the end result would look like that illustrated in Figure 3. And that's all there is to programming Karel!

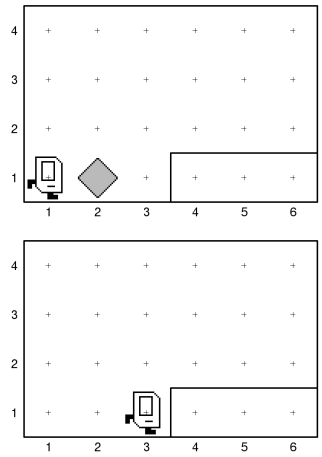

Figure 3: The initial state of our simple example world is shown on top. After running our very first Karel program, you'll see what's just above.

A Slightly More Interesting Problem

Keeping the same example world as before (Figure 1), suppose we would like Karel to perform a slightly more complex task. We want Karel to pick up the beeper located at the corner of 1st street and 2nd

avenue, walk over to 2nd street and 5th avenue, put the beeper down there, and finally move forward to 6th avenue. The initial and desired final state of the world are shown below.

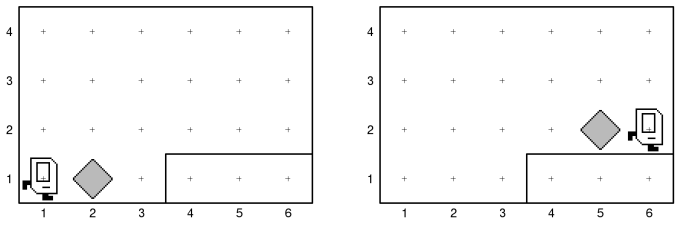

Figure 4: What we want Karel to do in this slightly more interesting problem.

The first part is easy: we can start with the same three commands as before to bring us to 1st street and 3rd avenue with the beeper. From there, we can `turn_left()` to face north, then `move()` onto 2nd street, resulting in the configuration shown on the right.

At this point, you get a chance to practice thinking like a computer scientist. You would now like to turn right so that you can continue along 2nd street, but Karel has no such command available. Perhaps you might be able to accomplish what you want using only what you have available? Of course, it turns out you can achieve the same effect (remember that Karel is incredibly capable), albeit taking a little longer, by turning left three times. To complete the task, you can ask Karel to move forward twice, put the beeper down, and move forward one last time. The final program listing is shown below.

beeper-toting-karel.py

```
"""
This program instructs Karel to pick up a beeper from 1st street
and carry that beeper to the centre of a ledge on 2nd street.
"""
from karel import *

begin_karel_program()
move()
pick_beeper()
move()
turn_left()
move()
turn_left()
turn_left()
turn_left()
move()
move()
put_beeper()
move()
end_karel_program()
```

Defining New Instructions

One drawback you may have noticed when reading the program from the previous program is that, even though we are able to turn left three times to make a right turn, the intent isn't necessarily clear from just looking at the program code. Furthermore, if we ever wanted to program Karel to perform a task that required many right turns, having to write `turn_left` three times repeatedly would start to get rather tedious.

Fortunately, the Karel programming language provides one key mechanism to help us with this problem: the ability to define new instructions, or *subroutines*, for Karel by composing existing ones. This is much more than a convenience, as our ability to name the new instructions descriptively allow us to create and communicate abstractions for ever more complex tasks. Thinking in terms of such abstractions is the cornerstone of the art of computer programming.

We define a new instruction using the following syntax:

`def␣`⟨*new instruction name*⟩`():`
`␣␣␣␣`⟨*commands that make up the instruction*⟩

The new instruction name must be a continuous string of characters that begins with a letter, and can include letters, numbers, and underscores. You can otherwise name the instruction whatever you'd like, though it's obviously best to be descriptive wherever possible. The the new instruction can be composed of a sequence of commands including the primitive Karel instructions, other instructions you previously defined, or the control statements we'll see later. One very important syntactical thing to note is that each instruction that belongs to the sequence of commands in your new instruction must be written indented by exactly four spaces.[1] Spaces are shown as ␣ characters above to make the required syntax clear.

Using this new mechanism, we can define a new `turn_right` instruction as follows:

```
def turn_right():
    turn_left()
    turn_left()
    turn_left()
```

We can now write a revised and more conceptually clear version of `beeper-toting-karel.py` as shown below.

beeper-toting-karel.py

```
"""
This program instructs Karel to pick up a beeper from 1st street
and carry that beeper to the centre of a ledge on 2nd street.
"""
from karel import *
```

[1] Because of this strict requirement on the format of indentation when defining new instructions, or using other statements you'll see later, we strongly recommend using a "proper" code editor to write your Karel programs. A good text editor for programming will automatically insert four spaces when you press the "tab" key on your keyboard. We don't want to get into specific recommendations here, but suffice it to say that using your word processor to write Karel programs is probably not a good idea!

```
def turn_right():
    """
    Turns Karel 90 degrees to the right by turning left thrice.
    """
    turn_left()
    turn_left()
    turn_left()

begin_karel_program()
move()
pick_beeper()
move()
turn_left()
move()
turn_right()
move()
move()
put_beeper()
move()
end_karel_program()
```

Problem Decomposition

With our most powerful tool now in hand, let's see if we can program Karel to solve another problem. If you've ever driven in Calgary after a long, hard winter, you will have noticed that the roads are full of gigantic potholes that can swallow a rabbit whole. Wouldn't it be nice if Karel would help us fill some of those potholes?

Imagine that Karel is travelling west to east along a "road" with a pothole in it, as shown in Figure 5. His job is to fill the pothole, with a beeper of course, so that the final world looks like that shown on the right. If you were to use only the four primitive Karel commands to solve this problem, the body of your program may look like this:

```
begin_karel_program()
move()
turn_left()
turn_left()
turn_left()
move()
put_beeper()
turn_left()
turn_left()
move()
turn_left()
turn_left()
turn_left()
move()
end_karel_program()
```

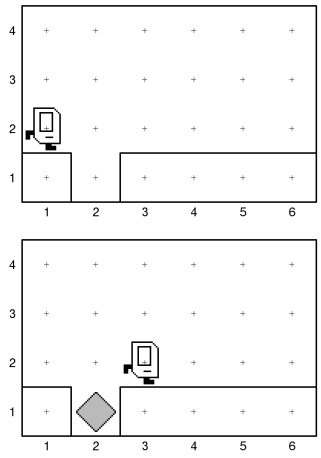

Figure 5: A "road" with a pothole in it is shown on the top. The world immediately below shows how Karel has filled the pothole with a beeper.

If were to apply what we learned previously to define two new instructions, `turn_right()` and `turn_around()`, we could rewrite the main section of the program to look like what's shown on the right. If you read through this revised section, you should notice that it is also easier to interpret and understand what Karel is doing. This is the power of abstraction.

By defining and using new instructions, we can make the code more intuitive and informative, like this:

```
begin_karel_program()
move()
turn_right()
move()
put_beeper()
turn_around()
move()
turn_right()
move()
end_karel_program()
```

The process of breaking a large problem down into smaller pieces that are easier to solve is called problem *decomposition*. As you can probably guess, you'll be doing a lot of this over the semester, and hopefully throughout your degree. We refer to the component parts of the larger problem as *subproblems*.

Looking at the pothole problem a bit more, you might think that it would be most reasonable to decompose it into the following tasks:

1. Move up to the pothole.
2. Fill the hole by dropping a beeper into it.
3. Move to the next street corner.

A corresponding, descriptive program body would then ideally look like what's shown on the right.

A program body that corresponds directly to our tasks may look like this:

```
begin_karel_program()
move()
fill_pothole()
move()
end_karel_program()
```

We can certainly achieve this by putting the commands required to fill a pothole into the definition of a new instruction. A final program using this decomposition may look like the listing below.

pothole-filling-karel.py

```
"""
This program instructs Karel to fill a single pothole on 2nd avenue.
"""
from karel import *

def turn_right():
    turn_left()
    turn_left()
    turn_left()

def turn_around():
    turn_left()
    turn_left()

def fill_pothole():
    turn_right()
    move()
    put_beeper()
    turn_around()
    move()
    turn_right()

begin_karel_program()
move()
```

```
fill_pothole()
move()
end_karel_program()
```

Which Decomposition Should I Choose?

The problem decomposition we chose to solve the pothole problem in the previous section seems to be pretty intuitive and reasonable enough. However, you can imagine that it's not the only decomposition that would solve the pothole problem correctly. For example, you might have considered this like:

1. Move forward and fill the pothole underneath.
2. Move to the next street corner.

Or perhaps another decomposition with the following steps:

1. Move forward and down into the pothole.
2. Fill the pothole that Karel is standing in.
3. Move up out of the pothole and forward.

We can see that each of these strategies would ultimately achieve the same goal. Which one should we choose to use? Is there a "best" choice?

In general, choosing how to decompose a larger problem will be one of the most difficult design decisions you will need to make as a programmer. Typically, the more complex a problem is, the more challenging it will be to choose an appropriate decomposition of it. The following guidelines may help:

- *Each subproblem should address a conceptually simple task.* No matter how many, or how few, commands go into the definition of a new instruction, it should end up accomplishing a conceptual task that is itself easy to describe. If it's easy for you to describe the effect of a new instruction with a simple and concise name, you probably have a good decomposition.
- *Each subproblem should address its task in as general a context as possible.* We would like to have instructions that are useful in a variety of different situations. A decomposition that breaks the program up into instructions that solve common subproblems in a variety of related situations is likely superior to a decomposition that only works in a very specific and particular situation.

Given the two guidelines above, we can see that our final choice of decomposition from the previous section is preferable to the alternatives discussed here.

CHAPTER 3

Control Statements

THE ABILITY TO DEFINE new instructions for Karel gave us the incredibly powerful tool of problem decomposition, but even with this tool, Karel's capabilities are quite limited. New instructions are essentially just shorthand for a longer sequence of commands, and Karel ultimately just executes a sequence of his four built-in commands in a fixed order, oblivious to the world around him. A nicely decomposed program is functionally no different from a the monolithic sequence of the same instructions that Karel ends up following.

In order to fully unlock Karel's problem-solving potential, we must allow Karel to observe the state of his world and to use this information to decide which set of instructions to execute. Statements that affect the order in which a program executes commands are called *control statements*. The Karel programming language has two types of control statements:

Conditional statements indicate that a certain set of instructions should be executed only if a particular condition is true.

Iterative statements specify that a certain set of instructions should be executed repeatedly in what we call a *loop*, either for a predetermined number of times, or while a particular condition is true.

Control statements can be placed wherever you may write any other Karel instruction (*e.g.* in the main instruction block of your program or under the definition of a new instruction). These control statements constitute the final element of the Karel programming language. We'll briefly examine the use of each type of control statement in the sections that follow.

Condition Tests

Conditional statements provide the means for Karel to examine the state of his world and react accordingly. Let's revisit the pothole filling problem as a simple example of where this may be helpful. Suppose that another robot—perhaps an inferior model—has already

come by to repair potholes on the road. Some potholes may already have been filled, but others might have been missed, and it is Karel's job to ensure that every pothole is filled. Thus, before Karel takes action to fill a pothole with a beeper, he would like to check if there is already a beeper filling the pothole to avoid placing a second one.

There are a total of nine different conditions that Karel can test, including the one he needs for the new pothole problem. These are shown in Table 1. At any moment while Karel is executing your program, each of these conditions is either true or false.

Condition	**Meaning**
`front_is_clear()`	No wall is in front of Karel
`left_is_clear()`	No wall is blocking Karel on the right
`right_is_clear()`	No wall is blocking Karel on the left
`beepers_present()`	There is a beeper where Karel is standing
`beepers_in_bag()`	Karel has at least one beeper left in his bag
`facing_north()`	Karel is facing north
`facing_east()`	Karel is facing east
`facing_south()`	Karel is facing south
`facing_west()`	Karel is facing west

Table 1: The nine conditions that Karel can test about himself or the world around him.

Like the names of the four built-in Karel commands, words in the condition names are separated by underscores. The empty parentheses following each condition are part of the syntax, and show that the condition test is being applied, just like executing an instruction. You can also test the opposite of any of these conditions. For example,

```
not front_is_clear()
```

evaluates true if Karel is being blocked by a wall in front of him. Note that you must separate the word "not" from the base condition with a space, and not an underscore.

In the Karel programming language, the `if` statement allows you to specify a set of instructions for Karel to execute only if a certain condition is true. Its syntax is as follows:

```
if ⟨condition⟩:
    ⟨statements executed only if condition is true⟩
```

The ⟨*condition*⟩ can be any of those listed in Table 1, or its opposite. Like the definition of new instructions, every statement that belongs to the group of statements executed only if the condition is true must be written indented by four spaces (shown as ␣ above).

Using the `if` statement, we can update Karel's pothole filling program to account for the other robot's work as follows:

```
def fill_pothole():
    turn_right()
    move()
```

```
    if not beepers_present():
        put_beeper()
    turn_around()
    move()
    turn_right()
```

Note that the `put_beeper()` command is now written indented by eight spaces total. It belongs within the `if` conditional, requiring four spaces, which itself appears within the definition of the `fill_pothole()` instruction, requiring an additional four spaces.

Even within this simple example, you can begin to see the *nested* control structure that most real-world programs take. If you were wondering whether or not an `if` statement can appear within the statement set of another `if` condition test, the answer is yes, of course, and this happes all the time. Suppose you wanted to add one more test to ensure that Karel has a beeper before filling the pothole with it, so that doesn't execute an error shutoff if he runs out of beepers on the job, you may write something like this:

```
if not beepers_present():
    if beepers_in_bag():
        put_beeper()
```

Sometimes you may want to divide your program into two different courses of action based on a condition to solve a particular problem. For example, you may want a certain set of instructions to executed only if a condition is true, and a different set of instructions to be executed if the condition is false. There is an extended form of the `if` statement that allows you to do exactly that:

```
if␣⟨condition⟩:
␣␣␣␣⟨statements executed only if condition is true⟩
else:
␣␣␣␣⟨statements executed only if condition is false⟩
```

Again, note the careful way in which the statements must be indented when written.

Repetition

Many problems you will solve with Karel will require repeating sets of instructions in various ways. After all, isn't that what robots and computers excel at? Unlike humans, they never seem to complain when you ask them to repeat a task hundreds, thousands, or even millions of times!

We can use the pothole filling example yet again to demonstrate the ease and power of repetition. More often than not, you'll see our roads in spring covered with many more than a single pothole. What good is a program can only fill one pothole? Consider the first

scenario below, where Karel is standing on a street full of potholes (which happen to be spaced evenly apart) to fill.

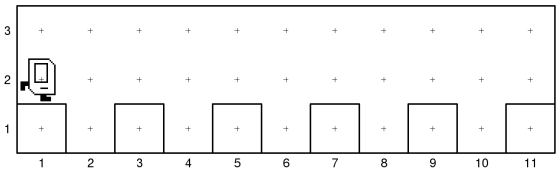

Figure 6: A long road with many potholes for Karel to fill.

If you've been following our examples, you should notice that this problem can be solved by repeating the instructions in the main body of `pothole-filling-karel.py` five times. And if you're also cringing at the thought of having to type out those instructions five times over, that's a very good sign! Modern text editors do give us a pretty convenient shortcut for doing this, via the "copy" and "paste" commands, but this would break one of our golden rules of programming: *Never copy and paste code!*[1]

[1] Except for some special situations, which do occur pretty often. So it's more of a recommendation. But we firmly believe that you will be adopting very good habits if you can refrain from copy-pasting code.

The first kind of iterative statement, the `for` statement, help us to do exactly what we want in this case. Its syntax is as follows:

```
for i in range(⟨count⟩):
    ⟨statements repeated count times⟩
```

The sequence of statements appearing indented under the `for` will be repeated ⟨*count*⟩ number of times. The same four-space indentation syntax rules apply here, though we haven't explicitly shown the spaces this time. The repetition ⟨*count*⟩ must be a positive integer.

You may also be wondering why the syntax of a `for` statement appears much less simple and graceful than the others you've learned. The reason is that other programming languages use the same syntax to produce more complex and intricate types of iteration than Karel needs or understands. When programming Karel, just copy the syntax shown exactly and replace ⟨*count*⟩ with a number.

One limitation with the `for` statement is that it is only useful when you know in advance the number of repetitions you need. When designing programs to solve real-world problems, you might often want to repeat instructions until a certain condition of the world is met. For example, it seems an unlikely scenario that Karel can count on having exactly five potholes to fill. Wouldn't it be nice to write a program that instructs Karel to fill potholes until he gets to the end of the road? Such a program would also work, without modification, on other roads with potholes spaced two avenues apart, like the one on the right.

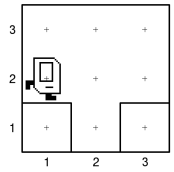

Figure 7: Karel should also be able to fill the pothole in this short little road by running the same program.

The `while` control statement provides us exactly the functionality we need to accomplish this. Its syntax is as follows:

```
while ⟨condition⟩:
    ⟨statements repeated while condition is true⟩
```

This instructs Karel to repeat a sequence of instructions while a specified ⟨*condition*⟩ is true. The usual indentation syntax rules apply. Karel tests the condition before executing the first instruction in the repeated set each time. If ever the condition evaluates false, including on the first test, Karel skips over the statements indented under the `while` and continues merrily along with the rest of your program.

In our pothole examples, Karel reaches the end of the road when the eastern wall of the world is directly in front of him. Thus, we can use the `front_is_clear()` condition to detect when Karel has made it to the end of the road. Incorporating these control statements, we can now write the program to fill any number of potholes spaced two avenues apart, shown in the listing below. We call such a program more *general* (as opposed to specific) than the ones we studied in the previous chapter because the same program can be used to solve the problem described in many worlds with different configurations. Generality is usually a very desirable trait.

road-repair-karel.py

```
"""
This program instructs Karel to fill potholes spaced two avenues
apart along a road until he reaches the end.
"""
from karel import *

def turn_right():
    turn_left()
    turn_left()
    turn_left()

def turn_around():
    turn_left()
    turn_left()

def fill_pothole():
    turn_right()
    move()
    if not beeper_is_present():
        put_beeper()
    turn_around()
    move()
    turn_right()

begin_karel_program()
while front_is_clear():
    move()
    fill_pothole()
    move()
end_karel_program()
```

Solving General Problems

Programs that can solve general problems in an elegant and efficient manner are in many ways the holy grail for computer programmers. In the real world, the activities of testing, debugging, and maintaining program code often take up much more time than it does to create the code itself. Thus, the more times and situations under which

you can use a well tested and verified program, the better.

Looking at our pothole problem one last time, as general as it already is, it still relies on some specific conditions that are unlikely to be true. Namely, the requirement that potholes are exactly two avenues apart seems rather limiting. We would ideally want to write a pothole filling program that works with as few specific requirements as possible. For example:

- *The program should work correctly on roads of any arbitrary length.* Although we do need a way to know when we've reached the end.
- *The potholes may occur at any position on the road.* A pothole is identify by an opening in the wall that represents the surface of the road below Karel, but there should be no restrictions on their number or spacing.
- *Existing potholes may already have been repaired.* Karel should not place an additional beeper in potholes that are already filled.

Writing the general version of the program requires us the rethink the strategy. Instead of having the `while` loop in the main part of the program repeatedly execute `fill_pothole()`, we should get it to check first whether there is a pothole to the south of Karel. The required change in the code isn't much, and you can rewrite the main part of the program as shown on the right.

```
begin_karel_program()
while front_is_clear():
    if right_is_clear():
        fill_pothole()
    move()
end_karel_program()
```

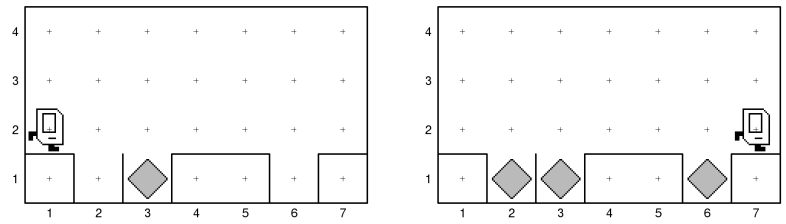

Figure 8: Karel fills a bunch of irregular potholes on the road.

If you run the program with this modification on the world shown on the left of Figure 8, you should get the output shown on the right. Everything looks fine and dandy, right? What would happen if you ran the program in the very similar world shown in Figure 9? You'll notice that Karel missed a pothole!

This is an example of a commonly-encountered programming mistake which we call an "off-by-one" error, or a "fencepost" error. The latter term comes from the analogous problem of planting posts to erect a fence. If you wanted to build a 10 metre fence that requires a post to support it every metre, how many fenceposts do you need in total? (Ten, right? No, we actually need eleven—don't forget the last one on the end!) This example also teaches us the importance of testing our programs in a variety of situations that they are supposed to work in.

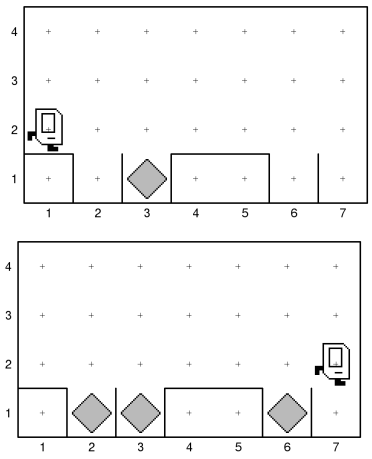

Figure 9: Oops! Karel missed a pothole!

Karel's problem is essentially the same as the fence problem. To repair a road seven blocks long, he needs to check for seven potential potholes, but only needs to move forward six times to do so. The best way to fix our program is to introduce one more instruction that abstracts the task of checking for a pothole, then rewrite the main section of the program to reflect our fence building strategy. Below is a listing of our final pothole repair program.

road-repair-karel.py

```
"""
This program instructs Karel to fill potholes spaced two avenues
apart along a road until he reaches the end.
"""
from karel import *

def turn_right():
    turn_left()
    turn_left()
    turn_left()

def turn_around():
    turn_left()
    turn_left()

def fill_pothole():
    turn_right()
    move()
    if not beeper_is_present():
        put_beeper()
    turn_around()
    move()
    turn_right()

def check_for_pothole():
    if right_is_clear():
        fill_pothole()

begin_karel_program()
while front_is_clear():
    check_for_pothole()
    move()
check_for_pothole()     # final "fencepost"
end_karel_program()
```

CHAPTER 4

Stepwise Refinement

EVEN WITH THE SIMPLE PROBLEMS we've encountered thus far, you can see there are many different programs we can write that would all solve the problem correctly. Computer programming is often called an "art"—it's actually not that different from essay writing, really—and one key aspect of this art is choosing which of the many correct alternatives to employ for the problem.

When programming to solve computational problems, we would naturally want to write code that is concise, correct, simple to read, and easy to understand. But aside from going with our gut instincts (which is usually good), how might we accomplish this?

As you might expect, programmers have contemplated this for decades. Much of our understanding and knowledge on the subject has been collected and studied under the discipline of *software engineering*. Of course, we won't be able to teach you all about software engineering in a day, or even just a fundamentals, but there is one tried-and-true strategy that we can use right now that will get you a long way. We refer to this strategy as *top-down design* or *stepwise refinement*.

The idea of stepwise refinement is to first look at the problem you're trying to solve as a whole. You make a concise plan of how to solve the problem by breaking it down in to a handful of steps. Then you look at each step in turn, breaking it down into smaller and simpler steps again if necessary, until you've managed to organize and describe your solution in terms of basic instructions. If it seems intuitive, you're right, it is, and you've probably applied this strategy elsewhere in your life. The purpose of this chapter is to learn and practice the technique of stepwise refinement by programming Karel to accomplish a slightly more complex task.

A Beeper Collecting Problem

We imagine that by now you're dreadfully tired of filling potholes.[1] Let's teach Karel to solve a new problem—and a very useful one—of picking things up and putting them away neatly. Karel is in a world

[1] If you're not tired of potholes by now, we should either praise you for your tenacity or reprimand you for skipping the previous chapters.

full of towers of beepers like that shown on the left in Figure 10. Perhaps Karel was playing with his beeper collection, building a city of tall towers with them, but now it's time to put all the beepers away. (If you ever played with LEGO® when you were young, you'd probably understand Karel's situation exactly.)

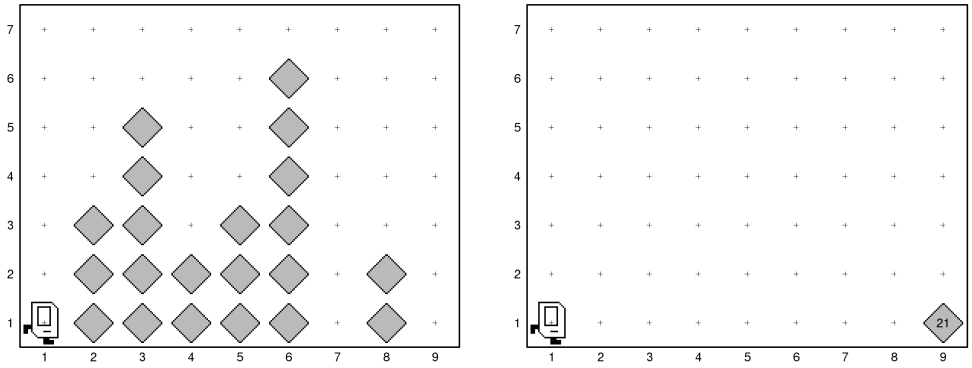

Figure 10: Towers of beepers for Karel to collect and put away neatly in a pile in the corner.

Each avenue in this world contains a tower of beepers starting on 1st street, though some avenues may be empty (*e.g.* 1st, 7th, and 9th avenues in the above example). We want to program Karel to pick up all the beepers from these towers and place them in a pile at the easternmost corner of 1st street, then return back to where he started. When Karel is finished, the world should like like what shown on the right in Figure 10.

Just as we saw before, the key to solving this problem is to come up with a good decomposition of the program. We'll do this decomposition as an exercise in stepwise refinement.

The Principle of Top-Down Design

The main principle of stepwise refinement is to start the design of your program from the conceptually highest and most abstract level. Looking at the description of Karel's task at hand, we can see that it is divided into three distinct stages:

1. Collecting all the beepers.
2. Putting the beepers down on the easternmost corner of 1st street.
3. Returning to the home position.

If we were to map these steps directly into our code, the main section of the program, where Karel starts executing instructions, it might look like the listing shown on the right. The result is that, at least so far, we have a nice, descriptive program that is easy to understand. Of course, the instructions we are using don't exist yet, but we have a mechanism for creating them, and we'll do that next.

Right now, it's important to look at the decomposition and convince yourself that each subproblem is well defined, and that you

Following these stages, the main section of our program may look like this:

```
begin_karel_program()
collect_all_beepers()
drop_all_beepers()
return_home()
end_karel_program()
```

will be able to write a subroutine to solve that problem correctly. If that is indeed the case, then you should have a solution to the problem as a whole.

Refining a Subproblem

Practicing stepwise refinement, our next job is to develop a solution to the first subproblem: that of collecting all the beepers. This task isn't perfectly straightforward itself, since there are multiple towers of beepers to collect. We can apply top-down design once again to develop a solution to this subproblem. This is the "refinement" part of stepwise refinement.

Let's think about what `collect_all_beepers()` must do. We must instruct Karel to move along 1st street, collecting towers of beepers at each intersection, until[2] he reaches the end of the street. Repeatedly collecting towers tells us that an iterative statement will likely be helpful. At this level, our new instruction for collecting all the beepers may have a structure like this:

[2] The presence of the word "while", or its cousin "until", in the task description is usually a strong sign to employ a `while` loop in your code.

```
def collect_all_beepers():
    while front_is_clear():
        collect_one_tower()
        move()
    collect_one_tower()  # watch for the last fencepost!
```

Now we've left ourselves with yet another undefined instruction, `collect_one_tower()`. But we know not to worry about that now. In fact, it's a good thing! Think about the subproblem of collecting one tower of beepers to yourself that it's well defined, and that you can create a new instruction to solve it. Then apply stepwise refinement.

Going Deeper

Let's look at one more subproblem and see if we can get the bottom of it. If we've understood our task and defined our subproblem right, when Karel is instructed to collect one tower, he is either standing on a beeper at the base of a tower, or on an empty street corner. In the former case, Karel must collected the beepers in the tower. In the latter case, he can simply move on. This sounds suspiciously like an application of the `if` control statement, and we might want to write something like

```
if beepers_present():
    collect_actual_tower()
```

Before adding such a statement to the code, it's best to think about whether or not we actually need to make this condition test. Specific conditions, such as avenues without towers in this example, often

appear in problems that look like they need to be treated as special cases. Sometimes, if you think about it carefully, you may be able to frame the problem in such a way that a special case is not needed. It's wroth putting thought into such cases because it may greatly simplify the program design, and as we've already learned, simplicity and generality are virtues.

With our current problem, the key insight is to think of *every* avenue as having a tower of beepers on it, but that the tower may be zero beepers high. Then if we create the `collect_one_tower()` instruction to work for this definition of a beeper tower, we no longer have to test for empty avenues. Collecting one such tower is still a complex task, so we can decompose it one more time into the following steps:

1. Turn left to face the beepers along the tower.
2. Collect the beepers in the tower, stopping when no more beepers are found.
3. Turn around to face the bottom of the world.
4. Return to the wall at ground level (1st street).
5. Turn left to face east and be ready to move to the next tower.

Once again, these steps provide a good model for how we might write the tower-collecting instruction.

```
def collect_one_tower():
    turn_left()
    collect_line_of_beepers()
    turn_around()
    move_to_wall()
    turn_left()
```

Though the instructions are getting progressively simpler—hopefully you noticed that!—we still haven't reached the end of our refinement process. We don't have definitions for `move_to_wall()` or `collect_line_of_beepers()` yet, but these last ones can be composed from primitive Karel commands. At this point, we trust that you're very capable of defining these subproblems and instructions to solve them correctly, and leave them as an "exercise for the reader".

Pre-Conditions and Post-Conditions

When we wrote the code to solve some of the subproblems in this chapter, you may have noticed that the new instructions would not work correctly if certain things are not true. For example, the `collect_one_tower()` instruction would not do what it was intended to if Karel is facing the wrong way when he starts executing the commands within that instruction. When we design a new instruction, we often must make certain assumptions about the state of Karel or that of the world he is in. In this example, `collect_one_tower()` assumes that Karel is facing east at the start of the instruction.

Conditions that must hold true for an instruction to work correctly are called *pre-conditions*. Similarly, conditions that will be true after an instruction completes are called *post-conditions*. For example, a pre-condition for the built-in command `put_beeper()` is that Karel has at least one beeper left in his bag. Post-conditions of

`collect_one_tower()` are that a line of beepers to the north of Karel has been picked up, and Karel is facing east.

Whenever a new instruction you define requires such conditions for it to function correctly, it is a very good idea to document these conditions. Many programming errors are caused by mismatches or misunderstandings of pre- and post-conditions when instructions or subroutines are executed. If you make a habit of carefully documenting pre- and post-conditions when you define new instructions, you cave yourself a lot of time, not to mention headaches, down the road.

The Complete Program

At this point, we've completed most of the hard work of using stepwise refinement to decompose the problem. There are still a few instructions like `drop_all_beepers()` and `collect_line_of_beepers()` that we have yet to define, but we're confident that by now you have all the knowledge of skills needed to write very good code for these yourself. You can compare your definitions with those in our full program that solves this problem, shown below. Note how pre-conditions and post-conditions were documented where we thought they'd be helpful.

beeper-collecting-karel.py

```
"""
This program instructs Karel collect all the beepers in a series of
vertical towers and deposit them at the southeasternmost corner.
"""
from karel import *

def turn_right():
    turn_left()
    turn_left()
    turn_left()

def turn_around():
    turn_left()
    turn_left()

def collect_all_beepers():
    """
    Move along 1st street and collect beepers in every tower.
        Post-condition: Karel is on the easternmost corner of
                        1st street, fasting east.
    """
    while front_is_clear():
        collect_one_tower()
        move()
    collect_one_tower()  # watch for the last fencepost!

def collect_one_tower():
    """
```

```
    Collects all beepers in a single vertical tower to the north.
        Pre-condition:  Karel is on 1st street, facing east.
        Post-condition: Karel is on the same corner, facing east.
    """
    turn_left()
    collect_line_of_beepers()
    turn_around()
    move_to_wall()
    turn_left()

def collect_line_of_beepers():
    """
    Collect a consecutive line of beepers in front of Karel.
    """
    while beepers_present():
        pick_beeper()
        if front_is_clear():
            move()

def drop_all_beepers():
    """
    Drop all beepers Karel is carrying at his current location.
    """
    while beepers_in_bag():
        put_beeper()

def return_home():
    """
    Return Karel to his favourite home position.
        Pre-condition:  Karel is on 1st street, facing east.
        Post-condition: Karel is on 1st avenue and 1st street,
                        facing east.
    """
    turn_around()
    move_to_wall()
    turn_around()

def move_to_wall():
    """
    Move forward until Karel is blocked by a wall.
    """
    while front_is_clear():
        move()

# The start of our program that instructs Karel to pick up all the
# towers of beepers and place them in a pile in the corner.
begin_karel_program()
collect_all_beepers()
drop_all_beepers()
return_home()
end_karel_program()
```

CHAPTER 5

Algorithms

The technique of stepwise refinement that we learned in the last chapter is a powerful tool that helps us craft very concise, descriptive, and correct programs to solve complex computational problems. It is easiest to apply when we encounter problems for which there is a clear division into logical steps, and that we can decompose into subproblems in a direct manner. Sometimes, we encounter problems that require a considerable amount of creativity to solve, not because they are excessively complex, but because it is not obvious how we might be able to apply our computational capabilities to solve the problem.

The design and analysis of solution strategies to computationally-solvable problems is a fundamental pillar of the discipline of computer science. We use the term *algorithm*[1] to refer to a solution strategy for a particular problem. Formally, in the context of computer science, an algorithm is a finite, deterministic, and effective problem-solving strategy that is suitable for implementation as a computer program.[2] The process of designing these strategies is called *algorithmic design*.

You will surely learn dozens of algorithms to solve common computational problems as you continue your study of computer science. You will also spend time studying techniques for the design and analysis of algorithms. But we're getting ahead of ourselves here. The purpose of this final chapter is to introduce you to few powerful algorithms that can be used to solve some challenging problems in Karel's world.

[1] The word *algorithm* comes from the name of the 9th century Persian mathematician Muḥammad ibn Mūsā al-Khwārizmī. He also wrote a book whose title gave us the word *algebra*.

[2] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011

Solving a Maze

Do you think it would be possible to write a program that instructs Karel to escape from a maze? That would seem like a tall order since Karel has awareness only of his immediate surroundings, and we've already seen that he has no memory to speak of.

In Karel's world, a maze may look like Figure 11 on the right. Without a birds-eye view though, it does look pretty hopeless. How-

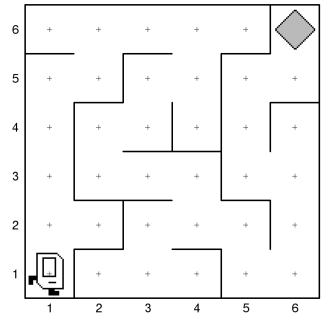

Figure 11: Karel needs to find his way out of a maze. The exit is marked by a single beeper.

ever, if you've ever been in a corn maze yourself, you might know that there exist strategies you can use. (Yes, bumbling around randomly is one of them, and that does work, but you probably won't make it out in time for dinner.)

For most mazes, including the one shown here, you can find your way to the exit by following the *right-hand rule*. The strategy is named this way because it instructs you to put your right hand against the wall of the maze, then keep walking without ever taking your hand off the wall until you find your way out.[3] Another way to express this strategy is to proceed through the maze one step at a time, always taking the rightmost available path.

Knowing this maze-solving algorithm, we are now able to write a program that instructs Karel to find his way out of the maze. The listing below is an *implementation* of this algorithm in the Karel programming language. You should take some time to work through the logic of the program to convince yourself that it indeed accomplishes the goal. Note how compact the program is, despite the apparent complexity of Karel's task. Coming up with the best algorithm to solve a problem often leads to extremely simple code.

[3] Formally proving that such a strategy actually works is a topic of *graph theory*, often studied under both mathematics and analysis of algorithms. Can you think of mazes for which this strategy fails? Can you describe precisely the conditions required of the maze for the algorithm to be correct?

maze-running-karel.py

```
"""
This program instructs Karel to find his way out of a maze, whose
exit is marked by a beeper, using the right-hand rule.
"""
from karel import *

def turn_right():
    turn_left()
    turn_left()
    turn_left()

begin_karel_program()
while not beepers_present():
    turn_right()
    while not front_is_clear():
        turn_left()
    move()
end_karel_program()
```

Multiplying by Two

Trying to get Karel to do a little bit of math is another problem that requires some interesting algorithmic design. Suppose we wanted to write a program that instructs Karel to double the number of beepers in a pile in front of him, as shown in Figure 12 on the right.

Applying top-down design, we might immediately write

```
move()
```

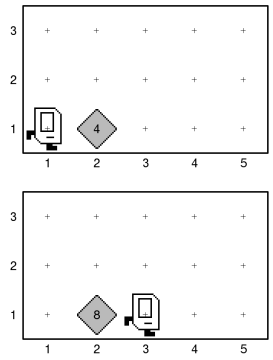

Figure 12: Karel's job is to double the number of beepers in the pile in front of him, as shown in the world on top. When he's done, the world should look like that immediately above.

```
double_beepers()
move()
```

But then how do we write the `double_beepers()` instruction? It is harder than it first appears, and here's where we have to apply some creative algorithmic design. Our initial thought might be to try to count the beepers by repeatedly picking one up, but if Karel picks them all up right away, he won't be able to remember how many he took! Like the solutions we've made for other Karel problems, we'll need to process the beepers one at a time.

One good solution to this problem involves the use of some extra space, a temporary "storehouse", to help us count out the correct number of beepers. We'll choose the corner of 1st street and 3rd avenue next door as the storehouse. Now, if every time Karel picks up a beeper from the original pile, he puts two beepers into the storehouse, we will end up with the desired doubling of the beepers. Thus, we might write a new instruction as follows:[4]

[4] Assuming that we've already defined the usual `turn_around()` instruction.

```
def double_into_storehouse():
    while beepers_present():
        pick_beeper()
        move()
        put_beeper()
        put_beeper()
        turn_around()
        move()
        turn_around()
```

Here is another instance where it's important to understand and document the pre-conditions and post-conditions for this new instruction. The pre-conditions are that Karel is standing at a corner with a pile of beepers, and that the corner in front of Karel is not blocked and empty of beepers. The post-conditions are that Karel is in the same location and facing the same direction as he started, the corner he is standing on has no more beepers, and the corner in front of him has twice the number of beepers as were in the original pile.

Although our new instruction completes the tough algorithmic work of multiplying by two, it does not satisfy the constraints of the problem by itself. We wanted the pile of beepers at the original location to have twice as many as it started with. We'll need to program Karel to move all the beepers from the temporary storehouse back to their original location to achieve this. Luckily it's pretty easy to define a new instruction, which looks remarkably similar to our previous instruction, to do this:

```
def transfer_beepers_back():
    while beepers_present():
        pick_beeper()
```

```
        move()
        put_beeper()
        turn_around()
        move()
        turn_around()
```

Then we can write your `double_beepers()` instruction as follows:

```
def double_beepers():
    double_into_storehouse()
    move()
    turn_around()
    transfer_beepers_back()
    move()
    turn_around()
```

Don't forget to document your pre-conditions and post-conditions!

THE STRATEGY JUST DESCRIBED is not the only one you can use to solve this problem. In many cases, there are algorithms that solve the same problem in a much more concise and efficient manner than the obvious strategies, but they are often quite difficult to come up with. Some require more sophisticated programming techniques that you will learn later on. For example, using *recursion*, where we define a new instruction for Karel that invokes itself, allows us to create a version of `double_beepers()` that gets the job done without needing a storehouse:

```
def double_beepers():
    if beepers_present():
        pick_beeper()
        double_beepers()
        put_beeper()
        put_beeper()
```

While it might be fun to try to figure out what this form of the instruction is doing, you shouldn't worry if you find it hard to understand. We simply wanted you to see that there are often many different algorithms for solving the same problem, some better than others. As you continue to study computer science, you will gain the knowledge and master the skills needed to take advantage of the best algorithms and develop such programs on your own. For now, just enjoy your time programming Karel!