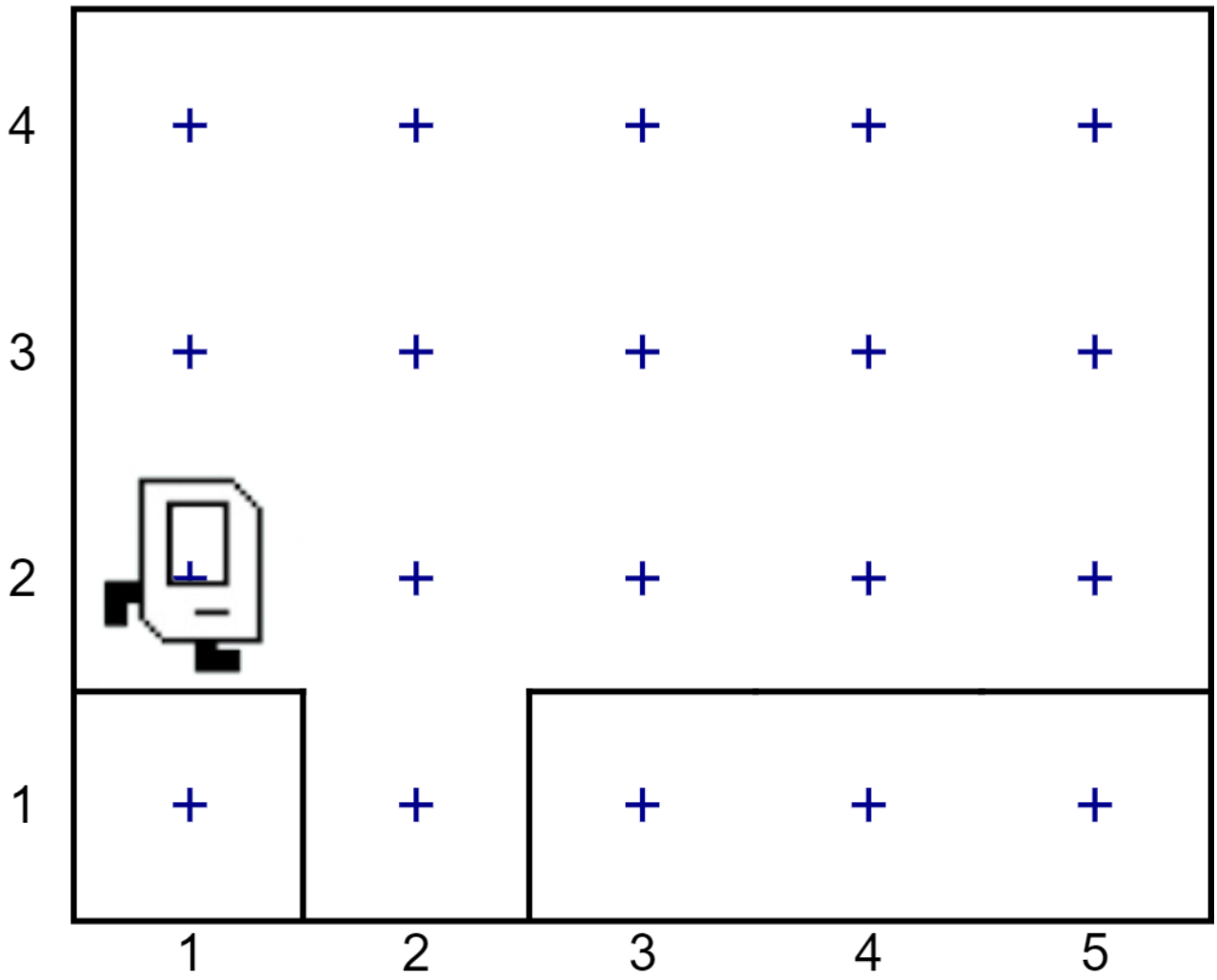
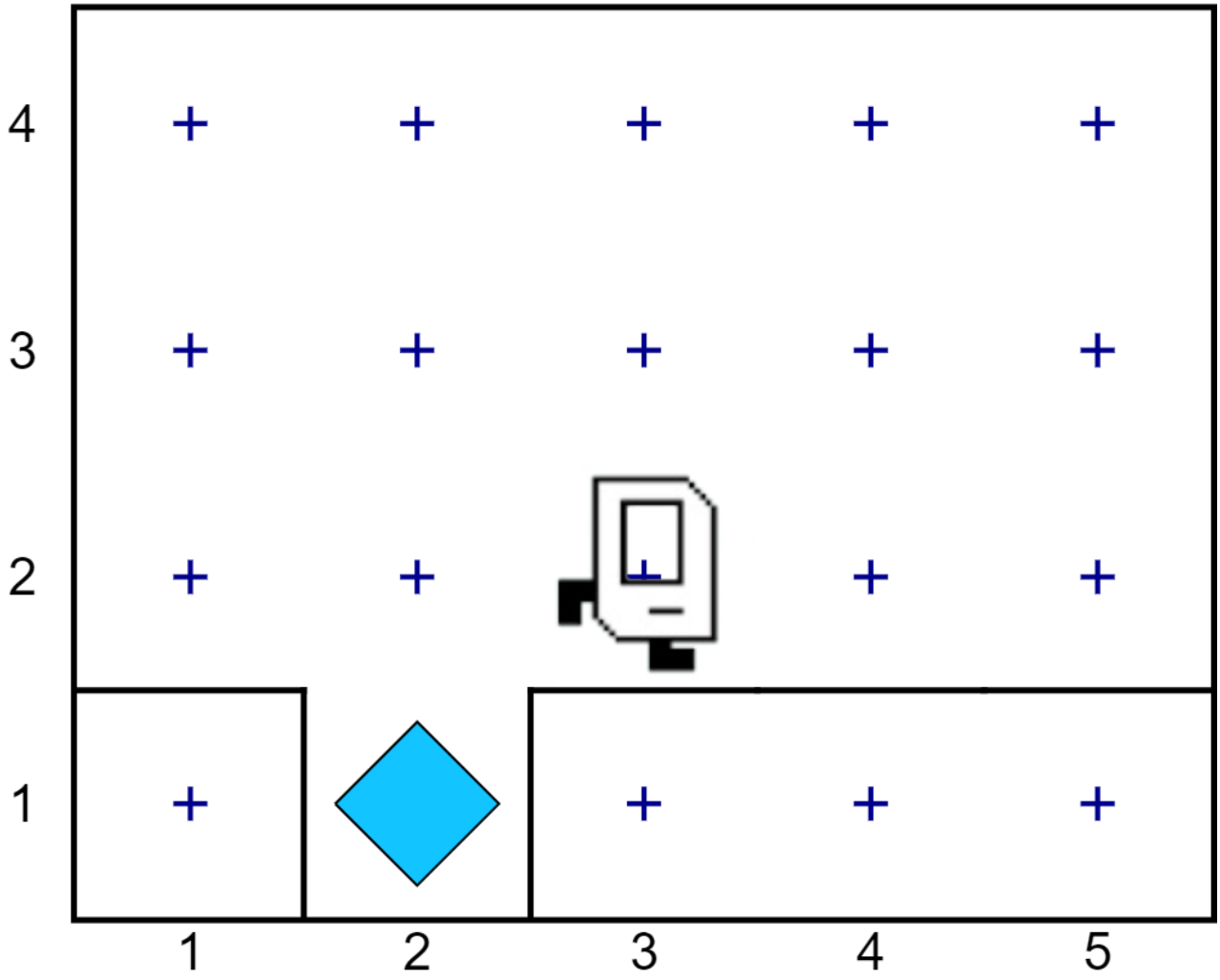
Chapter 4: Decomposition

As a way of illustrating more of the power that comes with being able to define new functions, it's useful to have Karel do something a little more practical than move a beeper from one place to another. The roadways often seem to be in need of repair, and it might be fun to see if Karel can fill potholes in its abstract world. For example, imagine that Karel is standing on the "road" shown in the left-hand figure, one corner to the left of a pothole in the road. Karel's job is to fill the hole with a beeper and proceed to the next corner. The diagram on the right illustrates how the world should look after the program execution.

Before:



After:



If you are limited to the four predefined commands, the `main()` function to solve this problem would look like this:

```
def main():
move()
turn_left()
turn_left()
turn_left()
move()
put_beeper()
turn_left()
turn_left()
move()
turn_left()
turn_left()
turn_left()
move()
```

The initial motivation for defining the `turn_right()` function was that it was cumbersome to keep repeating three `turn_left()` commands to accomplish a right turn. Defining new functions has another important purpose beyond allowing you to avoid repeating the same command sequences every time you want to perform a particular task. The power to define functions unlocks the most important strategy in

programming—the process of breaking a large problem down into smaller pieces that are easier to solve. The process of breaking a program down into smaller pieces is called decomposition, and the component parts of a large problem are called subproblems.

As an example, the problem of filling the hole in the roadway can be decomposed into the following subproblems:

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

If you think about the problem in this way, you can use function definitions to create a program that reflects your conception of the program structure. The main function would look like this:

```
def main():
move()
fill_pothole()
move()
```

The correspondence with the outline is immediately clear, and everything would be great if only you could get Karel to understand what you mean by `fill_pothole()`. Given the power to define functions, implementing `fill_pothole()` is extremely simple. All you have to do is define a `fill_pothole()` function whose body consists of the commands you have already written to do the job, like this:

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

Here is the complete program. Notice how you can understand the programmer's intent simply from reading the main function. When you run the program, the line highlighting shows how a computer will execute it, step by step. However, because the program is nicely broken down we can understand it on a human thought level:

from karel.stanfordkarel import *

def main():

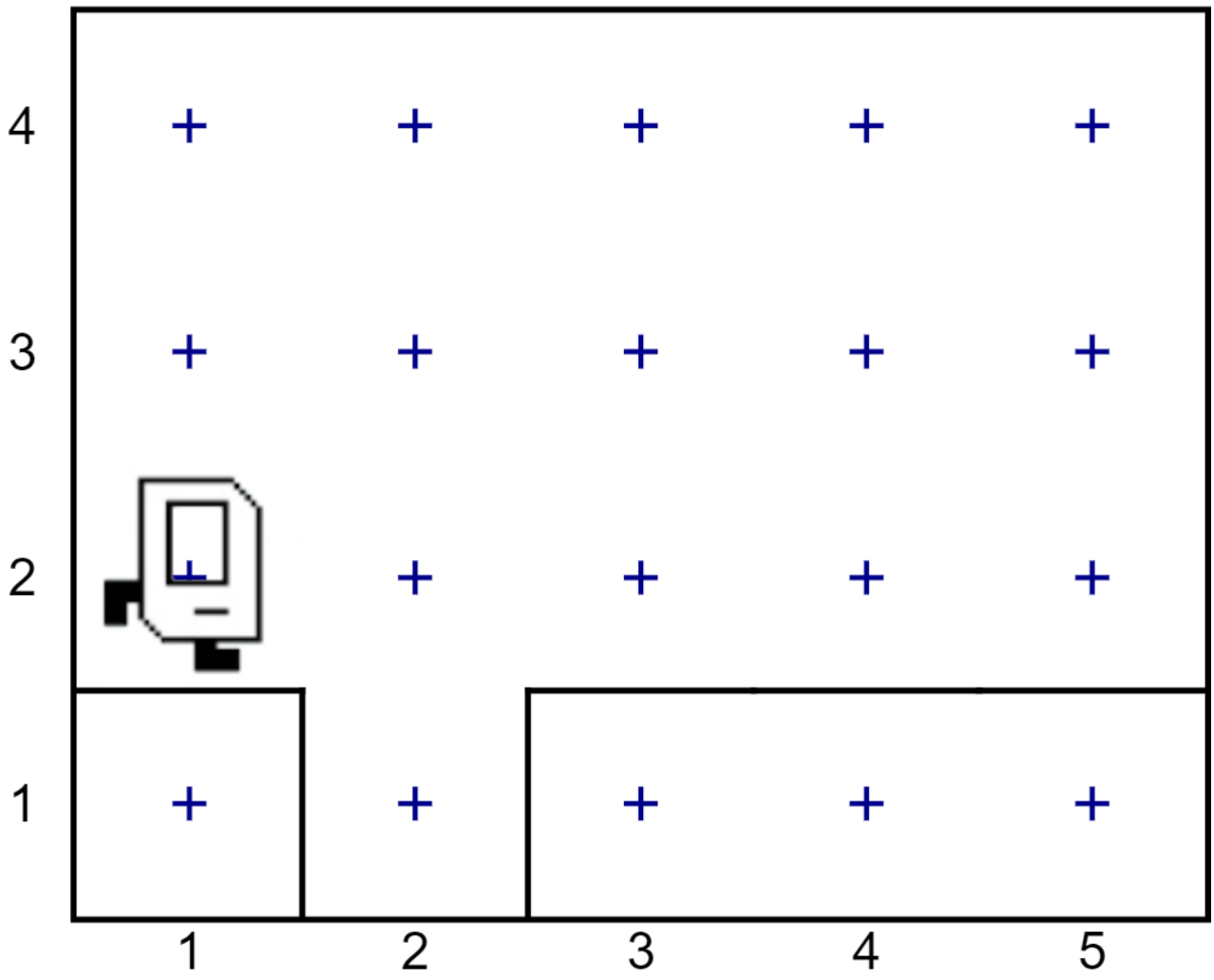
move()

fill_pothole()

move()

Fills the pothole beneath Karel's current position by

```
# placing a beeper on that corner. For this function to work
# correctly, Karel must be facing east immediately above the
# pothole. When execution is complete, Karel will have
# returned to the same square and will again be facing east.
def fill_pothole():
turn_right()
move()
put_beeper()
turn_around()
move()
turn_right()
# Turns Karel 90 degrees to the right.
def turn_right():
turn_left()
turn_left()
turn_left()
# Turns Karel around 180 degrees.
def turn_around():
turn_left()
turn_left()
```



Next Chapter