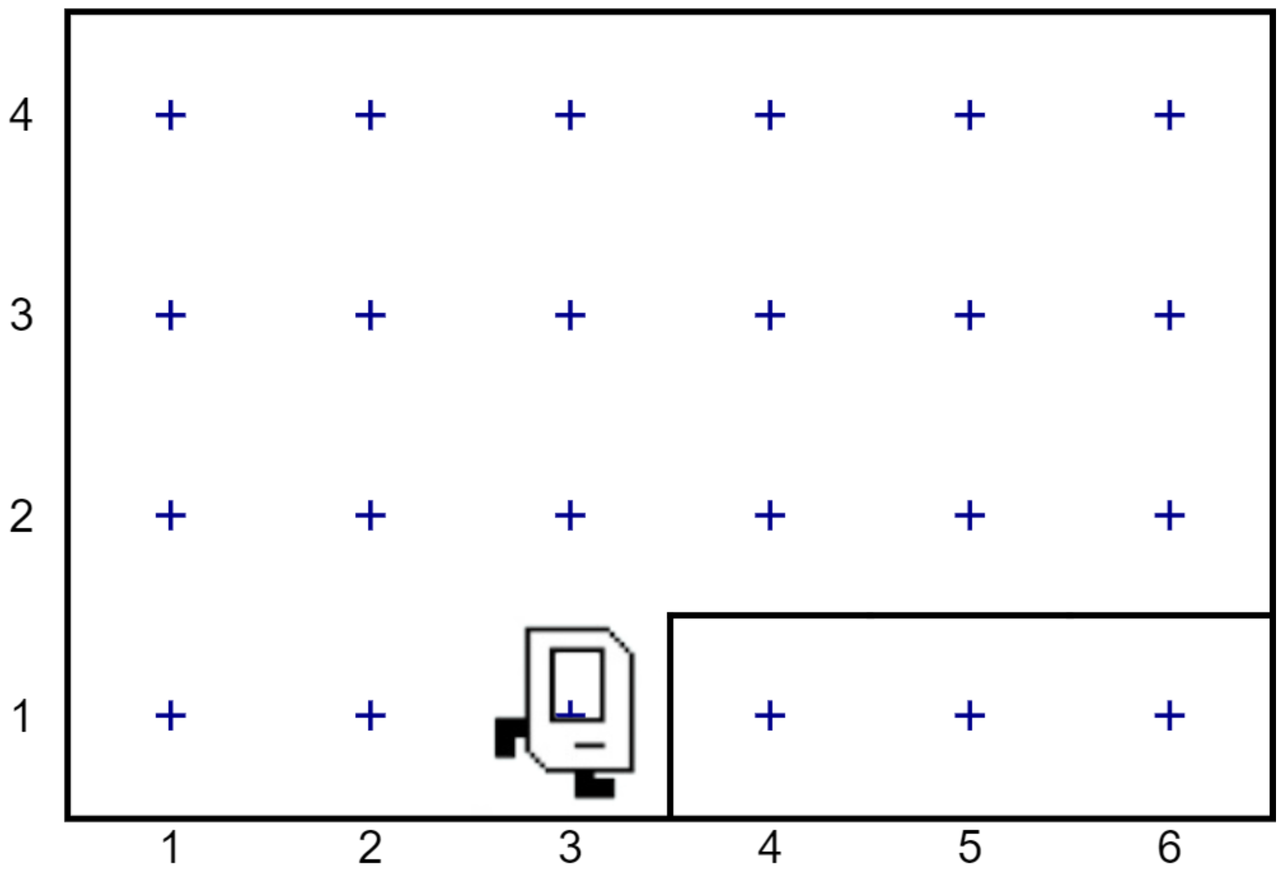
Chapter 2: Programming Karel

compedu.stanford.edu/karel-reader/docs/python/en/chapter2.html

The simplest style of Karel program uses text to specify a sequence of built-in commands that should be executed when the program is **run**. Consider the simple Karel program below. The text on the left is the program. The state of Karel's world is shown on the right:

```
# File: FirstKarel.py

# -----------------------------

# The FirstKarel program defines a "main"

# function with three commands. These commands cause

# Karel to move forward one block, pick up a beeper

# and then move ahead to the next corner.

from karel.stanfordkarel import *

def main():

move()

pick_beeper()

move()
```



Press the "Run" button to execute the program. Programs are typically written in a special application called an **Integrated Development Enviroment** (IDE) and most Karel programs are written in an IDE called PyCharm. Like an IDE, this reader has the ability to execute programs in order to help you *see* how things work as you learn.

The program is composed of several parts. The first part consists of the following lines:

```
# File: FirstKarel.py
# -----------------------------
# The FirstKarel program defines a "main"
# function with three commands. These commands cause
# Karel to move forward one block, pick up a beeper
# and then move ahead to the next corner.
```

These lines are an example of a **comment**, which is simply text designed to explain the operation of the program to human readers. Comments in both Karel and Python begin with the characters `#` and include the rest of the line. In a simple program, extensive comments may seem silly because the effect of the program is obvious, but they are extremely important as a means of documenting the design of larger, more complex programs. The second part of the program is the line:

```
from karel.stanford import *
```

This line requests the inclusion of all definitions from the `karel.stanford` library. This library contains the basic definitions necessary for writing Karel programs, such as the definitions of the standard operations `move()` and `pick_beeper()`. Because you always need access to these operations, every Karel program you write will include this import command before you write the actual program.

The final part of the Karel program consists of the following function definition:

```
    def main():
        move()
        pick_beeper()
        move()
```

These lines represent the definition of a new **function**, which specifies the sequence of steps necessary to respond to a command. As in the case of the FirstKarel program itself, the function definition consists of two parts that can be considered separately: The first line constitutes the function header and the indented code following is the function body. If you ignore the body for now, the function definition looks like this:

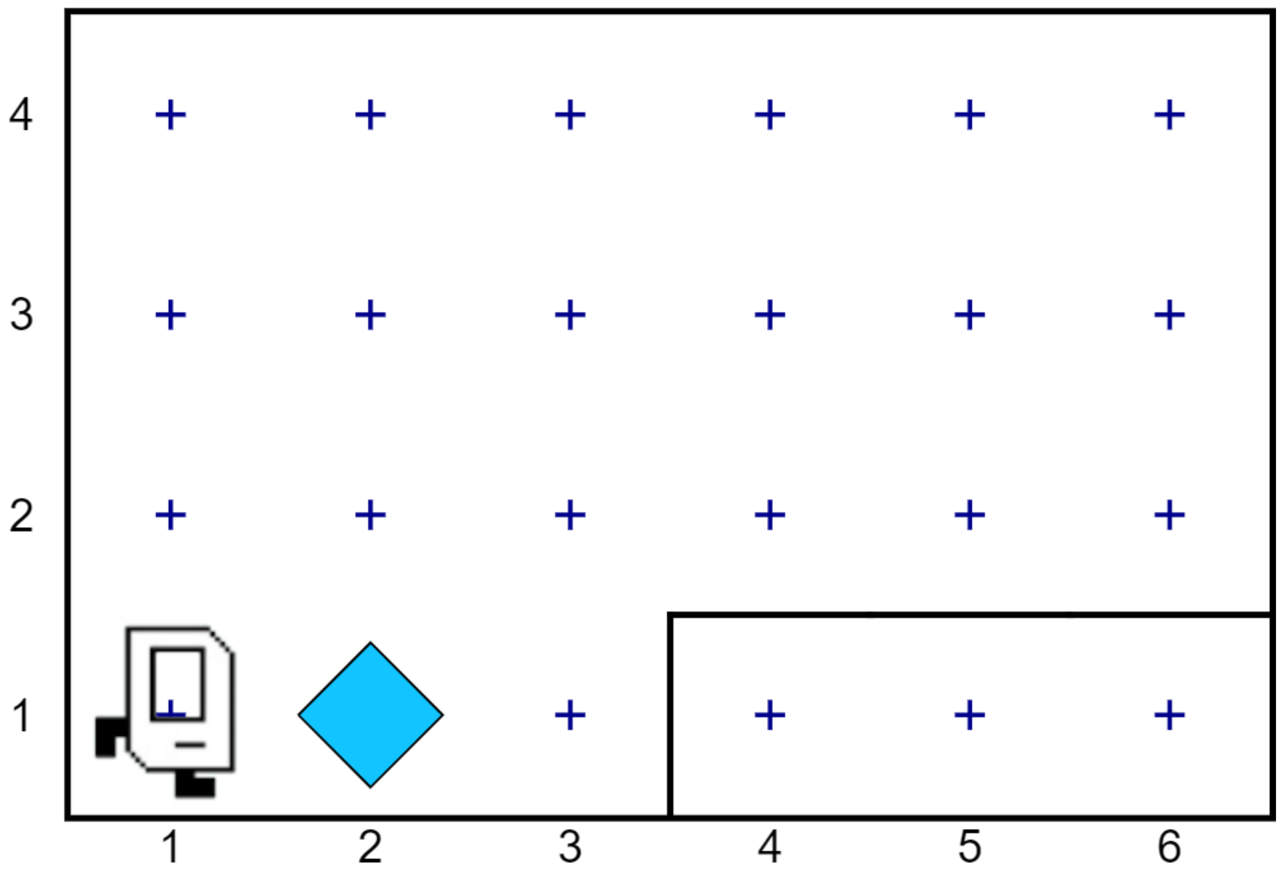
```
    def main():
```
body of the function definition

The first word in the function header, `def`, is part of Python's syntactic structure. It says that you are creating a new function. The next word on the header line specifies the name of the new function, which in this case is **main**. Defining a function means that Karel can now respond to a new command with that name. The `main()` command plays a special role in a Karel program. When you start a Karel program it creates a new Karel instance, adds that Karel to a world that you specify, and then issues the `main()` command. The effect of running the program is defined by the body of the `main()` function, which is a sequence of commands that the robot will execute in order. For example, the body of the `main()` function for the `FirstKarel` program is:

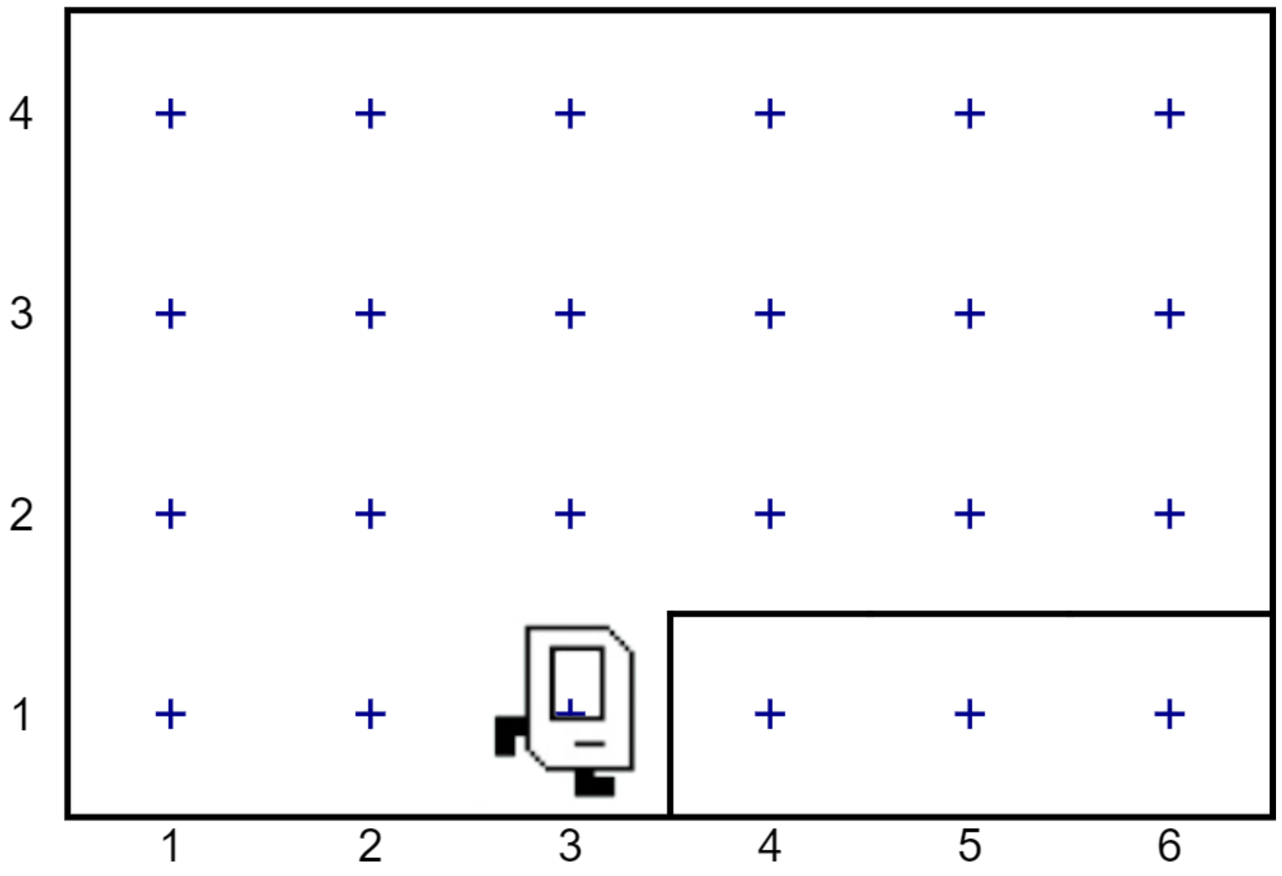
```
    move()
    pick_beeper()
    move()
```

Thus, if the initial state of the world matches the example given in Chapter 1, Karel first moves forward into the corner containing the beeper, picks up that beeper, and finally moves forward to the corner just before the wall, as shown in the following before-and-after diagram:

Before:



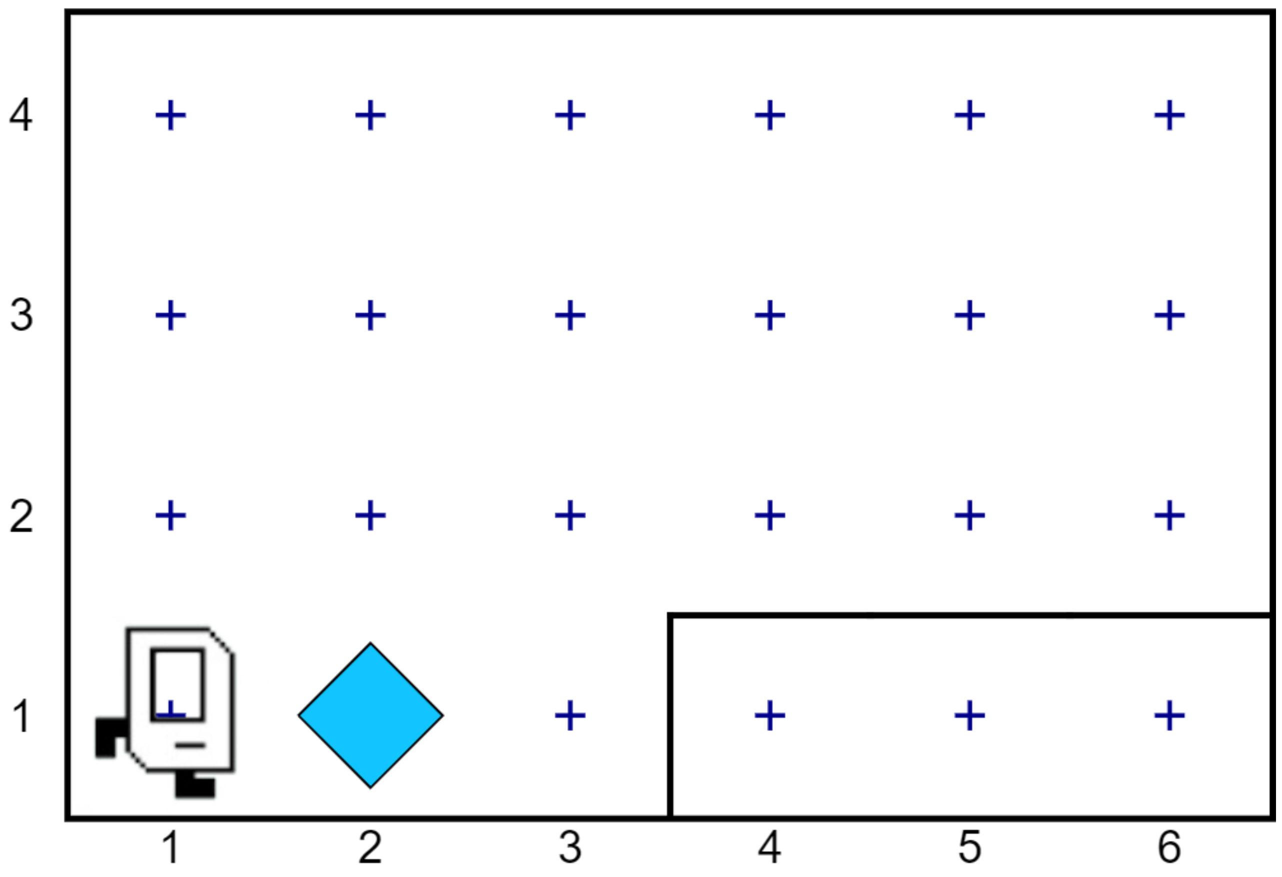
After:



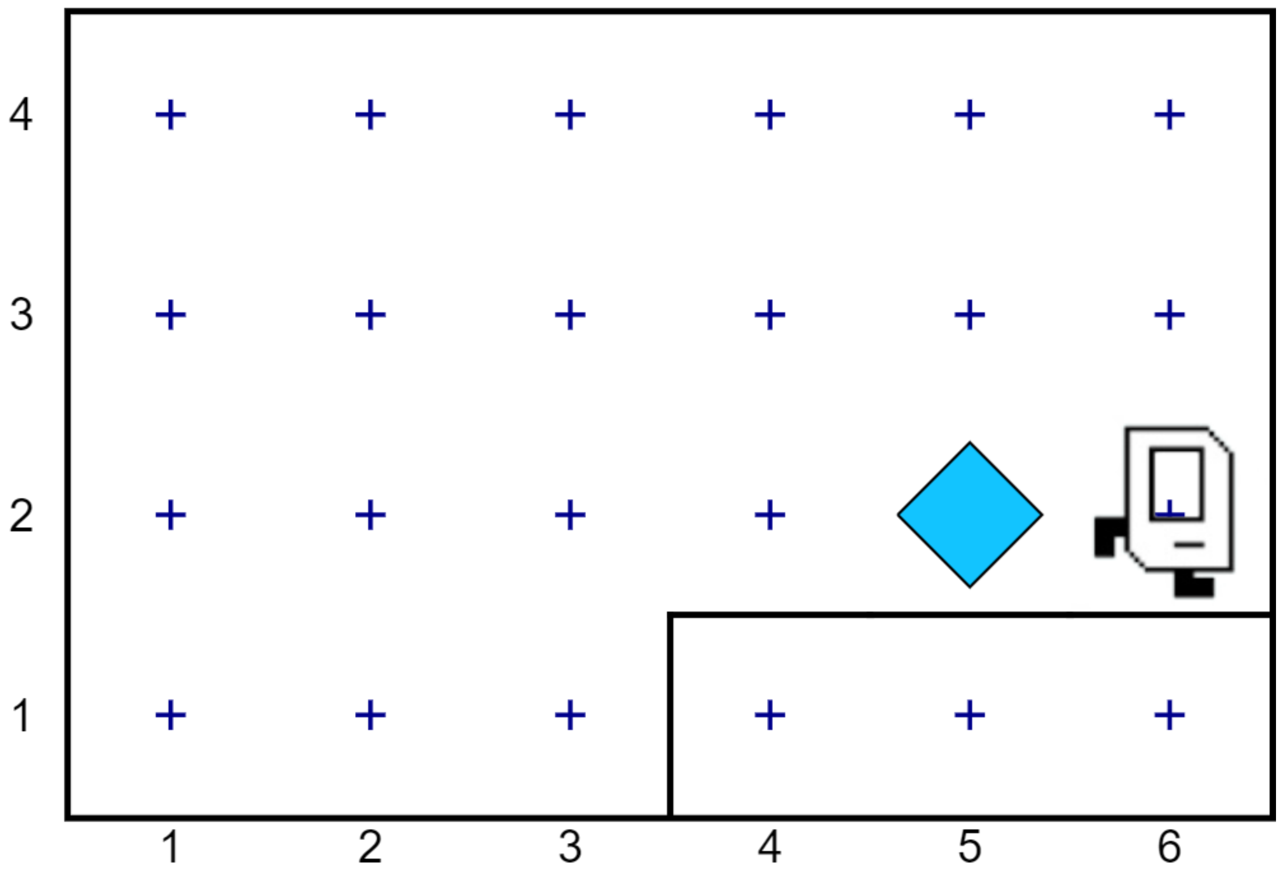
Solving a more interesting problem

The FirstKarel program defined above doesn't do very much as yet. Let's try to make it a little more interesting. Suppose that the goal is not simply to get Karel to pick up the beeper but to move the beeper from its initial position on 2nd column and 1st row to the center of a ledge. Thus, your next assignment is to define a new Karel program that accomplishes the task illustrated in this diagram:

Before:



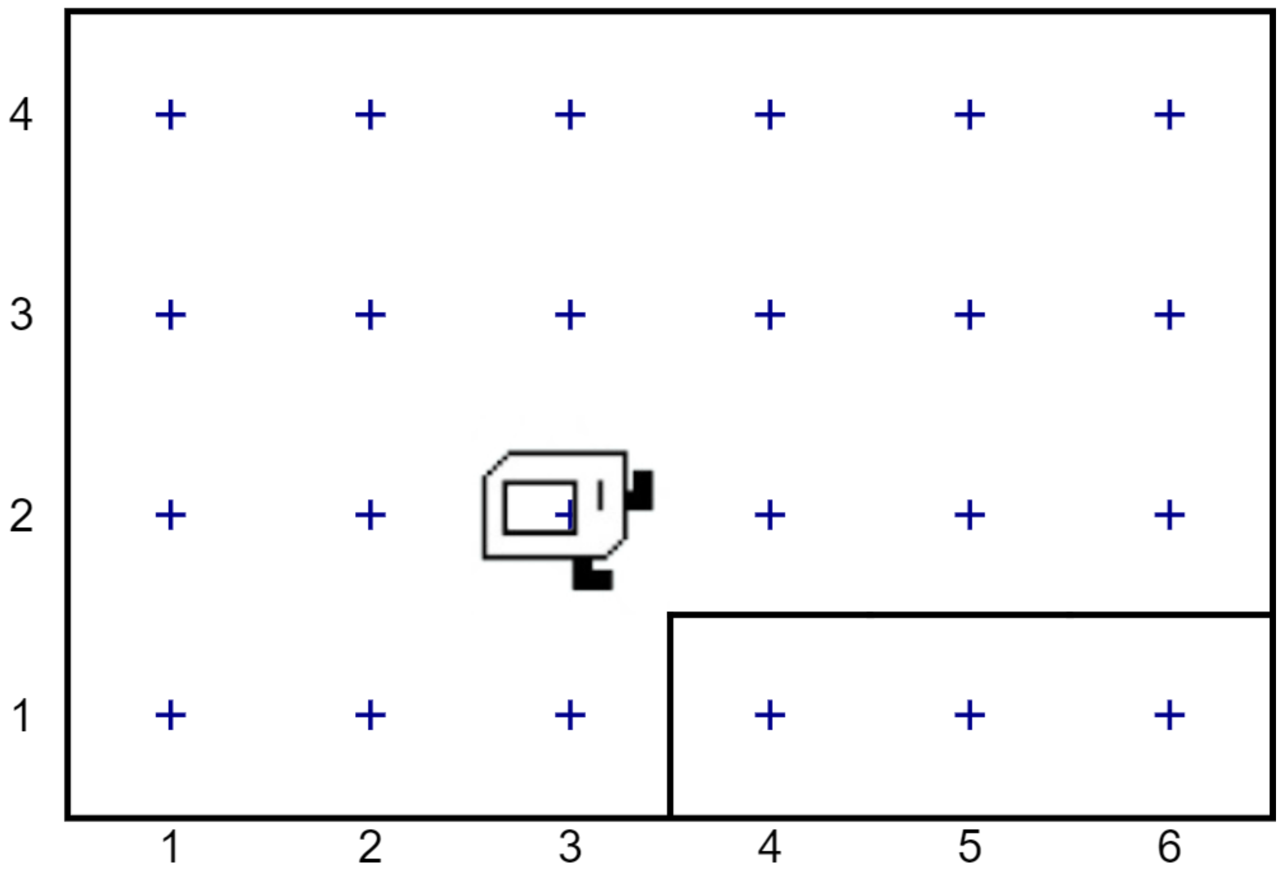
After:



The first three commands in the new program—the ones that move forward, pick up the beeper, and then move up to the ledge—are the same as before:

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

From here, the next step is to turn left to begin climbing the ledge. That operation is easy, because Karel has a `turn_left()` command in its standard repertoire. Executing a `turn_left()` command at the end of the preceding sequence of commands leaves Karel facing north on the corner of 1st row and 3rd column. If Karel then executes a `move()` command, it will move north to reach the following position:



From here, 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 just as easy as getting Karel to turn left, there is a slight problem: Karel's language includes a `turn_left()` command, but no `turn_right()` command. It's as if you bought the economy model and have now discovered that it is missing some important features.

At this point, you have your first opportunity to begin thinking like a programmer. You have one set of commands, but not exactly the set you need. What can you do? Can you accomplish the effect of a `turn_right()` command using only the capabilities you have? The answer, of course, is yes. You can accomplish the effect of turning right by turning left three times. After three left turns, Karel will be facing in the desired direction. From here, all you need to do is program Karel to move over to the center of the ledge, drop the beeper and then move forward to the final position. Here is a complete implementation of a program that accomplishes the entire task:

# File: FirstKarel.py

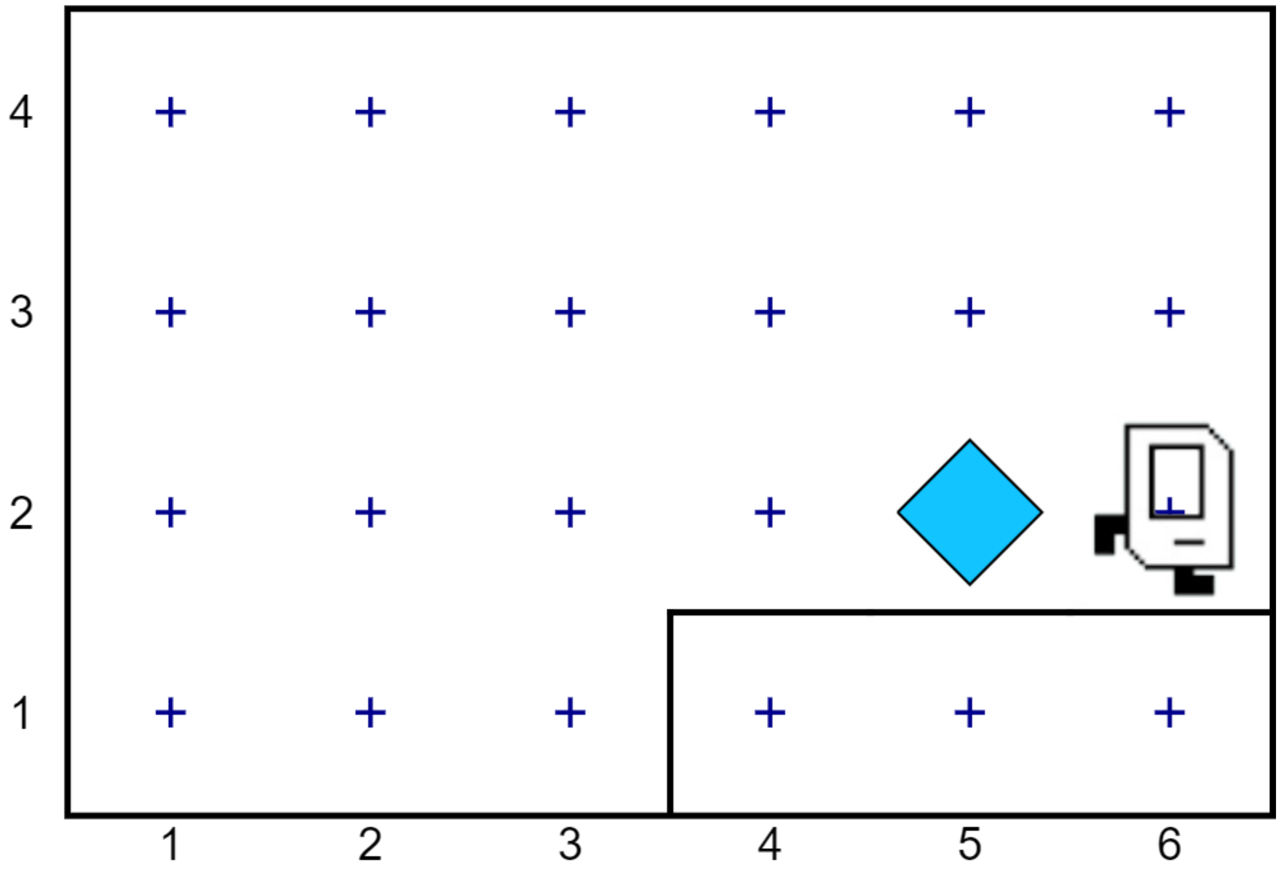
# The FirstKarel program defines a "main"

# function with commands that cause Karel to pick up

# a beeper and place it on a ledge.

from karel.stanfordkarel import *

```
def main():

move()

pick_beeper()

move()

turn_left()

move()

turn_left()

turn_left()

turn_left()

move()

move()

put_beeper()

move()
```



Next Chapter