Homework 1

Due midnight, Thursday, 9/13/2012

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

1. (10 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 the code for a function $\texttt{Halt}_\emptyset$ that can determine whether a program P that requires no input halts. To be more precise, assume that you are writing a C or Java program that reads in another program P as a string. Your program is allowed to call $\texttt{Halt}_\emptyset$ with the string P as an argument. A call to $\texttt{Halt}_\emptyset(P)$ has the following behavior:

   $\texttt{Halt}_\emptyset(P)$ 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.

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

   Can you solve the halting problem using $\texttt{Halt}_\emptyset$? More specifically, can you write a program `Halt` that reads a program text P as input, reads an integer n as input, and then decides whether 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?

   ```
   Ptext = readString(); // reads text of program P
   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. If you believe that the halting problem can be solved if you are given $\texttt{Halt}_\emptyset$, then explain your answer 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. If you believe that the halting problem cannot be solved using $\texttt{Halt}_\emptyset$, then outline a proof of why not.

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/`.

   You can run a program by loading it into the editing box in the upper left quadrant of the web page and then pressing the "Go" button. Output appears in the upper right quadrant of the page. Information about the compilation as well as error messages appear in the box on the left center of the window.

   Programs may be pasted in the editor window, but we recommend storing your programs as files and then loading them using the buttons in the lower left of the window. Click "Add a file" and then "Choose file" to load the program file (which should have a ".grace" suffix). When you load the file, the name should appear in the "Module Name" field. (If you are using the Safari web browser there is a funny glitch that adds "Cfakepath" as a prefix to the file name, but you can edit that prefix out to get the original name. You may find it easier just to use Firefox or some other browser.)

   If you use this technique to load your program, then each time you edit the original file (not the edit window!), you can just click on "Reload" to put the revised file in the edit window ready to run.

   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) **Warning: This problem has been revised.**
   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.

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 lists, 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:

```
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
 }
```

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

As usual, include code to test it. 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)
```