UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08025 INFORMATICS 1 - INTRODUCTION TO COMPUTATION

Tuesday 13 th August 2019

14:30 to 16:30

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: D.K.Arvind
External Examiner: J.Gibbons

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. (a) Write a function `f :: String -> Bool` that tests if all the lower case letters in a string are in even numbered positions, where positions begin with 0. For example:

```
f "" = True
f "ALL CAPS" = True
f "I LOvE FuNcTiOnAL pRoGrAmMiNg" = True
f "aLterNaTiNg" = False
f "aLtErNaTiNg" = True
f "WEe" = True
```

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 :: String -> Bool` 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 :: [(Int,Bool)] -> Bool` that tests whether or not the sum of the squares of the first components of the pairs in a list is even, including only pairs where the second component is True. For example:

```
p [] = True
p [(3,False)] = True
p [(7,True)] = False
p [(3,True),(2,True),(5,True)] = True
p [(3,False),(2,True),(5,True)] = False
p [(4,False),(3,True)] = False
```

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 function `q :: [(Int,Bool)] -> Bool` 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 function `r :: [(Int,Bool)] -> Bool` 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. The following data type is used to represent integer constants greater than or equal to zero.

```
type Nat = Int
```

We will only consider integer constants that are greater than or equal to zero.

The following data type represents arithmetic expressions over a single variable, X:

```
data Expr = X                          -- variable
          | C Nat                      -- integer constant >=0
          | Expr :+: Expr              -- addition
          | Expr :*: Expr              -- multiplication
          | Expr :^: Expr              -- exponentiation
```

The following data types represent polynomials in a single variable, X.

```
data Term = Tm Nat Nat   -- coefficient and power, both >= 0
data Poly = Pl [Term]    -- sum of terms
```

For instance, the expression

```
expr0 :: Expr
expr0 =  ((C 1 :*: (X :^: C 0)) :+:
         ((C 2 :*: (X :^: C 1)) :+:
         ((C 3 :*: (X :^: C 2)) :+:
         C 0)))
```

and the polynomial

```
poly0 :: Poly
poly0 =  Pl [Tm 1 0, Tm 2 1, Tm 3 2]
```

both represent the same function `1*X^0 + 2*X^1 + 3*X^2 + 0` of a single variable `X`.

The template file includes code to convert types `Expr`, `Term`, and `Poly` to strings, and code that enables QuickCheck to generate arbitrary values of types `Term` and `Poly` to aid testing.

QUESTION CONTINUES ON NEXT PAGE

(a) Write a function

```
evalExpr :: Expr -> Int -> Int
```

which given an expression and the value of the variable `X` returns the value of the expression. For example

```
evalExpr (C 1 :*: (X :^: C 0)) 5  =  1
evalExpr (C 2 :*: (X :^: C 1)) 5  =  10
evalExpr (C 3 :*: (X :^: C 2)) 5  =  75
evalExpr (C 0) 5   =  0
evalExpr expr0 5   =  86
evalExpr expr0 10  =  321
```

where `expr0` is as above. Credit may be given for indicating how you have tested your function. [*8 marks*]

(b) Write functions

```
evalTerm :: Term -> Int -> Int
evalPoly :: Poly -> Int -> Int
```

which given a term or polynomial and the value of the variable `X` returns the value of the term or polynomial. For example

```
evalTerm (Tm 1 0) 5  =  1
evalTerm (Tm 2 1) 5  =  10
evalTerm (Tm 3 2) 5  =  75
evalPoly (Pl []) 5   =  0
evalPoly poly0 5     =  86
evalPoly poly0 10    =  321
```

where `poly0` is as above. Credit may be given for indicating how you have tested your functions. [*12 marks*]

(c) Write functions

```
exprTerm :: Term -> Expr
exprPoly :: Poly -> Expr
```

which convert a term or a polynomial to the equivalent expression. For example,

```
exprTerm (Tm 1 0)  =  C 1 :*: (X :^: C 0)
exprTerm (Tm 2 1)  =  C 2 :*: (X :^: C 1)
exprTerm (Tm 3 2)  =  C 3 :*: (X :^: C 2)
exprPoly (Pl [])   =  C 0
exprPoly poly0     =  expr0
```

where `poly0` and `expr0` are as above. Credit may be given for indicating how you have tested your functions. [*12 marks*]