Homework 1

Due midnight, Thursday, 9/11/2014

Please turn in your homework solutions online at `http://www.dci.pomona.edu/tools-bin/cs131upload.php` by the due date.

1. (9 points) **Partial and Total Functions**

   For each of the following function definitions, give the graph of the function. (See the text for a definition of graph – a set of ordered pairs, NOT a picture!) Say whether this is a partial function or a total function on the integers. If the function is partial, say where the function is defined and undefined.

   For example, the graph of `f(x) = if x > 0 then x + 2 else x/0` is the set of ordered pairs $\{\langle x, x+2\rangle \mid x > 0\}$. This is a partial function. It is defined on all integers greater than 0 and undefined on integers less than or equal to 0.

   Functions:

   (a) `f(x) = if x + 2 > 3 then x + 5 else x/0`

   (b) `f(x) = if x < 0 then 1 else f(x - 2)`

   (c) `f(x) = if x = 0 then 1 else f(x - 2)`

2. (20 points) **Deciding Simple Properties of Programs**

   Suppose you are given a (*magical*) function $\texttt{Halt}_\emptyset$ that works with Java (or your favorite Turing-complete language) programs as follows. $\texttt{Halt}_\emptyset$ takes as input a string representing an arbitrary program P written in programming language L, and that requires no input.

   - $\texttt{Halt}_\emptyset(P)$ returns true if program P will halt without reading any input when executed.
   - $\texttt{Halt}_\emptyset(P)$ returns false if program P will not halt when executed with no input.

   You should not make any assumptions about the behavior of $\texttt{Halt}_\emptyset$ on arguments that do not consist of a syntactically correct program.

   (a) Show how to solve the (regular) halting problem using $\texttt{Halt}_\emptyset$. More specifically, write a program `Halt` that reads a program text P as input, reads a single integer n as input, and then decides whether or not P halts when it reads n as input. Such a `Halt` program would have the following form, and it would print yes if P halts when it runs and reads input n and no if P does not halt when it runs and reads input n.

   ```
   P = readString();
   n = readInteger();
   ...
   ```

   You may assume that any program P you are given begins with a read statement that reads a single integer from standard input. Thus P has the form

   ```
   x = readInteger(); Q
   ```

where Q is the rest of the program text, and Q does not perform any input.

Explain your solution by describing how a program solving the halting problem would work. To do this, just describe what replaces ... in the `Halt` program definition above – there is no need to write the program out fully.

Hint: Remember the key is to start with a possible input P and n for the halting problem and to modify the program P to a program P' (whose text depends on both P and n) and show that using $\texttt{Halt}_\emptyset(P)$ *on P' will determine whether the original program P will halt on n.*

(b) On the basis of the previous part, explain why $\texttt{Halt}_\emptyset$ could not be written in Java (or again, any other Turing-complete language).

3. (10 points) **Background**

Consider the compiler-based programming language that you have used the most. Answer the following questions about it:

(a) Describe two programming errors that the compiler identifies and reports while compiling a program in that language.

(b) Describe two programming errors that can cause your program to halt with an error message or crash after you compile and start to run it.

(c) What do you find to be the most difficult aspect of writing and debugging programs in that language?

(d) Does the language have any features that you rarely or never use? If so, why do you not use them?

4. (20 points) **Grace**

Read the Grace papers on the CS131 "LInks to useful information" web page before attempting the following programs. Use the Grace Draft Specification as a reference.

The simplest way to run Grace programs is to use the minigrace compiler that emits javascript code. It can be obtained by loading the web page at `http://www.cs.pomona.edu/~kim/minigrace/`. I strongly urge you to use Firefox for this as different browsers render the pages differently and the software has been tested most extensively in Firefox.

Start a new program by clicking on the green "New" button, and give it a name ending with ".grace". Please use the names suggested in the problem for the file so that we can easily test them. While you should write test code for your programs, they should go in a separate file that you may name as you like. However, we will be using our own test code for your programs and it will expect your files and classes, methods, etc. to have exactly the names and types specified in the problem. You will lose points if your code does not follow those conventions.

Type your program in the editor pane in the top right portion of the window. You can compile it by clicking on the "Build" button just below the editor pane. When it changes to "Run", click on that to run your program.

If you wish to load an existing program, click on "Upload" and select the file you wish to load. To save your program right-click (e.g., on the Mac hold down the "control" key and press) the "Download" button. From the pop-up menu, select "Save Link As ..." and then navigate to where you want to save your program. Be sure the name is correct before you click "save".

Output from your program appears in the bottom right pane in the window (just below the build/run button). Error messages appear in the same box.

Warning: Minigrace files may not contain tabs. The only white spaces allowed are spaces and returns (generated by the "enter" key). You will receive error messages if your file contains any other white space.

(a) Write a Grace program that defines a class representing mutable (updatable) sets of Numbers. Your sets should implement the following type:

```
type NumberSet = {
    contains(n:Number) -> Boolean
    add(n:Number) -> Done
    union(s:NumberSet) -> Done
    intersection(s:NumberSet) -> Done
    isEmpty -> Boolean
    asString -> String
    do(action) -> Done
}
```

Notice that the operations `add`, `union`, and `intersection` all modify the existing set rather than create a new one. Please name the file `Sets.grace`. Your class should be named `aSet.new`, take no parameters, and should construct an empty list.

Grace actually has a library to represent sets, but I don't want you to use it. Your implementation should use an immutable list (use our implementation from class – available on the class web page) as an instance variable to hold the elements of the list. (Do NOT inherit from it.) Remember that sets do not have duplicates of elements, so your list implementation should not include duplicates.

Show that your program works by building a couple of sets, exercise the operations (showing corner cases work) and printing the results.

Because this program represents mutable sets, there is no need for separate representations for empty and non-empty sets. Those variations will be taken care of by the list instance variable.

(b) Write a Grace program that represents a tree holding a number at each vertex. I suggest that you emulate the implementation of lists from class (remember that Grace does not have a null element!). Here is the type of the tree (and a type for arguments of the methods):

```
type UnaryFunction<T> = {
   apply(n:Number) -> T
}

type Tree = {
    size -> Number
    height -> Number
    isEmpty -> Boolean
    map(blk:UnaryFunction<Number>) -> Tree
    doInorder(blk:UnaryFunction<Done>) -> Done
 }
```

Your program should define an empty tree object named `emptyTree` and contain a class `aTree.root(n:Number)left(l:Tree)right(r:Tree)` that constructs a new class with root `n` and left and right subtrees given by `l` and `r`. Place your code in a file named `Trees.grace`

As you can see from the operations in the type, the tree is immutable.

As usual, write and include code to test it in a separate file. In particular, build an interesting tree `myTree` and then execute

```
myTree.doInorder {n -> print (n)}
var count:Number := 0
myTree.doInorder {n -> count := count +1}
print(count)
```