AP Computer Science Name: _____
Karel Chapter 5 Notes (1)

CONDITIONALLY EXECUTING INSTRUCTIONS

Previously, a robot's exact initial situation was known at the start of a task. When we wrote our programs, this information allowed karel to find beepers and avoid running into walls. However, these programs worked only in their specific initial situations. If a robot tried to execute one of these programs in a slightly different initial situation, the robot would almost certainly perform an error shutoff or a runtime error.

The `if` instruction:

The `if` instruction follows the general form.

```
if ( <test> )
{
     <instruction-list>
}
```

In Chapter 1 we briefly discussed the sensory capabilities of robots. We learned that a robot can see walls, hear beepers, determine which direction it is facing, and feel if there are any beepers in its beeper-bag or other robots on its current corner. The conditions that a robot can test are divided according to these same four categories.

Below is a new class with several conditions that robots of this class can test. This class can serve as the parent class of many of your own classes for the remainder of your visit to karel's world. You may use this class in the same way that you use `UrRobot`. You don't need to create it. Since these are the most common types of robot, the name of this class is simply `Robot`. These robots will be able to make good use of IF and other similar statements.

The Robot Class:

```
public class Robot extends UrRobot
{
     public boolean frontIsClear() {…}
     public boolean nextToABeeper() {…}
     public boolean nextToARobot() {…}
     public boolean facingNorth() {…}
     public boolean facingSouth() {…}
     public boolean facingEast() {…}
     public boolean facingWest() {…}
     public boolean anyBeepersInBeeperBag() {…}
}
```

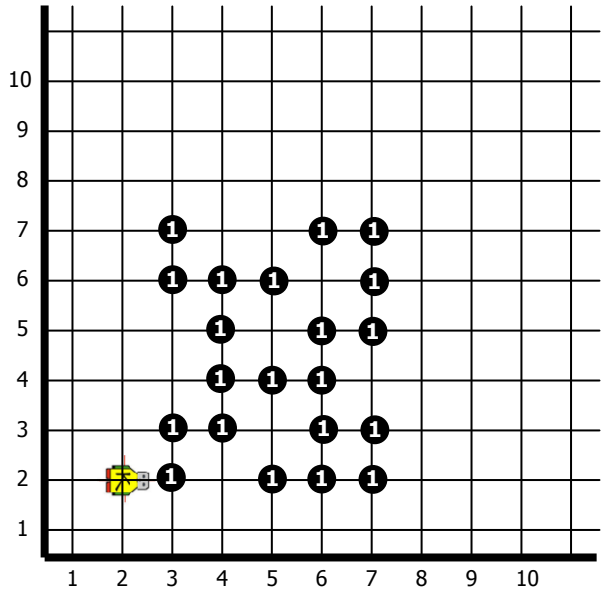
The items in the instruction list name the tests that a robot of the Robot class may perform using its sensors. They return true or false values to the robot mechanism and so we mark them as **boolean**. These methods are called *predicates*. They provide the means by which robots can be queried to decide whether certain conditions are true or false. On the other hand, actions like **move** and **turnOff** are flagged as **void** because the robot gets no feedback information from them. The word **void** indicates the absence of a returned value. In computer programming languages, parts of a program that have a value are called *expressions*. Expressions are usually associated with a *type*, giving the value values of the expression. A predicate represents a *boolean expression*, meaning that its value is either **true** or **false**.

Example 1: `SparseHarvester`

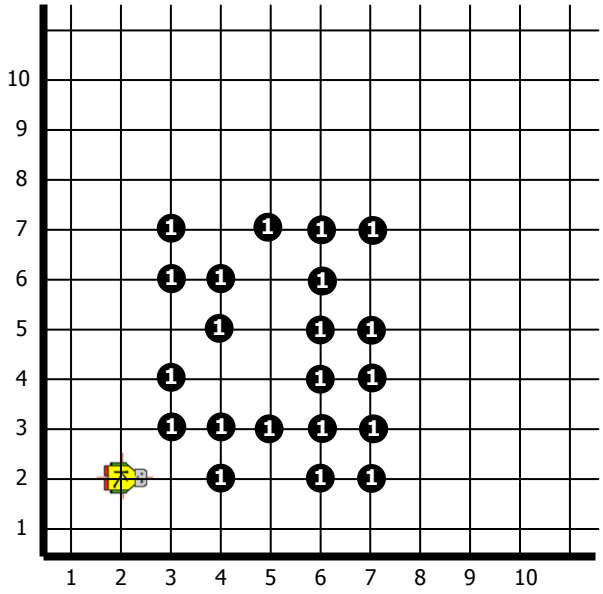
Let's give Karel a new task similar to the harvesting task we did earlier. Karel's new task still requires the robot to harvest the same size field, but this time there is no guarantee that a beeper is on each corner of the field. Because Karel's original program for this task would cause an error shutoff when it tried to execute a `pickBeeper` on any barren corner, we must modify it to avoid executing illegal `pickBeeper` instructions. Karel must harvest a beeper only if it determines that one is present. Name the new class "`SparseHarvester`."

Karel should be able to harvest each of the following fields.

FieldNum1:



FieldNum2:

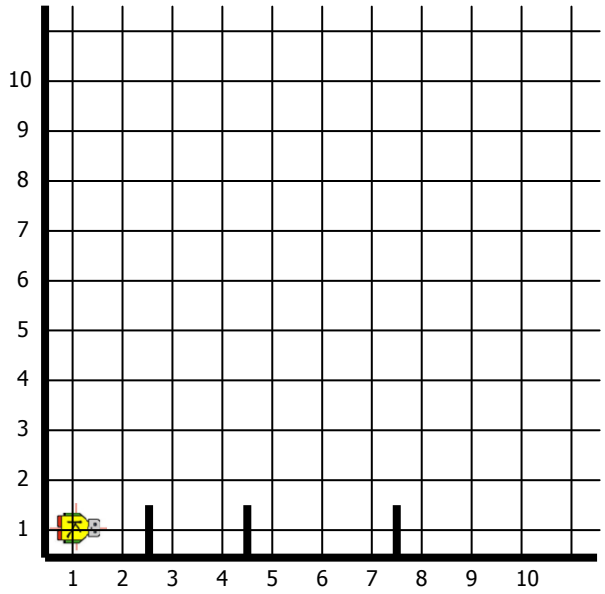


Example 2: The `faceNorthIfFacingSouth()` Method

Example 3: the `faceNorth()` method

Example 4: A Hurdle Jumping Race

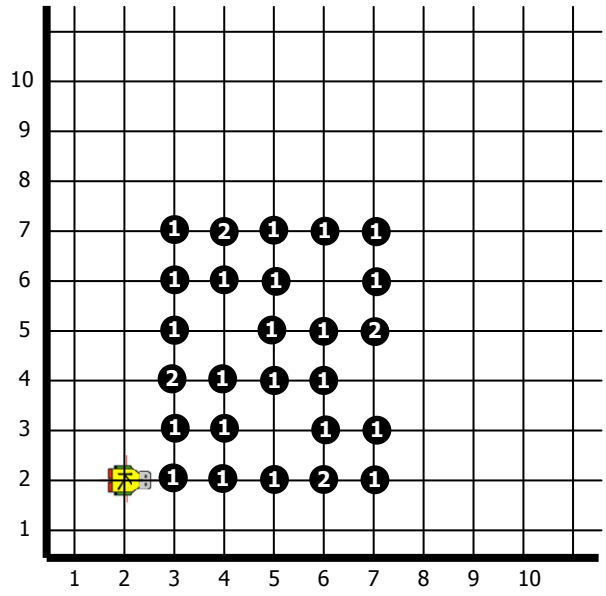
Suppose that we want to program a robot to run a one mile long hurdle race, where vertical wall sections represent hurdles. The hurdles are only one block high and are randomly placed between any two corners in the race course. One of the many possible race courses for this task is illustrated below. Here we think of the world as being vertical with down being south. We require the robot to jump if, and only if, faced with a hurdle.



Example 5: A Beeper Replanting Task

This task requires that a robot named karel traverse a field and leave exactly one beeper on each corner. The robot must plant a beeper on each barren corner and remove one beeper from every corner where two beepers are present. All corners in this task are constrained to have zero, one, or two beepers on them. One sample initial and final situation is displayed below. In these situations, multiple beepers on a corner are represented by a number. We can assume that karel has enough beepers in its beeper-bag to replant the necessary number of corners.

Start:



End: