UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Monday 13 th September 2010

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 every punctuation character in a string is a space. A character is punctuation if it is not a letter or a digit. The function should return `True` for strings that have no punctuation characters at all. For example,

```
f "Just two spaces"                == True
f "No other punctuation, period."  == False
f "No exclamations!"               == False
f "What the @#$!?"                 == False
f "l3tt3rs and d1g1ts 0k"          == True
f "NoSpacesAtAllOK"                == True
f ""                               == True
```

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 *intersperses* an item among the elements of another list. The result list should begin with the item, followed by the first element of the list, followed by the item, followed by the second element of the list, followed again by the item, and so on, ending with the last element of the list followed again by the item. If the list is empty, it should return a list just containing the item. For example:

```
p 'x' "ABCD"        ==  "xAxBxCxDx"
p 'a' "XY"          ==  "aXaYa"
p '-' "Hello"       ==  "-H-e-l-l-o-"
p '_' ""            ==  "_"
p 0 [1,2,3,4,5]     ==  [0,1,0,2,0,3,0,4,0,5,0]
```

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

(b) Write a second function `q :: [a] -> [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 = X
            | Y
            | DX Int Points
            | DY Int Points
            | U Points Points
```

The grid has `(0,0)` in the middle. The first coordinate of a point (the x coordinate) represents the horizontal distance from the origin, the second (the y coordinate) represents the vertical distance. As usual, we will visualise x coordinates as increasing to the right, and y coordinates as increasing moving up.

The constructor `X` selects all points on the x axis, that is, points with any x coordinate and with a y coordinate of zero:

```
... (-2,0)  (-1,0)  (0,0)  (1,0)  (2,0)  ...
```

The constructor `Y` selects all points on the y axis, that is, points with an x coordinate of zero and any y coordinate:

```
...
(0,2)
(0,1)
(0,0)
(0,-1)
(0,-2)
...
```

The constructor `DX dx p`, where `dx` is an integer and `p` is itself a set of points, represents the points `p` shifted left by the amount `dx`. For example, `DX 2 Y`, shifts the y axis two units to the right:

```
...
(2,2)
(2,1)
(2,0)
(2,-1)
(2,-2)
...
```

Note that `DX dx X` and `X` represent the same set of points for any `dx`, since shifting the x axis along itself results in just the x axis. Also, note that a point `(x,y)` belongs to points `DX dx p` exactly when the point `(x-dx,y)` belongs to points `p`.

The constructor `DY dy p`, where `dy` is an integer and `p` is itself a set of points, represents the points `p` shifted up by the amount `dy`. For example, `DY 3 X`, shifts the x axis three units upward:

```
... (-2,3)  (-1,3)  (0,3)  (1,3)  (2,3)  ...
```

Note that `DY dy Y` and `Y` represent the same set of points for any `dy`, since shifting the y axis along itself results in just the y axis. Also, note that a point `(x,y)` belongs to points `DY dy p` exactly when the point `(x,y-dy)` belongs to points `p`.

Finally, the constructor `U p q`, where both `p` and `q` are themselves sets of points represents the union of the points in both `p` and `q`. For example,

```
U (U X Y) (U (DY 3 X) (DX 2 Y))
```

represents the following set of points:

```
                    ...           ...
                    (0,4)         (2,4)
... (-2,3)  (-1,3)  (0,3)  (1,3)  (2,3)  ...
                    (0,2)         (2,2)
                    (0,1)         (2,1)
... (-2,0)  (-1,0)  (0,0)  (1,0)  (2,0)  ...
                    (0,-1)        (2,-1)
                    (0,-2)        (2,-2)
                    ...           ...
```

(a) Write a function

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

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

```
inPoints (3,0) X                                  ==  True
inPoints (0,1) Y                                  ==  True
inPoints (3,3) (DY 3 X)                           ==  True
inPoints (2,1) (DX 2 Y)                           ==  True
inPoints (3,0) (U X Y)                            ==  True
inPoints (0,1) (U X Y)                            ==  True
inPoints (3,3) (U (DY 3 X) (DX 2 Y))              ==  True
inPoints (2,1) (U (DY 3 X) (DX 2 Y))              ==  True
inPoints (3,0) (U (U X Y) (U (DX 2 Y) (DY 3 X)))  ==  True
inPoints (0,1) (U (U X Y) (U (DX 2 Y) (DY 3 X)))  ==  True
inPoints (3,3) (U (U X Y) (U (DX 2 Y) (DY 3 X)))  ==  True
inPoints (2,1) (U (U X Y) (U (DX 2 Y) (DY 3 X)))  ==  True
inPoints (1,1) X                                  ==  False
inPoints (1,1) Y                                  ==  False
inPoints (1,1) (DY 3 X)                           ==  False
inPoints (1,1) (DX 2 Y)                           ==  False
inPoints (1,1) (U X Y)                            ==  False
inPoints (1,1) (U X Y)                            ==  False
inPoints (1,1) (U (DY 3 X) (DX 2 Y))              ==  False
inPoints (1,1) (U (DY 3 X) (DX 2 Y))              ==  False
inPoints (1,1) (U (U X Y) (U (DX 2 Y) (DY 3 X)))  ==  False
```

[*16 marks*]

(b) Write a function

```
countAxes :: Points -> Int
```

that counts the number of times `X` or `Y` appears in the representation of a point set. Each axis should be counted once for each time it appears. For example:

```
countAxes X                                  ==  1
countAxes Y                                  ==  1
countAxes (U X Y)                            ==  2
countAxes (U (DY 3 X) (DX 2 Y))              ==  2
countAxes (U (U X Y) (U (DX 2 Y) (DY 3 X)))  ==  4
countAxes (U (U X Y) X)                      ==  3
```

[*16 marks*]