UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Monday 7 December 2009

09:30 to 11:30

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 every vowel in a string is uppercase. In English, the following characters are the vowels: `'a'`, `'A'`, `'e'`, `'E'`,`'i'`, `'I'`, `'o'`, `'O'`, `'u'`, and `'U'`. The function should return `True` for strings that have no vowels at all. For example,

```
f "ALL CAPS"    == True
f "r3cURsI0n"   == True
f []            == True
f "normal text" == 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] -> [a]` that *interleaves* two lists. The result should start with the first item of the first list, then the first of the second list, then the second of the first, and so on. If one list is longer than the other, the remainder of the longer list should be kept as the tail of the interleaved list. For example:

```
p "itrev" "nelae" == "interleave"
p "" "justalist"  == "justalist"
p [0,0,0,0,0,0] [1,2,3] == [0,1,0,2,0,3,0,0,0]
```

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

(b) Write a second function `q :: String -> String -> String` 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 = Rectangle Point Point
            | Union Points Points
            | Difference 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 `Rectangle` selects all points in a rectangular area. For example,

```
Rectangle (0,0) (2,1)
```

gives the top left and bottom right corners of a rectangle, and represents all the points in between (inclusive):

```
(0,0)  (1,0)  (2,0)
(0,1)  (1,1)  (2,1)
```

Secondly, `Union` combines two collections of points; for example,

```
Union (Rectangle (0,0) (1,1)) (Rectangle (1,0) (2,1))
```

represents the same collection of points as above. Finally, the constructor `Difference` selects those points that are in the first collection but not in the second. For example:

```
Difference (Rectangle (0,0) (2,2)) (Rectangle (0,2) (3,2))
```

again gives the same collection of points as above.

(a) Write a function

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

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

```
inPoints (1,1) (Rectangle (0,0) (2,1))  == True
inPoints (3,4) (Rectangle (0,0) (2,1))  == False
inPoints (1,1) (Union (Rectangle (0,0) (0,1))
                      (Rectangle (1,0) (1,1))) == True
inPoints (2,2) (Union (Rectangle (0,0) (0,1))
                      (Rectangle (1,0) (1,1))) == False
inPoints (1,1) (Difference (Rectangle (0,0) (1,1))
                           (Rectangle (0,0) (0,1))) == True
inPoints (0,0) (Difference (Rectangle (0,0) (1,1))
                           (Rectangle (0,0) (0,1))) == 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 (4,2) (Rectangle (1,1) (3,3) ==
  ["     ",
   " *** ",
   " *** "]
showPoints (5,2) (Difference (Rectangle (0,0) (4,1))
                             (Rectangle (2,0) (2,2))) ==
  ["** ** ",
   "** ** ",
   "      "]
```

[*16 marks*]