UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08013 INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Tuesday 20th December 2016

09:30 to 11:30

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.**
3. **This is an OPEN BOOK examination: notes and printed material are allowed, and USB sticks (read only), but no electronic devices.**
4. **CALCULATORS MAY NOT BE USED IN THIS EXAMINATION**

Convener: I. Simpson
External Examiner: I. Gent

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. (a) Write a function `f :: [Int] -> [Int] -> Int` that computes the sum of the numbers in its first argument that are divisible by the number at the corresponding position in its second argument. If the lengths of the two lists do not match, the extra elements in the longer list are ignored. Assume that none of the numbers in the second argument are 0. For example:

```
f [6,9,2,7] [2,3,5,1]       = 22
f [6,9,2] [2,3,5,1]         = 15
f [1,2,3,4,5] [5,4,3,2,1]   = 12
f [10,20,30,40] [3,4,5,6,7] = 50
```

Use *basic functions*, *list comprehension*, and *library functions*, but *not recursion*. Credit may be given for indicating how you have tested your function. [*16 marks*]

(b) Write a second function `g :: [Int] -> [Int] -> Int` that behaves like `f`, this time using *basic functions* and *recursion*, but *not list comprehension* or *library functions*. Credit may be given for indicating how you have tested your function. [*16 marks*]

2. (a) Write a function `p :: String -> Int` that returns the maximum decimal digit in a string, with a result of 0 if the string contains no digits. For example:

```
p "Inf1-FP"      = 1
p "Functional"   = 0
p "1+1=2"        = 2
p "3.157/3 > 19" = 9
```

Use *basic functions*, *list comprehension*, and *library functions*, but *not recursion*. Credit may be given for indicating how you have tested your function. [*12 marks*]

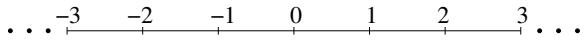
(b) Write a second function `q :: String -> Int` that behaves like `p`, this time using *basic functions* and *recursion*, but *not list comprehension* or *library functions*. Credit may be given for indicating how you have tested your function. [*12 marks*]

(c) Write a third function `r :: String -> Int` that also behaves like `p`, 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 use basic functions but do *not* use *recursion*, *list comprehension* or *library functions* other than these three. Credit may be given for indicating how you have tested your function. [*12 marks*]

3. This question concerns simple commands for moving a robot back and forth along an infinite line like this:



The following declarations define the commands:

```
data Move =
     Go Int            -- move the given distance in the current direction
   | Turn              -- reverse direction
   | Dance             -- dance in place, without changing direction

data Command =
     Nil               -- do nothing
   | Command :#: Move  -- do a command followed by a move
```

Before and after each move, the robot is in some "state" — in a position on the line, and facing left or right:

```
type Position = Int
data Direction = L | R
type State = (Position, Direction)
```

The above declarations are provided in the template file, together with code to print values of type `Move`, `Command` and `State` and to compare them for equality, along with code that enables QuickCheck to generate arbitrary values of type `Move`, `Command` and `State`, to aid testing.

(a) Write a function `state :: Move -> State -> State` that, given a move and the current state of the robot, returns the state of the robot following the move. For example:

```
state (Go 3) (0,R) = (3,R)
state (Go 3) (0,L) = (-3,L)
state Turn (-2,L)  = (-2,R)
state Dance (4,R)  = (4,R)
```

Credit may be given for indicating how you have tested your function. [*8 marks*]

(b) When a robot moves according to the directions given in a command, it goes through a sequence of states, starting with its original state and ending with its final state. Write a function `trace :: Command -> State -> [State]` that computes this sequence of states. For example:

QUESTION CONTINUES ON NEXT PAGE

```
trace (Nil) (3,R) = [(3,R)]
trace (Nil :#: Go 3 :#: Turn :#: Go 4) (0,L)
                  = [(0,L),(-3,L),(-3,R),(1,R)]
trace (Nil :#: Go 3 :#: Dance :#: Turn :#: Turn) (0,R)
                  = [(0,R),(3,R),(3,R),(3,L),(3,R)]
trace (Nil :#: Go 3 :#: Turn :#: Go 2 :#: Go 1 :#: Turn :#: Go 4) (4,L)
                  = [(4,L),(1,L),(1,R),(3,R),(4,R),(4,L),(0,L)]
```

Credit may be given for indicating how you have tested your function. [*12 marks*]

(c) Each time the robot reaches a position it has already visited — irrespective of the direction it is facing — it does a dance. Write a function `dancify :: Command -> Command` that takes a command and inserts `Dance` after each move (except immediately after a `Dance` move) that brings the robot to a position it has already visited. For example:

```
dancify Nil = Nil
dancify (Nil :#: Go 3 :#: Turn :#: Go 4)
            = Nil :#: Go 3 :#: Turn :#: Dance :#: Go 4
dancify (Nil :#: Go 3 :#: Dance :#: Turn :#: Turn)
            = Nil :#: Go 3 :#: Dance :#: Turn :#: Dance :#: Turn :#: Dance
dancify (Nil :#: Go 3 :#: Turn :#: Go 2 :#: Go 1 :#: Turn :#: Go 4)
            = Nil :#: Go 3 :#: Turn :#: Dance :#: Go 2 :#: Go 1 :#: Dance
                                              :#: Turn :#: Dance :#: Go 4
```

(Hint: The correct placement of the `Dance` moves doesn't depend on the robot's original state.) Credit may be given for indicating how you have tested your function. [*12 marks*]