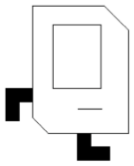
Eric Roberts
CS 208E

Handout #4
October 4, 2016

Karel the Robot

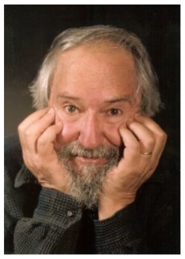
Karel the Robot



Eric Roberts
CS 208E
October 4, 2016

The Project LOGO Turtle

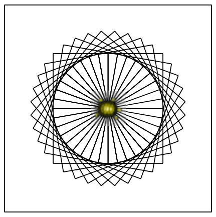
- In the 1960s, the late Seymour Papert and his colleagues at MIT developed the Project LOGO turtle and began using it to teach schoolchildren how to program.
- The LOGO turtle was one of the first examples of a ***microworld,*** a simple, self-contained programming environment designed for teaching.
- Papert described his experiences and his theories about education in his book *Mindstorms,* which remains one of the most important books about computer science pedagogy.



Programming the LOGO Turtle

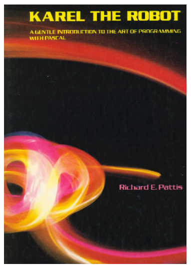
```
to square
  repeat 4
    forward 40
    left 90
  end
end

to flower
   repeat 36
      square
      left 10
   end
end
```



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.



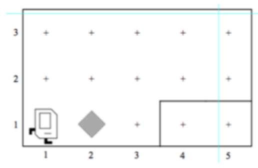
Meet Karel the Robot

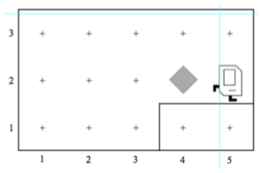
- Karel the Robot was developed here at Stanford by Richard Pattis over 30 years ago. Since then Karel has given many generations of Stanford students a "gentle introduction" to programming and problem solving.
- Karel's world is composed of streets and avenues numbered from the southwest corner. Karel's world is surrounded by a solid wall through which it cannot move. There may also be walls in the interior of the world that block Karel's passage.
- The only other objects that exist in Karel's world are *beepers,* which are small plastic cones that emit a quiet beeping noise.
- Initially, Karel understands only four primitive 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

Your First Challenge

- How would you program Karel to pick up the beeper and transport it to the top of the ledge? Karel should drop the beeper at the corner of 2nd Street and 4th Avenue and then continue one more corner to the east, ending up on 5th Avenue.





The `moveBeeperToLedge` Function

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

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

Defining New Functions

- A Karel program consists of a collection of ***functions***, each of which is a sequence of statements that has been collected together and given a name. The pattern for defining a new function looks like this:

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

- In patterns of this sort, the boldfaced words are fixed parts of the pattern; the italicized parts represent the parts you can change. Thus, every helper function will include the keyword **`function`** along with the parentheses and braces shown. You get to choose the name and the sequence of statements performs the desired operation.

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.

Adding Functions to a Program

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

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

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

Exercise: Defining Functions

- Define a function called **`turnAround`** that turns Karel around 180 degrees without moving.

- 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 three 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 is used to repeat 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
}
```

- Like most 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 affected by the **`repeat`**
- Note that most of the header line appears in boldface, which means that it is a fixed part of the **`repeat`** statement pattern. The only thing you are allowed to change is the number of repetitions, which is indicated by the placeholder *count*.

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
}
```

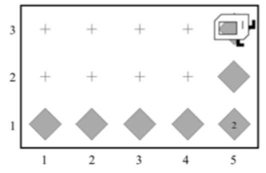
  - An **`if-else`** statement for situations in which you must choose between two different actions:

```
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

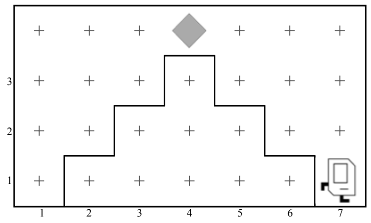
- Write a function **`putBeeperLine`** that adds one beeper to every intersection up to the next wall.
- Your function should operate correctly no matter how far Karel is from the wall or what direction Karel is facing.
- Consider, for example, the following function called **`test`**:

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



Climbing Mountains

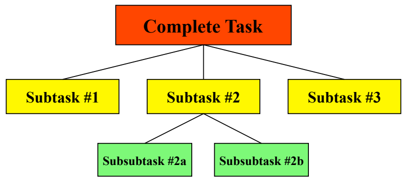
- Our next task explores the use of functions and control statements in the context of teaching Karel to climb stair-step mountains to produce a result that looks something like this:



- Our first program will work only in a particular world, but the goal is to have Karel be able to climb any stair-step mountain.

Stepwise Refinement

- The most effective way to solve a complex problem is to break it down into successively simpler subproblems.
- You start by breaking the whole task down into simpler parts.
- Some of those tasks may themselves need subdivision.
- This process is called ***stepwise refinement*** or ***decomposition***.

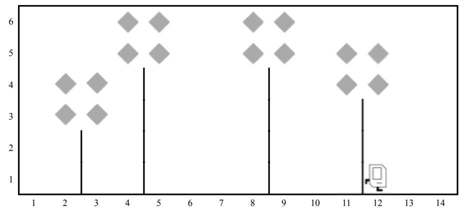


Criteria for Choosing a Decomposition

1. ***The proposed steps should be easy to explain.*** One indication that you have succeeded is being able to find simple names.
2. ***The steps should be as general as possible.*** Programming tools get reused all the time. If your functions perform general tasks, they are much easier to reuse.
3. ***The steps should make sense at the level of abstraction at which they are used.*** If you have a function that does the right job but whose name doesn't make sense in the context of the problem, it is probably worth defining a new function that calls the old one.

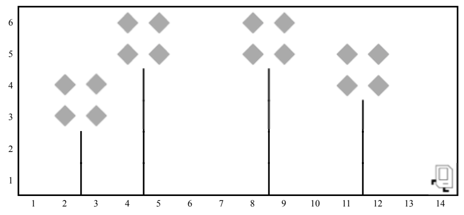
Exercise: Banishing Winter

- In this problem, Karel is supposed to usher in springtime by placing bundles of leaves at the top of each "tree" in the world.
- Given a world with empty trees, Karel should produce this:



Understanding the Problem

- One of the first things you need to do given a problem of this sort is to make sure you understand all the details.
- In this problem, it is easiest to have Karel stop when it runs out of beepers. Why can't it just stop at the end of 1st Street?



The Top-Level Decomposition

- You can break this program down into two tasks that are executed repeatedly:
  - Find the next tree.
  - Decorate that tree with leaves.