Eric Roberts
CS 106A

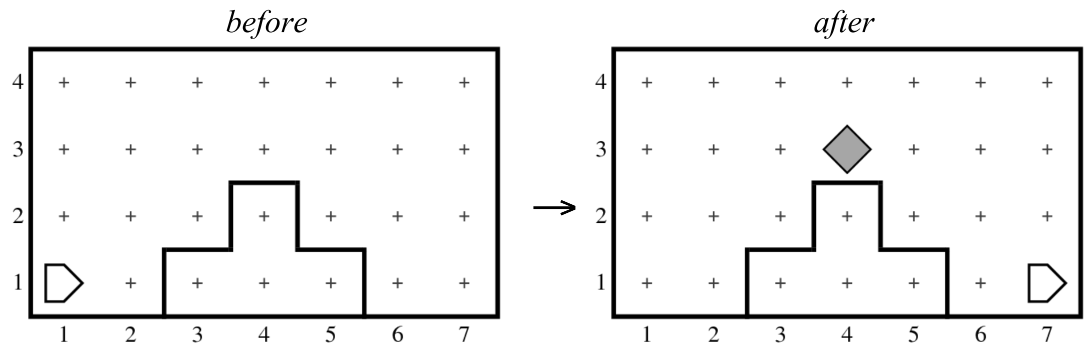
Handout #3
April 5, 2017

Programming in Karel

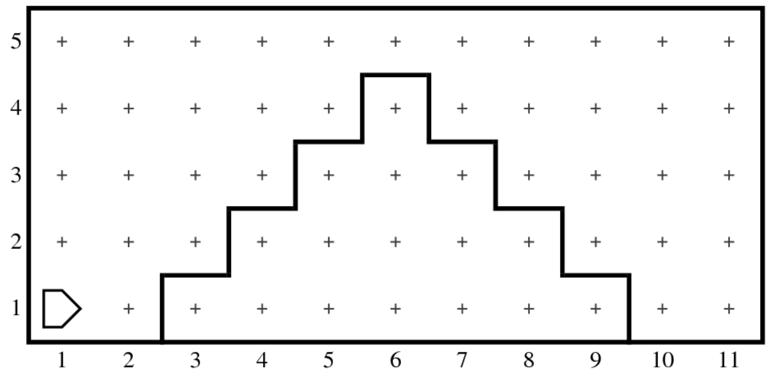
Chapter 1 of the draft textbook includes several examples that illustrate the use of control statements in Karel's world. Whenever possible, I like to solve different problems in lecture so that you can see a different set of examples than the ones in the text. By offering two sets of examples, you are in a better position to understand the general principles as opposed to the details of a particular problem.

In lecture, the goal is to get you to think about the problems and solve them as we go. If I put the solution on the handout, it's far too tempting just to look at the answer instead of trying to work things out on your own. Thus, I use handouts to describe the problems and then put the solutions up on the web so that you can look them over after class.

The problem we will solve today is that of getting Karel to climb mountains. Like everything else in Karel's world, the mountain is abstract and must be constructed from the available materials, specifically beepers and walls. The goal is to get Karel to move up to a mountain marked out by walls, climb the mountain by moving up each stair-step, put down a beeper to serve as a flag, climb back down the other side, and finally move ahead to the end of the world. This problem is illustrated in the following diagram:



At first, the goal is simply to solve the specific problem posed by this mountain. From there, however, the more interesting task is to generalize the problem so that Karel can climb larger mountains with the same stair-step structure, like this:



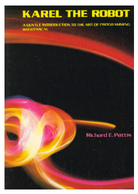
Programming in Karel

Eric Roberts and Jerry Cain
CS 106J
April 5, 2017

Once upon a time . . .

Rich Pattis and Karel the Robot

- Karel the Robot was developed by Rich Pattis in the 1970s when he was a graduate student at Stanford.
- In 1981, Pattis published *Karel the Robot: A Gentle Introduction to the Art of Programming,* which became a best-selling introductory text.
- Pattis chose the name *Karel* in honor of the Czech playwright Karel Čapek, who introduced the word *robot* in his 1921 play *R.U.R.*
- In 2006, Pattis received the annual award for Outstanding Contributions to Computer Science Education given by the ACM professional society.

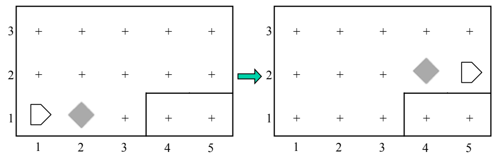


Review: Primitive Karel Commands

- On Monday, you learned that Karel understands the following commands:

`move()`	Move forward one square
`turnLeft()`	Turn 90 degrees to the left
`pickBeeper()`	Pick up a beeper from the current square
`putBeeper()`	Put down a beeper on the current square

- At the end of class, we designed a Karel program to solve the following problem:



The `MoveBeeperToLedge` Program

```
/*
 * File: MoveBeeperToLedge.k
 * -------------------------
 * This program moves a beeper to a ledge.
 */

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

Syntactic Rules and Patterns

- The definition of `MoveBeeperToLedge` on the preceding slide includes various symbols (such as curly braces, parentheses, and semicolons) and special keywords (such as `function`) whose meaning may not be immediately clear. These symbols and keywords are required by the ***syntactic rules*** of the Karel programming language, in much the same way that syntactic rules govern human languages.
- When you are learning a programming language, it is often wise to ignore the details of the language syntax and focus instead on learning a few general patterns. Karel programs, for example, fit a common pattern in that they define one or more functions that describe the steps Karel must perform in order to solve a particular problem.

Defining New Functions

- In Karel—and in JavaScript as you will see beginning next week—a ***function*** is a sequence of statements that has been collected together and given a name. All functions in Karel have the following form:

```
function name() {
   statements that implement the desired operation
}
```

- The first function in a Karel program is the ***main function,*** which is called when you press the **Run** button at the bottom of the screen.
- Most Karel programs define additional ***helper functions*** that implement individual steps in the complete solution.

The `turnRight` Function

- As a simple example, the following function definition allows Karel to turn right by executing three `turnLeft` operations:

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

- Once you have made this definition, you can use `turnRight` in your programs in exactly the same way you use `turnLeft`.
- In a sense, defining a new function is analogous to teaching Karel a new word. The name of the function becomes part of Karel's vocabulary and extends the set of operations the robot can perform.

Helper Functions in a Program

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

/* Turns Karel right 90 degrees */

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

Exercise: Defining functions

- Define a function `turnAround` that turns Karel around 180°.
- The `turnRight` and `turnAround` functions are so important that they are included in a library called `"turns"`.
- Define a function `backup` that moves Karel backward one square, leaving Karel facing in the same direction.

Control Statements

- In addition to allowing you to define new functions, Karel also includes statement forms that allow you to change the order in which statements are executed. Such statements are called ***control statements***.
- The control statements available in Karel are:
  - The `repeat` statement, which repeats a set of statements a predetermined number of times.
  - The `while` statement, which repeats a set of statements as long as some condition holds.
  - The `if` statement, which applies a conditional test to determine whether a set of statements should be executed at all.
  - The `if-else` statement, which uses a conditional test to choose between two possible actions.

The `repeat` Statement

- In Karel, the `repeat` statement has the following form:

```
repeat (count) {
   statements to be repeated
}
```

- As with the other control statements, the `repeat` statement consists of two parts:
  - The ***header line***, which specifies the number of repetitions
  - The ***body***, which is the set of statements to be repeated
- The keyword `repeat` and the various punctuation marks appear in boldface, which means that they are part of the `repeat` statement pattern. The things you can change appear in italics: the number of repetitions and the statements in the body.

Using the `repeat` Statement

- You can use **`repeat`** to redefine **`turnRight`** as follows:

```
function turnRight() {
   repeat (3) {
      turnLeft();
   }
}
```

- The following function creates a square of four beepers, leaving Karel in its original position:

```
function makeBeeperSquare() {
   repeat (4) {
      putBeeper();
      move();
      turnLeft();
   }
}
```

Conditions in Karel

- Karel can test the following conditions:

positive condition	*negative condition*
`frontIsClear()`	`frontIsBlocked()`
`leftIsClear()`	`leftIsBlocked()`
`rightIsClear()`	`rightIsBlocked()`
`beepersPresent()`	`noBeepersPresent()`
`beepersInBag()`	`noBeepersInBag()`
`facingNorth()`	`notFacingNorth()`
`facingEast()`	`notFacingEast()`
`facingSouth()`	`notFacingSouth()`
`facingWest()`	`notFacingWest()`

The `while` Statement

- The general form of the **`while`** statement looks like this:

```
while (condition) {
   statements to be repeated
}
```

- The simplest example of the **`while`** statement is the function **`moveToWall`**, which comes in handy in lots of programs:

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

The `if` and `if-else` Statements

- The **`if`** statement in Karel comes in two forms:
  - A simple **`if`** statement for situations in which you may or may not want to perform an action:

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

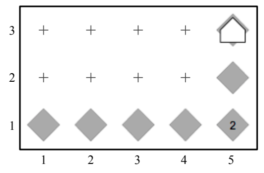
  - An **`if-else`** statement for situations in which you must choose between two different actions:

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

Exercise: Creating a Beeper Line

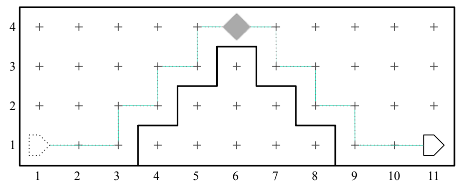
- Write a function **`putBeeperLine`** that adds one beeper to every intersection up to the next wall.
- Your function should operate correctly no matter how far Karel is from the wall or what direction Karel is facing.
- Consider, for example, the following function called **`test`**:

```
function test() {
   putBeeperLine();
   turnLeft();
   putBeeperLine();
}
```



Climbing Mountains

- For the rest of today, we'll explore the use of functions and control statements in the context of teaching Karel to climb stair-step mountains that look something like this:



- The initial version will work only in this world, but later examples will be able to climb mountains of any height.