UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Monday 7 December 2009

14:30 to 16:30

Convener: J Bradfield
External Examiner: A Preece

INSTRUCTIONS TO CANDIDATES

1. **Note that ALL QUESTIONS ARE COMPULSORY.**
2. **DIFFERENT QUESTIONS MAY HAVE DIFFERENT NUMBERS OF TOTAL MARKS. Take note of this in allocating time to questions.**

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. (a) Write a function to `f :: String -> Bool` to verify that all the digits in a string are equal to or greater than 5. For example,

```
f "normal text"    == True
f "number 75"      == True
f ""               == True
f "17 is a prime"  == False
```

Your definition may use *basic functions*, *list comprehension*, and *library functions*, but not recursion. [*12 marks*]

(b) Write a second function `g :: String -> Bool` that behaves like `f`, this time using *basic functions* and *recursion*, but not list comprehension or other library functions. [*12 marks*]

(c) Write a third function `h :: String -> Bool` that also behaves like `f`, this time using one or more of the following higher-order library functions:

```
map    :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldr  :: (a -> b -> b) -> b -> [a] -> b
```

You may also use *basic functions*, but not list comprehension, other library functions, or recursion. [*12 marks*]

2. (a) Write a polymorphic function `p :: [a] -> [a]` that swaps every two items in a list. Your function should swap the first with the second item, the third with the fourth, and so on. You may assume that the length of an input list is even. For example:

```
p "swapping" == "wspaipgn"
p [1,2,3,4]  == [2,1,4,3]
p []         == []
```

Your function may use *basic functions*, *list comprehension*, and *library functions*, but not recursion. [*16 marks*]

(b) Write a second function `q :: [a] -> [a]` that behaves like `p`, this time using *basic functions* and *recursion*, but not list comprehension or library functions. Remember that you can define auxiliary functions, if you wish. [*16 marks*]

3. We introduce a data type to represent collections of points in a grid:

```
type Point = (Int,Int)
data Points = Lines Int Int
            | Columns Int Int
            | Union Points Points
            | Intersection Points Points
```

The grid starts with `(0,0)` in the top left corner. The first coordinate of a point represents the horizontal distance from the origin, the second represents the vertical distance.

The constructor `Lines` selects all points on a given range of lines (inclusive). For example,

```
Lines 2 4
```

selects all points at a vertical distance of 2, 3 or 4 from the origin:

```
(0,2)  (1,2)  (2,2)  (3,2)  (4,2)  (5,2)  etc.
(0,3)  (1,3)  (2,3)  (3,3)  (4,3)  (5,3)  etc.
(0,4)  (1,4)  (2,4)  (3,4)  (4,4)  (5,4)  etc.
```

The constructor `Columns` selects all points on a given range of columns (inclusive). For example,

```
Columns 5 7
```

selects all points at a horizontal distance of 5, 6 or 7 from the origin:

```
(5,0)  (6,0)  (7,0)
(5,1)  (6,1)  (7,1)
(5,2)  (6,2)  (7,2)
(5,3)  (6,3)  (7,3)
 etc.   etc.   etc.
```

The constructor `Union` combines two collections of points; for example,

```
Union (Lines 1 2) (Columns 1 3)
```

selects the following points:

```
       (1,0)  (2,0)  (3,0)
(0,1)  (1,1)  (2,1)  (3,1)  (4,1)  (5,1)  etc.
(0,2)  (1,2)  (2,2)  (3,2)  (4,2)  (5,1)  etc.
       (1,3)  (2,3)  (3,3)
       (1,4)  (2,4)  (3,4)
        etc.   etc.   etc.
```

The constructor `Intersection` selects only points that occur in *both* collections. For example,

```
Intersection (Lines 2 3) (Columns 4 5)
```

selects only the points that are on the second or third line, and at the same time on the 4th and 5th column. In other words, it selects the following four points:

```
(4,2)   (5,2)
(4,3)   (5,3)
```

(a) Write a function

```
inPoints :: Point -> Points -> Bool
```

to determine whether a point is in a given collection. For example:

```
inPoints (5,1) (Lines 1 2)   == True
inPoints (5,4) (Lines 1 2)   == False
inPoints (5,1) (Columns 4 5) == True
inPoints (1,2) (Columns 4 5) == False
inPoints (1,2) (Union (Lines 1 2)
                      (Columns 7 8))  == True
inPoints (5,4) (Union (Lines 1 2)
                      (Columns 7 8))  == False
inPoints (1,2) (Intersection (Lines 2 3)
                             (Columns 0 1)) == True
inPoints (1,1) (Intersection (Lines 2 2)
                             (Columns 2 2 )) ==False
```

[16 marks]

(b) Write a function

```
showPoints :: Point -> Points -> [String]
```

to show a collection of points as a list of strings, representing the points on a grid. The grid starts with `(0,0)` in the top left corner, while the bottom right corner, which determines the size of the grid, is given by the first argument to the function `showPoints`. The strings in the list that is returned should correspond to the rows (not the columns) of the grid. Use an asterisk ('*') to represent a point, and use blank space (' ') to fill out the lines. For example:

```
showPoints (4,3) (Union (Lines 1 1) (Columns 2 3)) ==
  ["  ** ",
   "*****",
   "  ** ",
   "  ** "]
showPoints (4,2) (Intersection (Lines 1 1) (Columns 2 3)) ==
  ["     ",
   "  ** ",
   "     "]
```

[*16 marks*]