AP Computer Science Name: ______________________________
Karel Chapter 5 Notes (2)

CONDITIONALLY EXECUTING INSTRUCTIONS

Negation:

We will often want both positive and negative forms of many predicates. For example, we would probably want a predicate `frontIsBlocked` as the negative of `frontIsClear`. Only the positive forms are provided by the Robot class, however. To aid in the writing of such negative forms, we will rely on the logical negation operator. In English this is usually written **not**. In the robot programming language we use the *negation operator*, "!", for this.

For example, we have `nextToABeeper`. If we also want "**not** `nextToABeeper`", what we write is "`!nextToABeeper()`". Any message evaluating a predicate can be "negated" by preceding it with the negation operator, "`!`". Thus, if a robot karel has beepers in its beeper-bag, then it could respond to the instruction

```
     if(! karel.nextToABeeper())    // if karel is NOT next to a beeper
     {
          karel.putBeeper();
     }
```

Writing New Predicates:

While the eight predicates defined above are built into the language, the user can also write new predicates. Predicates return Boolean values, **true** and **false**. Therefore, in the block of the definition of a new predicate we need to indicate what value is to be returned. For this we need a new kind of instruction: the RETURN instruction. The form of the RETURN instruction is the reserved word **return**, followed by an expression. In a `boolean` method the value of the expression must be true or false. RETURN instructions are only legal in predicates. They won't be used in ordinary (void) methods (for now), nor in the main task block.

We might want predicates that are negative forms of the Robot class predicates. They can be defined using the not operator. For example, in a class `CheckerRobot`, we might want the following as well as some others.

```
     public class CheckerRobot extends Robot
     {
          public boolean frontIsBlocked()    {…}

          public boolean notNextToABeeper()   {…}

          public boolean rightIsClear()   {…}

          public boolean facingEastOrWest()  {…}

     }
```

The IF/ELSE Instruction:

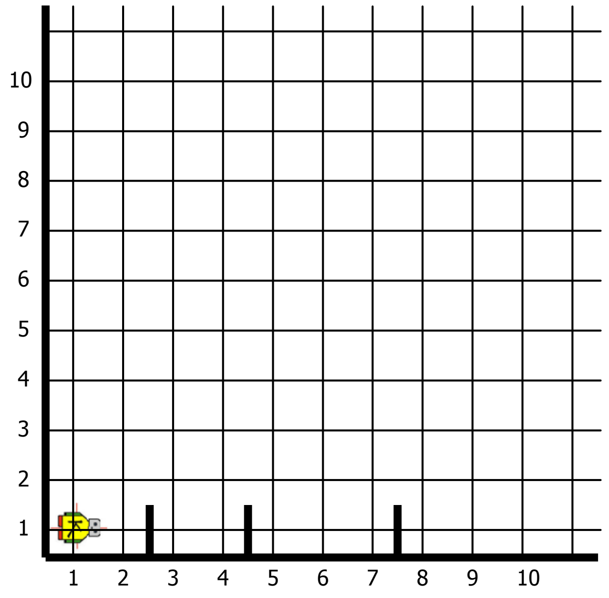
The IF/ELSE instruction is useful when, depending on the result of some test, a robot must execute one of two alternative instructions. The general form of the IF/ELSE is given below:

```
if ( <test> )
{
     <instruction-list-1>
}
else
{
     <instruction-list-2>
}
```

The form of the IF/ELSE is similar to the IF instruction, except that it includes an ELSE clause. Note the absence of a semi-colon before the word else and at the end. A robot executes an IF/ELSE in much the same manner as an IF. It first determines whether <test> is true or false in the current situation. If <test> is true, the robot executes <instruction-list-1>; if <test> is false, it executes <instruction-list-2>. Thus, depending on its current situation, the robot executes either <instruction-list-1> or <instruction-list-2>, but not both. By the way, the first instruction list in an IF/ELSE instruction is called the *THEN clause* and the second instruction is called the *ELSE clause*.

Example 1: 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.



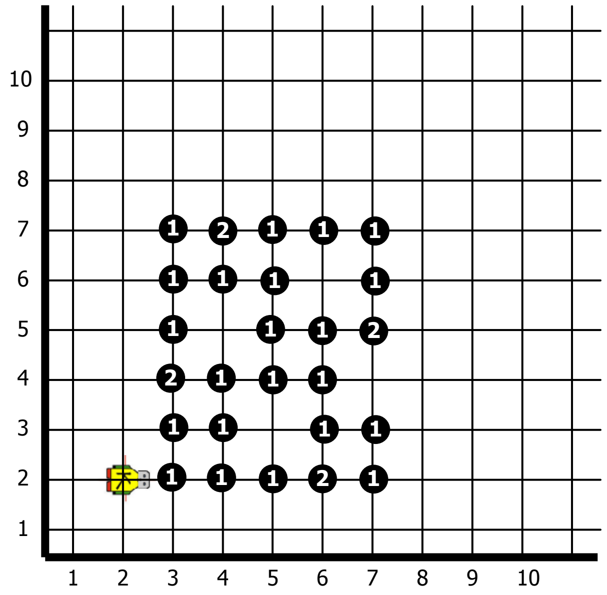
Nested IF Instructions:

Although we have seen many IF instructions, we have ignored an entire class of complex IF'S. These are known as nested IF instructions because they are written with an IF instructed nested inside the THEN or ELSE clause of another IF.

Example 2: 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: