UNIVERSITY OF EDINBURGH
COLLEGE OF SCIENCE AND ENGINEERING
SCHOOL OF INFORMATICS

INFR08013 INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Tuesday 19th December 2017

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 \( f : : [\text{Int}] \rightarrow [\text{Int}] \) that produces a list of distances between consecutive numbers in a list, in those cases where the first number is less than the second number. For example:

\[
\begin{align*}
f [4,2,5,6,1,8] &= [3,1,7] \\
f [\_] &= [] \\
f [3] &= [] \\
f [3,3,1,-3] &= [] \\
\end{align*}
\]

Use basic functions, list comprehension, and library functions, but not recursion. Credit may be given for indicating how you have tested your function. \(16 \text{ marks}\)

(b) Write a second function \( g : : [\text{Int}] \rightarrow [\text{Int}] \) 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. \(16 \text{ marks}\)
2. An *initialism* is an abbreviation consisting entirely of capital letters, for example RAM and MP. For the purposes of this question, we will require initialisms to contain at least 2 characters, in order to exclude the words “I” and “A”.

(a) Write a function `isInitialism :: String -> Bool` that tests whether or not a string is an initialism, and a function `p :: [String] -> Int` that computes the combined lengths of all initialisms in a list of strings. For example:

```haskell
isInitialism "A" = False
isInitialism "AWOL" = True
isInitialism "Ltd" = False
p ["I", "played", "the", "BBC", "DVD", "on", "my", "TV"] = 8
p ["The", "DUP", "MP", "is", "not", "OK"] = 7
p ["The", "SNP", "won", "in", "South", "Morningside"] = 3
p [] = 0
```

Use *basic functions*, *list comprehension*, and *library functions*, but not recursion. Credit may be given for indicating how you have tested your functions. [15 marks]

(b) Write functions `isInitialism' :: String -> Bool` and `q :: [String] -> Int` that behave like `isInitialism` and `p`, this time using *basic functions* and *recursion*, but *not list comprehension* or *library functions* other than `length`. Credit may be given for indicating how you have tested your functions. [15 marks]

(c) Write a function `r :: [String] -> Int` that also behaves like `p`, this time using one or more of the following higher-order library functions:

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

You may use basic functions and your function `isInitialism` but do not use recursion, *list comprehension* or *library functions* other than these three and `length`. Credit may be given for indicating how you have tested your function. [6 marks]
3. The following data type represents arithmetic expressions over a single variable, $X$:

```haskell
data Expr = X       -- variable
           | Const Int       -- integer constant $\geq 0$
           | Expr :+: Expr   -- addition
           | Expr :*: Expr   -- multiplication
```

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

The template file includes a function `showExpr :: Expr -> String` which converts expressions into a readable format, and code that enables QuickCheck to generate arbitrary values of type `Expr`, to aid testing.

(a) Write a function `eval :: Expr -> Int -> Int`, which given an expression and the value of the variable $X$ returns the value of the expression. For example,

- `eval ((X :*: Const 3) :+: (Const 0 :*: X)) 2 = 6`
- `eval (X :*: (Const 3 :+: Const 4)) 2 = 14`
- `eval (Const 4 :+: (Const 3 :*: X)) 3 = 13`
- `eval (((Const 1 :*: Const 2) :*: (X :+: Const 1)) :*: Const 2) 3 = 16`

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

(b) We call an expression simple if it contains no applications of multiplication where the left-hand argument is an integer constant.

Write a function `isSimple :: Expr -> Bool` that determines whether or not an expression is simple. For example,

- `isSimple ((X :*: Const 3) :+: (Const 0 :*: X)) = False`
- `isSimple (X :*: (Const 3 :+: Const 4)) = True`
- `isSimple (Const 4 :+: (Const 3 :*: X)) = False`
- `isSimple (((Const 1 :*: Const 2) :*: (X :+: Const 1)) :*: Const 2) = False`

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

*QUESTION CONTINUES ON NEXT PAGE*
(c) Write a function \texttt{simplify :: Expr -> Expr} that converts an expression to an equivalent simple expression by use of the following laws:

\begin{align*}
\text{Const 0 :*: } e &= \text{Const 0} \\
\text{Const 1 :*: } e &= e \\
\text{Const } n :*: e &= e :+: \cdots :+: e \quad (n \text{ times})
\end{align*}

For example,

\begin{align*}
\text{simplify ((X :*: Const 3) :+: (Const 0 :*: X))} &= \\
&= (X :*: \text{Const 3}) :+: \text{Const 0} \\
\text{simplify (X :*: (Const 3 :+: Const 4))} &= \\
&= X :*: (\text{Const 3 :+: Const 4}) \\
\text{simplify (Const 4 :+: (Const 3 :*: X))} &= \\
&= \text{either Const 4 :+: (X :+: (X :+: X))} \\
&\text{or Const 4 :+: ((X :+: X) :+: X)} \\
\text{simplify (((Const 1 :*: Const 2) :*: (X :+: Const 1)) :*: Const 2)} &= \\
&= ((X :+: \text{Const 1}) :+: (X :+: \text{Const 1})) :*: \text{Const 2}
\end{align*}

(Note that further simplifications are possible, using other laws, but \texttt{simplify} should do only those indicated above.) Credit may be given for indicating how you have tested your function. \[16 \text{ marks}\]