CS112 Lecture: Karel Iteration

Last revised 1/9/06

Objectives:

1. To introduce looping
2. To introduce definite iteration
3. To introduce indefinite iteration
4. To introduce reasoning about loops (invariants and termination)

Materials:

1. Projectable Version of Problem 6.10 in the book
2. BlueJ package containing Maze, SimpleEscaperRobot, Maze_while, SimpleEscaper_while, and various worlds (Maze, Maze0, Maze8, Maze20), plus my version of Problem6_10 and HurdlerRobot
3. Demonstration of a robot that executes Euclid's algorithm

I. Introduction

A. Today, we learn the last of our simulated robots' capabilities: iteration (also known as looping.)

B. Alas - this is also the end of our time with Karel; but the concepts we have learned will show up again in Java

C. Several times in the programs we have written, it has been necessary to write an instruction or group of instructions over and over.

Example: In our two maze examples, Karel was told he would need to move exactly 20 blocks to get out of the maze. The main program consisted of 20 repetitions of calling the advanceOneBlock() method.

Clearly, this is an unpleasant and error-prone part of programming.

D. Moreover, there are many situations in which we don't know ahead of time the exact number of repetitions of a block of code that will be required.

Example: Consider trying to write a method that picks up *all* the beepers on a corner, regardless of how many there might be.

E. Fortunately, there is an alternative. The Java language (which is the basis of our robot language) has several instructions that specify that a given block of code is to be repeated some number of times:

1. One form of instruction is used when we know ahead of time the exact number of repetitions that will be needed. This is called *definite iteration*.

2. Another form of instruction is used when we don't know ahead of time the exact number of repetitions that will be needed. This is called *indefinite iteration*. Both the robot language and Java use *while* for this.

II. Definite Iteration

A. Recall both versions of the maze program, in which the escape() method of the robot involved twenty repetitions of calls to the advanceOneBlock() method:

SHOW: SimpleEscaperRobot from last time again

B. This can be revised as follows:

SHOW: Revised version of SimpleEscaperRobot

DEMO

Discussion:

1. A Java for loop header has 3 parts: initialization, test, and increment

2. In this case, in the initialization, we create an integer variable called count, initialized to 0. Notice that we are using count to stand for the number of iterations that have been *completed*.

3. In the test, we continue as long as count is less than 20. (We stop when we have *completed* 20 iterations.)

4. The increment part adds 1 to count. count ++ is Java shorthand for this.

5. The for loop is actually a very powerful construct. For now, we will limit ourselves to a "boilerplate" version like this example.

C. As was true with if, the instruction controlled by for can be a compound instruction.

Example: Recall our bar5 instruction that put down a row of 5 beepers.

This could be revised as follows. (Note that we iterate a pair of instructions 4 times, then do one last putBeeper, since we move 4 times but put down 5 beepers - an example of what Bergin calls "the fence post problem"):

1. Original:

```
public void bar5()
{
     putBeeper();
     move();
     putBeeper();
     move();
     putBeeper();
     move();
     putBeeper();
     move();
     putBeeper();
}
```

2. Revised:

```
public void bar5()
{
     for (int count = 0; count < 4; count ++)
     {
          putBeeper();
          move();
     }
     putBeeper();
}
```

Note: In this case, the loop version is not much shorter, but it is immediately clear how many times each operation is done. The difference would be much more dramatic if wanted to produce a bar 10 or 20 (or more) beepers long!

III. Indefinite Iteration

A. A more flexible form of looping is provided by the while instruction, which allows the number of iterations to be determined by the robot's environment, rather than up-front by the programmer.

1. For example, let's consider our first maze problem, but with the further assumption that Karel is not told ahead of time how many blocks he must move. Instead, he is told that the exit square from the maze is

marked by a beeper; when he gets to it, he's out. (Recall that, for the first maze problem, we didn't use beepers to get Karel through the maze - instead, his instructions were to move straight ahead whenever possible, and to turn left otherwise.)

2. This revised task can be accomplished as follows:

   SHOW: Maze_while.java

   DEMO: - with Maze20.world and Maze8.world

   (**NOTE:** The world file name is specified as a string parameter to main when starting)

B. The iteration provided by the while is called *indefinite iteration*, as opposed to the definite iteration provided by for. (Actually, the Java for can be used for indefinite iteration too - but that's a story for another day!) Indefinite iteration allows a robot's looping to be entirely controlled by its environment. Of course, this opens the possibility of an infinite loop!

   DEMO: Maze_while specifying Maze.world - which contains no beeper

C. One important thing to note about a while loop is that, if the condition it tests is false to begin with, then it will be done zero times.

   DEMO: Maze_while specifying Maze0.world

D. Another example: Problem 6.10 in the book.

   1. The problem: *PROJECT*

   2. Developing the program:

      a) Clearly, Karel is to execute some operation continually until he reaches the beeper that marks the end of the race. Thus, the body of the runRace() method must include a while loop:

         What should the condition in this loop be?

         ASK

         The while condition is while Karel is NOT next to a beeper. We want to stop when we reach a beeper; so we keep going while we have not yet done so. What should comprise the body of the loop?

ASK

Using stepwise refinement, we might decide to have Karel go forward one block each time through the loop. This gives:

```
while (! nextToABeeper())
   advanceOneBlock();
```

b) Now what should comprise the `advanceOneBlock()` method?

ASK

He must do one of two things:

(1) if there is no hurdle in front of him, he can simply move.

(2) otherwise, he has to climb the hurdle.

This gives us the body of the `advanceOneBlock()` method:

```
if (frontIsClear())
      move();
else
      climbHurdle();
```

c) THE REMAINING CODE WILL NOT GO ON THE BOARD, SINCE I SOMETIMES ASSIGN THIS AS A PROJECT.

d) Now, all that is left is to define the method climb-hurdle.

In brief, what Karel must do is this:

(1) Turn left (so he is facing up.)

(2) Move forward until he gets past the top of the hurdle.

(3) Turn right, then move forward to go around the top of the hurdle.

(4) Turn right (so he is facing down.)

(5) Move forward until he reaches the track again.

(6) Turn left (so he is facing his original direction.)

e) Most of these operations can be accomplished directly by primitives; but moving up the top of the hurdle and then moving back down to the track will require more than this, since we do not know how high the hurdle is.

(1) Now when Karel is climbing up, how does he know that he still has to climb further?

ASK

Simply this: so long as there is a wall on his right (his right is NOT clear), he needs to keep moving. NOTE: In the lecture on conditionals, we showed how to develop a leftIsClear() method. A rightIsClear() method is similar.

(2) Now when Karel is climbing down, how does he know that he still has to go down further?

ASK

Simply this: when he reaches the track, he stops - that is, he stops when his front is blocked, so he keeps going while his front is clear.

IV. Loop Invariants and Termination

A. Two more thing we need to consider in conjunction with loops are the concepts of a *loop invariant* and *loop termination*.

B. A loop invariant is a statement about the environment that is true when the loop is first encountered, and remains true after each time through the loop body (though it may be momentarily false inside the body.) As a result, it is necessarily true when the loop terminates as well.

1. That is, a loop invariant has two essential characteristics

a) Establishment (initially)

b) Preservation (if the loop body is entered with the invariant true, it is true upon exit from the loop body)

2. Note that an invariant is totally different from the loop condition

a) The invariant remains true when the loop terminates

b) The condition eventually becomes false, causing the loop to terminate.

3. Obviously, there are many possible loop invariants for any given loop.

   a) Some are trivial and useless.

      Example: in our hurdle problem, a possible invariant for the main program while loop is that there is a beeper somewhere. That, however, doesn't help us much with the loop.

   b) However, the choice of an appropriate invariant can assist in both the design and the correctness-verification of the loop.

      Example: A useful invariant for the hurdle problem main program while loop is that the robot is on the track (1st St) facing toward the goal beeper (east).

      (1) This is true when the while loop is first encountered, because of the way we have set up our problem.

      (2) Each execution of the loop body leaves it true:

         (a) If the robot's front is clear, it simply moves forward. This does not alter the street it is on (since it is facing down the street), nor the direction it is facing. Further, since the while loop condition was true, it was not on the beeper square; therefore, it does not move past it.

         (b) If the robot's front is blocked, the effect of climb-hurdle is still to move it forward one square, so the argument we just gave still applies.

      (3) Notice that the invariant can become momentarily false inside the loop body. In particular, climb-hurdle alters the robot's direction several times, and takes it off 1st St; but we require that when it finishes the robot is back on 1st St facing the right way.

4. What would be a good invariant for the first while loop in climb-hurdle?

   ASK CLASS

   a) "There is a wall on the robot's right" would not be a good invariant. Why?

      ASK CLASS

      the last execution of the while makes this false.

b) A good invariant: the robot is facing up on the avenue just west of a hurdle.

Note how this invariant, coupled with the fact that on loop exit the loop condition is false (so the robot's right side is not blocked) guarantees that the turnRight(); move(); turnRight(); sequence puts it west of the hurdle facing down.

5. What would be a good invariant for the second while?

ASK CLASS

a) The robot is facing down on the avenue just east of the hurdle.

b) Note how this invariant, coupled with the fact that on loop exit his front is blocked, guarantees that the turnLeft() leaves the robot on 1st St facing east, as required by the invariant of the main program while.

C. One last thing we need to consider in conjunction with loops (specifically with while loop) is an approach to convincing ourselves that the loop will, eventually, terminate.

1. Most computer users have had some experience - perhaps painful - with computer systems that "freeze". The most common reason why this occurs is an infinite loop - one that fails to terminate.

2. For each while loop we write, we want to spend some time convincing ourselves that it must always terminate - i.e. that the condition in the loop header will eventually become false.

3. This can often be done by arguing that each iteration of the body moves us closer to the termination condition.

a) Example: main while: each iteration moves us one block down 1st St. Given the loop invariant that the robot is always facing the beeper, this must move us closer to our goal of being on the corner with the beeper, so we're assured of eventually getting there and terminating.

b) Example: first climb-hurdle while: each iteration moves us up one block along the avenue east of the hurdle. If the hurdle is of finite height, we must eventually move past its end - whereupon rightIsClear() becomes true and the loop terminates.

c) *Example:* second climb-hurdle while: each iteration moves us down one block along the avenue west of the hurdle. This must eventually bring us to the boundary wall south of 1st St, and which point frontIsClear() becomes false and the loop terminates.

D. We now consider a very sophisticated illustration of looping: a robot that implements Euclid's GCD algorithm:

gcd(a, b) = if b = 0 then a
else gcd(b, a mod b)

1. DEMONSTRATE

2. PROJECT code