CS112 Lecture: Karel Conditional Instructions

Last Revised 1/17/08

Objectives:

1. To introduce the if () and if () ... else instructions.
2. To introduce the basic tests a robot can execute
3. To show how more complex tests can be achieved by nesting conditionals
4. To discuss ways to improve the readability of if statements

Materials:

1. BlueJ project containing Maze1 and SimpleEscaperRobot, Replant and ReplanterRobot, Survey and SurveyFixerRobot, Maze2 and AssistedEscaperRobot
2. Slide show showing successive improvements to a code example

I. Introduction

A. Thus far, all of the example programs we have done have been written with an advance knowledge of the exact environment in which they are to be executed. Such programs are, in a real sense, rather unintelligent and uninteresting. One mark of intelligence (in a human or a machine) is the ability to sense the environment and respond to it - an ability lacking in the programs we have written thus far.

1. Today, we introduce the conditional instruction, which allows Karel to be programmed to perform a given instruction if and only if a certain condition is met.

2. A robot is capable of testing to see whether certain conditions hold. For example, it can determine if

   a) The way ahead is clear.
   b) There is a beeper (or beepers) on the corner it is on.
   c) There is another robot (or robots) on the corner it is on.
   d) It is currently facing in a specified direction (north, west, south, or east.)
   e) There is a beeper (or beepers) in its beeper bag.

   Further, we will see shortly that it can also determine if the opposites of any of these hold - e.g. it can determine if there are no beepers on the corner he is on.

3. A robot can be told to perform a certain instruction if and only if a certain condition holds.

a) Example: we could tell Karel to pickup a beeper if and only if there is one to pick up as follows:

```
if (karel.nextToABeeper())
    karel.pickBeeper();
```

b) Example: we could tell him to plant a beeper on a corner if there is NOT one there already as follows:

```
if( ! karel.nextToABeeper())
    karel.putBeeper();
```

Note: The ! operator is used to invert the sense of a test, and is read as "NOT"

B. *A Complete Example*:

Karel has been placed in a maze, from which he can escape by moving a total of 20 blocks. The maze is configured so that the only turn he need ever make to get out is a left turn, and he only needs to do this if the way ahead is blocked.

1. SHOW: Maze1 world

2. How would we develop a solution to this problem?

   ASK CLASS

   The problem can be solved if we can simply do 20 repetitions of advancing one block. Each time, the operation that needs to be done is to turn left if necessary, and then move.

3. SHOW Maze1, SimpleEscaperRobot

   DEMO

4. This is a rather simple problem as it stands. We will make it more interesting soon!

C. Another Example:

Consider the following task: yesterday Karel planted a 5 x 6 field of beepers; but during the night some robot raccoons came along and stole a few of the beepers. Today, he needs to go back over the field and replant any spots where the beeper is missing.

SHOW: Replant.world

1. The task could be done easily if we could define a method for replanting a single row and then use it 6 times. We can define a method called checkFurrow() that does this

   SHOW: replant() method of ReplanterRobot

2. The checkFurrow() method, in turn, could be easily defined if we could define a method for handling one corner, and then use it 5 times.

   SHOW: checkFurrow() method definition

3. Clearly, the heart of the task is the method replantBeeperIfNecessary(). How would this be defined?

   ASK CLASS

   SHOW replantBeeperIfNecessary() method

   Note that we have made the replant(), checkFurrow() and replantBeeperIfNecessary() methods (along with moveToNextFurrowOnLeft()/Right()) be methods of a new class called ReplanterRobot(), in which we also included our standard definition for turnRight(). We could also have done this by making ReplanterRobot a subclass of RightTurnerRobot we defined previously - this is just a bit simpler.

4. *DEMO* program

II. The if ... else Instruction

A. The conditional instruction we have worked with thus far - if () - allows one to either do an instruction or skip it, depending on some conditions. Sometimes, instead of just doing something or skipping it, we want to choose between two alternatives depending on the condition.

B. For example, consider the following problem:

1. Karel was hired to survey a piece of land to be used in constructing a new plant for producing more robots. The land is 8 streets x 8 avenues; and he laid out the outline of the plant's foundation on it by planting beepers.

2. During the night, however, an enemy robot came along and tried to mess him up by removing all the beepers he had put down, and also putting down beepers where there had been none (so that now the outline of the plant is marked by vacant corners instead of beepers!) Today, Karel has to go back over his work and undo the damage.

   a) SHOW: Survey world

   b) The problem can be solved as follows:

   SHOW: fixupSurvey() method of SurveyFixerRobot

   SHOW: fixupOneStreet() method of SurveyFixerRobot

   (NOTE: The moveToNextStreetOnLeft() / Right() methods are similar to ones we used in the replant problem.)

   c) Now the crucial method is fixupOneCorner(). Basically, what we want Karel to do is this: if there is already a beeper on the corner then remove it, else put one there. This can be done as follows:

```
public void fixupOneCorner()
{
     if (nextToABeeper())
          pickBeeper();
     else
          putBeeper();
}
```

   SHOW: Code for the above

   DEMO: operation of program

III. Nested Conditions.

A. Now it's time to pull out all the stops and consider some more complex uses of the if instruction. Consider another maze problem.

1. This time, the maze is more complicated, but someone has made life simpler for Karel by leaving behind some clues:

   a) If the path ahead is clear, and there is no beeper on the current corner, then Karel can just go ahead.

   b) If the path ahead is clear, but there is one beeper on the current corner, then Karel should turn left.

   c) If the path ahead is clear, but there are two beepers on the current corner, then Karel should turn right.

   d) If the path ahead is blocked, Karel should turn left unless his left side is also blocked, or there is a beeper on the corner, in which case he should turn right.

2. Because Karel's friend who left the clues for him does not want to get caught,he has asked Karel to pick up the beepers as he encounters them.

3. *SHOW*: Maze2 world

4. The escape method for this task can be the same as for our previous maze task.

   SHOW: AssistedEscaperRobot escape() method - note same as in SimpleEscaperRobot

5. What needs to be different is the advanceOneBlock() method.

   SHOW: advanceOneBlock() method in AssistedEcaperRobot.

   Discussion of features:

   a) The basic syntax of the if and if .. else instructions is

```
if (condition)
      instruction
```

or

```
if (condition)
      instruction
else
      instruction
```

b) The instruction allowed in the true part or in the else part of an if can be any legal Karel instruction - including another if. (We call this *nested if's*)

Example: The if (frontIsClear()) has an if nested in its true part (which in turn has an if nested in its true part) and an if nested in its else part (which in turn has an if nested in its else part.)

c) The instruction allowed in the true part or in the else part of an if must, however, be a **single** Karel instruction. If we want to control several instructions, then we must enclose them in braces ({ }) to make a single, *compound instruction.*

(1) Example: The true part of the first if (nextToABeeper()).

We want to pick up the beeper and then see if we have a second

(2) *Example:* There are two more places where braces are used for this reason (plus two uses for other reasons.)

ASK CLASS

The other two if (nextToABeeper())

(3) Sometimes, we must enclose the true part of an if in braces even when it consists of only a single instruction.

Example: This is the case with the true part of the if (frontIsClear()) which consists of only a single instruction - a nested if. Nonetheless, if we removed the braces, we would have a program that would not work. (I know .. I forgot them originally and it didn't work!.)

Why? *ASK CLASS*

(a) Notice that the inner if is an if with no else. Were it not enclosed in braces, an ambiguity would arise as to whether the else immediately following it belongs to it, or to the outer if. This problem is called the *dangling else* problem, and arises because the else part of an if is optional.

(b) Programming languages like Java typically resolve this ambiguity by pairing the else with the *nearest* if that has none - in this case if (nextToABeeper()) To prevent this wrong interpretation, we use the braces.

(4) There are two sets of braces that is strictly unnecessary, and can be removed without altering the meaning of the program.

ASK

the braces surrounding the clauses of the outermost if. These are present only for readability.

(5) Actually, deeply nested IF's like we have here are hard to read, and should be avoided where possible.

d) There is one more issue we need to consider. The method leftIsClear() is not actually a built-in method. Instead, it is one we have to define.

(1) What would it take to implement this test, given the operations available to us?

ASK

(2) SHOW code for leftIsClear() method.

(3) Note that this method is different from all the others we have defined thus far. How?

ASK

(a) The methods we have used thus far have not returned any information to the object that requests them. (That is why they were declared as void - they return nothing.)

(b) This method must return a true or false value, which is then tested by an if statement. In Java, the data type whose possible values are true and false is called *boolean*, in honor of George Boole, the inventor the the algebra that is used with boolean values.

(c) Notice the return statement in the method body, which serves to both terminate the method and specify the value that is to be sent back.

IV. Transformations of if Statements

A. It is easy to write if statements whose logic is hard for a reader to follow. Thus, where possible, we want to simplify the statements we write to aid readability.

B. The book discussed four transformations that can be applied to improve the readability of if statements.

ASK

1. Test reversal

```
if (condition)
        instruction1
else
        instruction2
```

is equivalent to

```
if (! condition)
        instruction2
else
        instruction1
```

This is especially useful if the we don't want to do anything if the condition is true

Example: Tell a robot to pick up a beeper if it doesn't already have one:

```
if (anyBeepersInBeeperBag())
        -- do nothing
else
        pickBeeper();
```

transforms to:

```
if (! anyBeepersInBeeperBag())
        pickBeeper();
```

2. We will illustrate the remaining three transformations by improving the following body of code (from exercise 5.8)

PROJECT - show just original code

```
if (facingWest())
{
        move();
        turnRight();
        if (facingNorth())
                move();
        turnAround();
}
else
{
        move();
        turnLeft();
        move();
        turnAround();
}
```

3. Another improvement is *bottom factoring*. If the last thing done in both the true and else part is the same, it can be factored out and moved to after the if statement:

   Example: The above with turnAround() factored out of the bottom (*Project* next column in example)

```
if (facingWest())
{
        move();
        turnRight();
        if (facingNorth())
                move();
}
else
{
        move();
        turnLeft();
        move();
}
turnAround();
```

4. A third improvement is *top factoring*. If the first thing done in both the true and else parts is the same - *and it does not affect the test* - it can be factored out and moved to before the if statement

   Example: The above with move() factored out of the top. (*Project* next column in example)

```
move();
if (facingWest())
{
        turnRight();
        if (facingNorth())
                move();
}
else
{
        turnLeft();
        move();
}
turnAround();
```

5. A final improvement is removing redundant tests. If some condition must necessarily be true when we reach it, we don't need to test it (and shouldn't, for reasons of efficiency and clarity.)

   Example:

   In the above, if the robot is facing West and turns right, what way must it be facing?

   ASK

   In light of this fact, we can eliminate the redundant test, to get the following:

   PROJECT next page of Example

```
move();
if (facingWest())
{
        turnRight();
        move();
}
else
{
        turnLeft();
        move();
}
turnAround();
```

(Note: if we can determine that a test must always be false, we can remove it and its then part totally)

6. At this point, one further improvement is possible. What is it? - *ASK*

   Example code with one more statement bottom factored: (*Project)*

```
move();
if (facingWest())
        turnRight();
else
        turnLeft();
move();
turnAround();
```

7. Contrast this with the original code before improvements were made:

   (*Project last column of second page)*