Jerry Cain
CS 106AJ

Handout #23
October 22, 2018

Practice Midterm Examination #1

Review session: **Sunday, October 28, 7:00–9:00P.M., Lathrop 282**
Midterm exams: **Tuesday, October 31, 3:30–5:30P.M., Educ 128**
Tuesday, October 31, 7:00–9:00P.M., 380-380C

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the midterm examination a week from Tuesday. A solution set to this practice examination will be distributed on Monday, along with a second practice exam.

Time and place of the exam

The midterm exam is scheduled at two different times to accommodate those of you who have scheduling constraints. You may take the exam at either time and need not give advance notice of which exam you plan to take. If you cannot take the exam at either of the two scheduled times or if you need special accommodations, please send an email message to **`rebs@stanford.edu`** stating the following:

- The reason you cannot take the exam at either of the scheduled times and/or the details of the special accommodations you require.
- A list of two-hour periods (or longer if you have OAE accommodations) on Tuesday or Wednesday at which you could take the exam. These time blocks must be during the regular working day and must therefore start between 8:30 and end by 5:00.

In order to schedule an alternate exam, we must receive an email message from you by 5:00P.M. on Friday, October 26. Late requests will not be honored. Instructions for taking the alternate midterm will be sent to you via email by Monday, October 29th.

Review session

There will be a review session on Sunday evening from 3:00 to 5:00P.M. in Lathrop 282 at which we will go over the solutions to the practice exams and answer questions.

Coverage

The exam covers the material presented in class through Friday, October 26, which means that you are responsible for the material in Chapters 1 through 7 of the course reader. You may also notice that one problem on each of the two practice midterms is an exercise from the book. The same will be true for the actual midterm. Thus, if you solve all of the exercises in the book 😄, you will have already solved one of the five problems.

Note: To conserve trees, we have cut back on answer space for the practice midterm. The actual exam will have much more room for your answers and for any scratch work.

Please remember that the midterm is <u>open-book.</u>

General instructions

Answer each of the five questions included in the exam. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit.

Each question is marked with the number of points assigned to that problem. The total number of points is 70. We intend for the number of points to be roughly comparable to the number of minutes you should spend on that problem. This leaves you with an additional 50 minutes to check your work or recover from false starts.

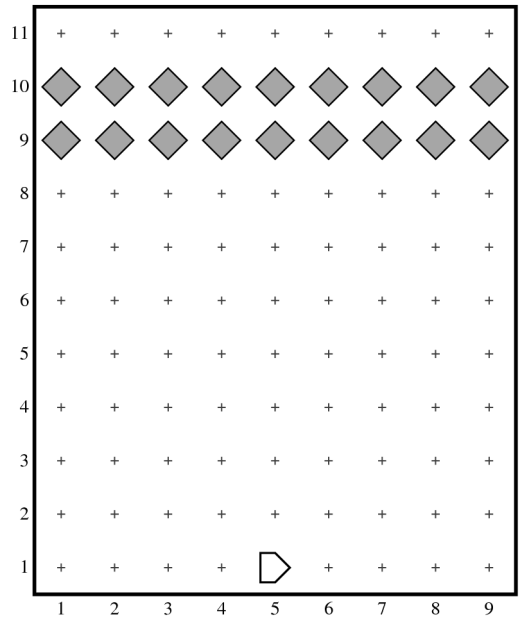
In all questions, you may include methods or definitions that have been developed in the course, either by writing the **`import`** line for the appropriate Karel package or by giving the name of the method and the handout or chapter number in which that definition appears.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit if they help us determine what you were trying to do.

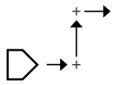
The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

Problem 1: Karel the Robot (10 points)

Not wanting to miss out on all the fun, Karel has decided that it too can learn to play Breakout! Imagine that Karel starts out in a world that looks like this:

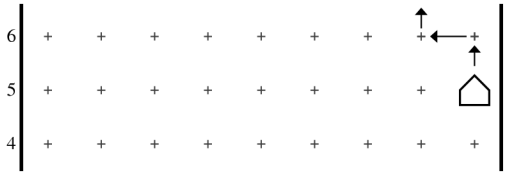


Your job is to teach Karel to play a simple game of Breakout. The first step is to get Karel to move in a more-or-less diagonal line, which of course really requires Karel to move in little stair steps like this:



Fairly soon, Karel will come up against the east wall of the world and find itself in the following position on 5th Street:

Karel now has to bounce, just as in the Breakout game you wrote in JavaScript. Bouncing sounds tricky, but since Karel is always moving at 45°, you can implement a standard bounce operation simply by turning left. Doing so puts Karel in the following position, from which it can again implement its stair-step diagonal motion:



As Karel proceeds, it will eventually hit one of the beepers near the top of the world. When it does, Karel can make it look like Breakout simply by picking the beeper up and then bouncing just as if it had hit a wall. Karel then keeps going, bouncing off beepers and walls (including the south wall, since there is no paddle).

There are two important aspects of this problem that you should keep in mind:

1. If Karel ends up in a corner, it will have to bounce twice—once off each wall—to avoid being blocked. The easiest way to manage this is to have Karel check if it is still blocked after making its first bounce, and, if so, bounce again.
2. Karel cannot simply take both steps in its diagonal motion without checking to make sure that (a) it can move in each of the directions without hitting a wall and (b) that there is no beeper on the square in the middle, which it would surely hit first. Thus, you have to check for bounces halfway through each step as well as at the end.

Write the program to implement this simulation of Breakout. In your solution, you may use any Karel method defined in the course handouts just by giving its name. For example, you may use **`turnRight`** or **`moveToWall`** without writing down the complete definition. You may also assume that Karel always begins facing east on 1st Street and that the world is at least 2×2 in size. You may also make the following assumptions:

1. *Don't worry about getting Karel to stop.* Given that Karel can't count, there isn't any easy way for Karel to determine whether the bricks are gone. The easiest thing to do is to let the program run forever. You can achieve this goal without violating Karel's rules by starting with at least one beeper in the bag and using a loop of the form

```
while (beepersInBag()) {
    Have Karel take one diagonal step, bouncing as appropriate.
}
```

2. *Don't worry about whether Karel can in fact hit all the beepers.* Given that there is no paddle and everything is behaving deterministically, Karel could easily (and indeed eventually does) get stuck in a loop where it just keeps going back and forth without hitting the last few beepers.

Problem 2: Simple JavaScript expressions, statements, and methods (10 points)

(2a) Compute the value of each of the following JavaScript expressions:

```
5 % 4 - 4 % 5
```
_____

```
7 < 9 - 5 && 3 % 0 === 3
```
_____

```
"B" + 3 * 4
```
_____

(2b) Assume that the function **`mystery`** has been defined as given below:

```
function mystery(str) {
   let result = "";
   let len = str.length;
   let j = 0;
   let k = 9;
   while (j < k) {
      if (j < 4) {
         result += str.charAt(k % len);
      }
      if (Math.floor(j / 2) !== 1) {
         result += str.charAt(j % len);
      }
      j++;
      k--;
   }
   return result;
}
```

What is the value of

```
mystery("abcdefg")
```

(2c) What output is printed by a call to **`problem2c()`**?

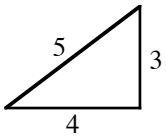
```
/**
 * File: Problem2c.js
 * ------------------
 * This program doesn't do anything useful and exists only to
 * exercise your understanding of parameters and string methods.
 */

function problem2c() {
   let s1 = "To err";
   let s2 = "is human!";
   s1 = forgive(s1, s2);
   console.log(s1 + " " + s2);
}

function forgive(me, you) {
   let heart = me.substring(0, you.length - me.length);
   you = "" + you.charAt(me.length);
   let amount = heart.length;
   me = me.substring(amount + 2) + me.charAt(amount);
   heart += understanding(you, 2) + you + me;
   return heart;
}

function understanding(you, num) {
   return String.fromCharCode(you.charCodeAt(0) + num);
}
```

Problem 3: Simple JavaScript programs (15 points)

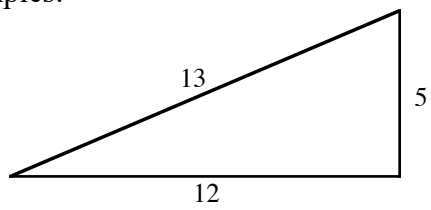
As you undoubtedly learned in school, the Pythagorean theorem holds that the length of the hypotenuse (z) of a right triangle with sides x and y is given by the following formula:

$$x^2 + y^2 = z^2$$

As it turns out, there are an infinite number of triangles in which all three of these edge lengths are integers, including the following examples:



$3^2 + 4^2 = 5^2$



$5^2 + 12^2 = 13^2$

Because of this connection to the Pythagorean theorem, any set of integers x, y, and z that meets this condition is called a ***Pythagorean triple.***

Write a JavaScript program that prints out all Pythagorean triples in which both x and y are less than or equal to a named constant **`MAX`** and x is less than y For example, if **`MAX`** is 25, your program should generate the following sample run:

JavaScript Console

```
3, 4, 5
5, 12, 13
6, 8, 10
7, 24, 25
8, 15, 17
9, 12, 15
10, 24, 26
12, 16, 20
15, 20, 25
18, 24, 30
20, 21, 29
```

In writing this problem, you should not worry at all about efficiency. Trying every possible pairing of x and y and seeing whether it works is perfectly acceptable.

Problem 4: Using graphics and animation (20 points)

Write a graphical program that does the following:

1. Creates the following cross as a **`GCompound`** containing two filled rectangles:

   The color of the cross should be red, as in the emblem of the International Red Cross (it actually is red in the diagram, but that's hard to see on the printed copies, which are of course in black and white). The horizontal rectangle should be 60×20 pixels in size and the vertical one should be 20×60. They cross at their centers.
2. Adds the cross to the graphics window so that it appears at the center.
3. Moves the cross at a speed of 2 pixels every 20 milliseconds in a random direction, which is specified as a random real number between 0 and 360 degrees.
4. Every time you click the mouse inside the cross, its direction changes to some new random direction—again chosen as a random real number between 0 and 360 degrees—but its velocity remains the same. Clicks outside the cross have no effect.

If you were actually to write such a program, you would presumably supply some means of making it stop, such as when the cross moves off the screen. For this problem, just have the program run continuously without worrying about objects moving off screen or how to stop the timer.

Problem 5: Strings (15 points)

A ***spoonerism*** is a phrase in which the leading consonant strings of the first and last words are inadvertently swapped, generally with comic effect. Some examples of spoonerisms include the following phrases and their spoonerized counterparts (the consonant strings that get swapped are underlined):

crushing blow → blushing crow
sons of toil → tons of soil
pack of lies → lack of pies
jelly beans → belly jeans
flutter by → butter fly

In this problem, your job is to write a function

```
function spoonerize(phrase)
```

that takes a multiword phrase as its argument and returns its spoonerized equivalent. For example, you should be able to use your function to duplicate the following console session in which all the examples come from Shel Silverstein's spoonerism-filled children's book *Runny Babbit:*

JavaScript Console

```
> spoonerize("bunny rabbit")
runny babbit
> spoonerize("silly book")
billy sook
> spoonerize("take a shower")
shake a tower
> spoonerize("wash the dishes")
dash the wishes
>
```

In this problem, you are not responsible for any error-checking. You may assume that the phrase passed to **`spoonerize`** contains nothing but lowercase letters along with spaces to separate the words. You may also assume that the phrase contains at least two words, that there are no extra spaces, and that each word contains at least one vowel. What your method needs to do is extract the initial consonant substrings from the first and last words and then exchange those strings, leaving the rest of the phrase alone.

Hint: Remember that you can use methods from the book. The **`findFirstVowel`** and **`isEnglishVowel`** methods from the Pig Latin program will certainly come in handy.