Chris Piech and Mehran Sahami
Code in Place

Handout
April 13, 2020

Assignment #1: Karel the Robot

Due: 11:59PM (Anywhere on Earth) on Wed, April 22nd

Based on problems by Nick Parlante and Eric Roberts, lovingly modified by your current Code in Place staff.

This assignment consists of four Karel programs (and one extra credit optional program). There is a starter project including all of these problems on the CS106A website under the "Assignments" tab. Before you start on this assignment, make sure to read the handout Using Karel with PyCharm in its entirety. When you are ready to start working on these programs, you need to:

1. Download the starter project as described in Handout (Using Karel with PyCharm).
2. Edit the Karel program files so that the assignment actually does what it's supposed to do. This will involve a cycle of coding, testing, and debugging until everything works.
3. Once you have gotten each part of the program to run correctly in the default world associated with the problem, you should make sure that your code runs properly in all of the worlds that we have provided for a given problem. Instructions on how to load new worlds for Karel to run in can be found in Handout (Using Karel with PyCharm).
4. We will explain how you can submit your work by Friday midnight anywhere on earth. If you finish early, consider making a creative karel project!

The three Karel problems to solve are described on the following pages. We hope that most folks can get through the first two problems. The third problem is a beautiful challenge which we encourage folks to try.

Please remember that your Karel programs must limit themselves to the language features described in the *Karel the Robot Learns Python* reader. You may not use other features of Python (including variables, parameters, `break`, and `return`), even though the PyCharm-based version of Karel accepts them.

In assignment 0, we programmed Karel in the browser. Can't we just do that instead of installing PyCharm? Answer: Yes we do have an online programming environment for Karel. We want you to try to install PyCharm though, but if you have trouble the browser IDE will be made available for these problems!

Problem 1 (`TripleKarel.py`)

Your first task is to help Karel paint the exterior of some oddly-shaped buildings using beepers! For this problem, Karel starts facing west next to a "building" (represented by a rectangle, constructed from walls) whose sides span one or more corners. Karel's goal is to paint all of the buildings present in the world by placing beepers along three of the sides of each of the buildings.

We recommend breaking down the problem into the following steps:

1. First, Karel should paint one side of the rectangle, placing beepers on all corners that are adjacent to the wall of the building. Note that there's a boundary detail here: **the last square where Karel ends should not have a beeper on it**.
2. Next, Karel should accomplish the task of painting a single rectangle. Think about how you can use the functionality of the previous subtask to help you accomplish this goal. You may need to write a small amount of code to reposition Karel in between painting individual walls of a building.
3. Finally, the overall `TripleKarel` problem is just painting all three buildings in the world. Again, you may need to write a small amount of code to reposition Karel in between painting individual buildings.

Figures demonstrating the before and after stages of each of the three steps are shown on the following page.

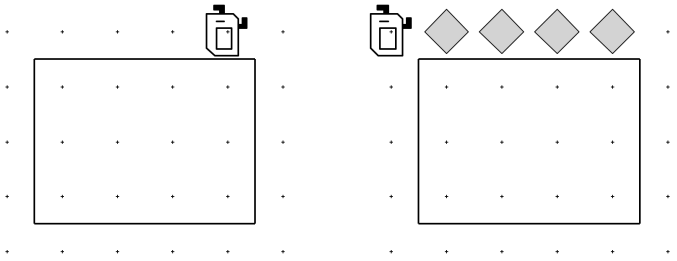

Figure 2: After you've completed the first step, Karel should be able to paint one side of one building. Running the program would result in the above start (left) and end (right) states.

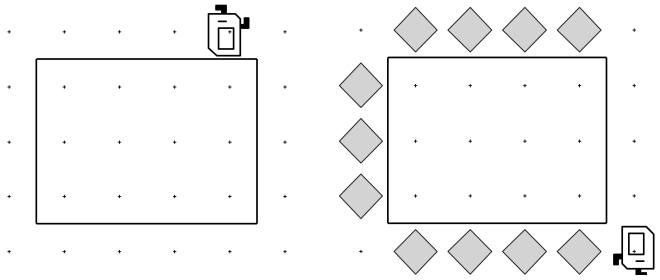

Figure 3: After you've completed the second step, Karel should be able to paint one building. Running the program would result in the above start (left) and end (right) states.

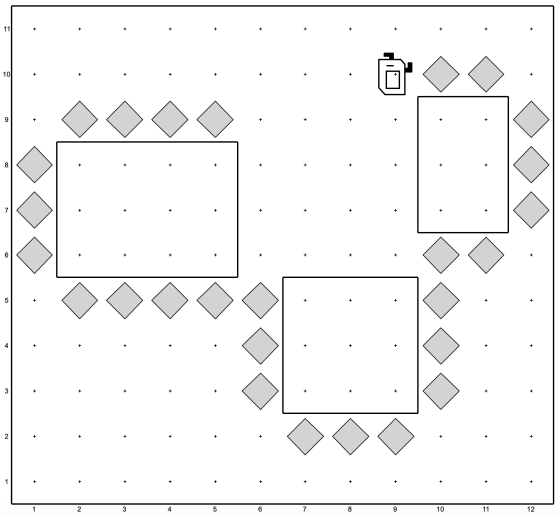

Figure 4: After you have painted the whole world, the end result should look like this.

You can assume that:

- Karel will always start facing west at the upper right corner of the leftmost building (at the position where the first beeper should be placed).
- Karel will have infinite beepers in his beeper bag, so he can paint any size of buildings.
- Although buildings may be of varying sizes, there will always be exactly three of them, and their relative position to one another will always be the same (as displayed in Figure 6). If you are still confused about what assumptions you can make about the world, see the additional **`Triple`** world files we have included.

You should make sure your program runs successfully in all of the following worlds (which are just a few different examples to test out the generality of your solution):
`TripleKarel.w` (default world), **`Triple1.w`**, **`Triple2.w`**, **`Triple3.w`**

Note that all Karel worlds are located in the **`worlds`** folder in the Assignment 1 project folder.

Problem 2 (`StoneMasonKarel.py`)

Your second task is to repair the damage done to the Main Quad in the 1989 Loma Prieta earthquake. In particular, Karel should repair a set of arches where some of the stones (represented by beepers, of course) are missing from the columns supporting the arches, as illustrated in Figure 5 (below).

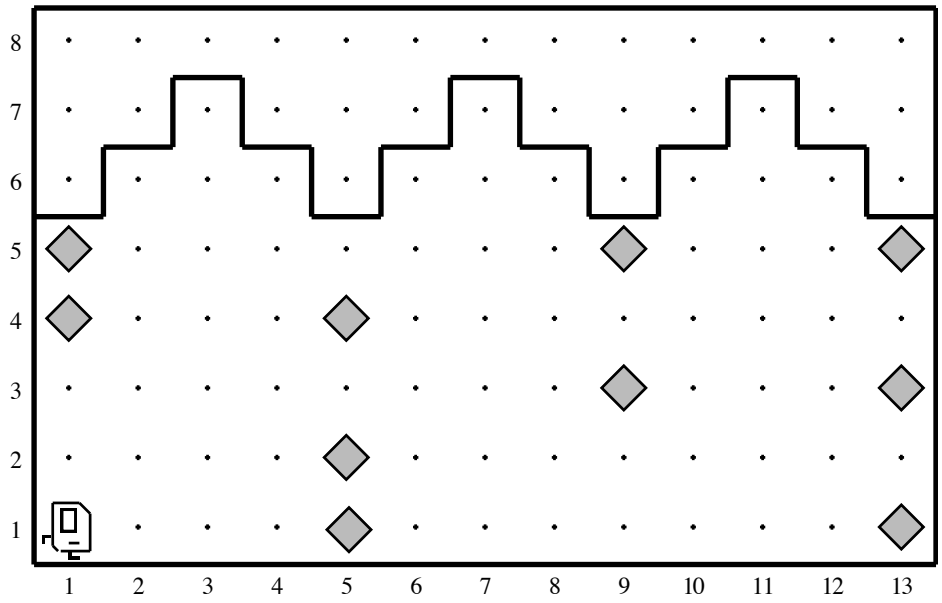

Figure 5: An initial example world with broken arches that `StoneMasonKarel` must repair

Your program should work on the world shown above, but it should be general enough to handle any world that meets the basic conditions outlined at the end of this problem. There are several example worlds in the starter folder, and your program should work correctly in all of them.

When Karel is done, the missing stones in the columns should be replaced by beepers, so that the final picture resulting from the initial world shown in Figure 5 would look like the illustration in Figure 6.

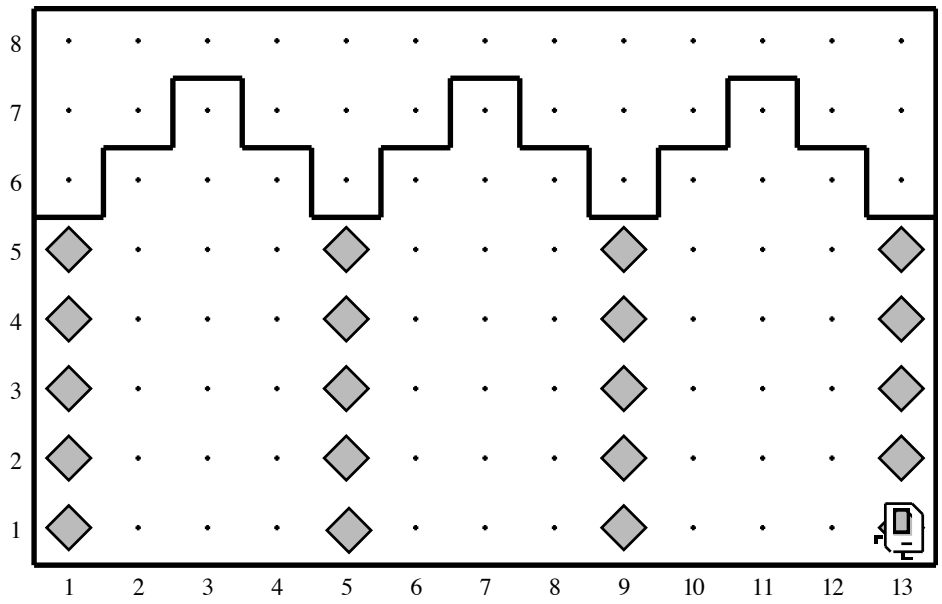

Figure 6: Karel should repair the Main Quad to a structurally sound state after completion.

Karel's final location and the final direction Karel is facing at the end of the run do not matter.

Karel may count on the following facts about the world:

- Karel starts at the 1st row and 1st col, facing east, with an infinite number of beepers in Karel's beeper bag.
- The columns are always exactly four corners apart, so they would be built on 1st col, 5th col, 9th col, and so on.
- The final column will always have a wall immediately after it.
- The top of a beeper column will always be marked by a wall. Karel cannot assume that columns are always five units high, or even that all columns within a given world are the same height.
- In an initial world, some columns may already contain beepers representing stones that are still in place. Your program should not put a second beeper on corners that already have beepers.

You should make sure your program runs successfully in all of the following worlds (which are just a few different examples to test out the generality of your solution):
`StoneMasonKarel.w` (default world), **`SampleQuad1.w`**, **`SampleQuad2.w`**

Note that all Karel worlds are located in the **`worlds`** folder in the Assignment 1 project folder.

Challenge Problem

Challenge: Problem 3 (`MidpointKarel.py`)

If you finished the other two problems, consider this challenge.

.

As an exercise in solving algorithmic problems, program Karel to place a single beeper at the center of 1st row. For example, say Karel starts in the 5x5 world pictured in Figure 9.

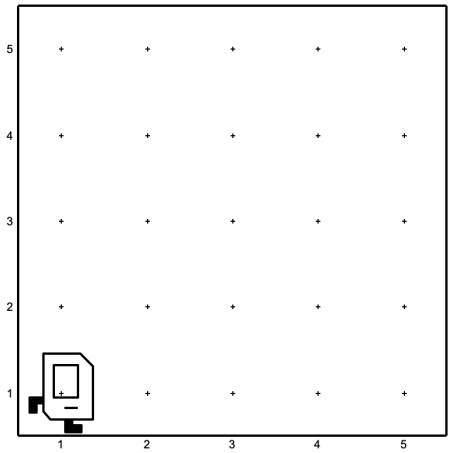

Figure 9: The beginning state for **`MidpointKarel`**

Karel should end with Karel standing on a beeper in the following position (Figure 10):

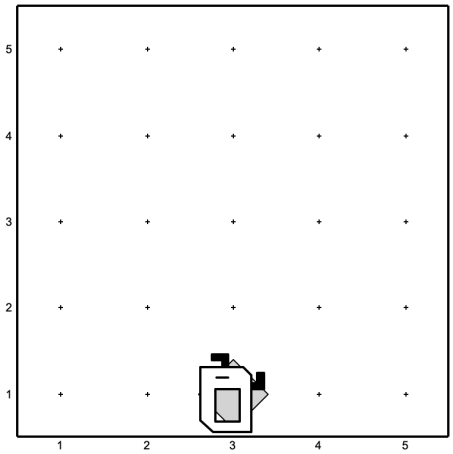

Figure 10: The end state for **`MidpointKarel`**

Note that the final configuration of the world should have only a single beeper at the midpoint of 1st row. Along the way, Karel is allowed to place additional beepers wherever it wants to, but must pick them all up again before it finishes. Similarly, if Karel paints/colors any of the corners in the world, they must all be uncolored before Karel finishes.

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

- Karel starts at 1st row and 1st column, 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.

Your program, moreover, can assume the following simplifications:

- 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 so feel free to be creative!

You should make sure your program runs successfully in all of the following worlds (which are just a few different examples to test out the generality of your solution):
`MidpointKarel.w` (default world)**,** **`Midpoint1.w`,** **`Midpoint2.w`,** **`Midpoint8.w`**

Note that all Karel worlds are located in the **`worlds`** folder in the Assignment 1 project folder.

Make Something Creative

Bonus: Create your own Karel project

If you finish early, you may optionally write a Karel project of your own choice. Modify **`ExtensionKarel.py`** to use Karel to complete any task of your choosing. Extensions are a great chance for practice and, if your extension is substantial enough, it might help you earn a + score. Make sure to write comments to explain what your program is doing and update **`ExtensionKarel.w`** to be an appropriate world for your program.

Advice, Tips, and Tricks

All of the Karel problems you will solve, except for **`CollectNewspaperKarel`**, should be able to work in a variety of different worlds that match the problem specifications. When you first run your Karel programs, you will be presented with a sample world in which you can get started writing and testing your solution. However, we will test your solutions to each of the Karel programs, except for **`CollectNewspaperKarel`**, in a variety of test worlds. Unfortunately, each quarter, many students submit Karel programs that work brilliantly in the default worlds but which fail catastrophically in some of the other test worlds. Before you submit your Karel programs, ***be sure to test them out in as many different worlds as you can.*** We've provided several test worlds in which you can experiment, but you can also develop your own worlds for testing.

When writing your Karel programs, to the maximum extent possible, try to use the top-down design techniques we developed in class. Break the task down into smaller pieces until each subtask is something that you know how to do using the basic Karel commands and control statements. These Karel problems are somewhat tricky, but appropriate use of top-down design can greatly simplify them.

As mentioned in class, it is just as important to write clean and readable code as it is to write correct and functional code. A portion of your grade on this assignment (and the assignments that follow)

will be based on how well-styled your code is. Before you submit your assignment, take a minute to review your code to check for stylistic issues like those mentioned below.

Have you added comments to your methods? To make your program easier to read, you can add comments before and inside your methods to make your intention clearer. Good comments give the reader a clue about what a method does and, in some cases, how it works. We recommend writing pre- and post-conditions for each function, as shown in the table below.

Not-So-Good Style	Good Style

`def fill_row_with_beepers():`
`  while front_is_clear():`
`  put_beeper()`
`  move()`
`  put_beeper()`
	`def fill_row_with_beepers():`
	`  """`
	`  Makes Karel move to the end of`
	`  the row, dropping a beeper`
	`  before each step it takes.`
	
	`  Pre-condition: None`
	
	`  Post-condition: Karel is facing`
	`  the same direction as before,`
	`  and every step between Karel's`
	`  old position and new position`
	`  has had a beeper added to it.`
	`  """`
	`  while front_is_clear():`
	`  put_beeper()`
	`  move()`
	`  put_beeper()`

Did you decompose the problem? There are many ways to break these Karel problems down into smaller, more manageable pieces. Decomposing the problem elegantly into smaller sub-problems will result in a small number of easy-to-read functions, each of which performs just one small task. Decomposing the problem in other ways may result in functions that are trickier to understand and test. Look over your code and check to see whether you've decomposed the problem into smaller pieces. Does your code consist of a few enormous functions (not so good), or many smaller functions (good)?

This is not an exhaustive list of stylistic conventions, but it should help you get started. As always, if you have any questions on what constitutes good style, feel free to see the course helpers at the LaIR with questions, come visit the teaching team during office hours, or email your section leader with questions!