UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08013 INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Friday 20 th December 2013

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, but no electronic devices or electronic media.**

Convener: J. Bradfield
External Examiner: C. Johnson

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. (a) Write a function `f :: String -> Int` that converts a list of base-3 digits to the corresponding numerical value. For example:

```
f "201" = (2 * 3^2) + (0 * 3^1) + (1 * 3^0) = 19
f "12" = 5
f "1202" = 47
f "120221" = 430
```

Use *basic functions*, *list comprehension*, and *library functions*, but not recursion. You may assume that the input is a string of base-3 digits. Credit may be given for indicating how you have tested your function.

[**Hint:** Start by reversing the order of the digits in the list.] [*12 marks*]

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

[**Hint:** Start by reversing the order of the digits in the list.] [*12 marks*]

2. (a) Write a function `p :: [Int] -> Bool` that, given a non-empty list of numbers beginning with a non-zero number, returns `True` if each positive number in the list is divisible by the first number in the list. For example:

```
p [2,6,-3,0,18,-17,10]  =  True
p [-13]                 =  True
p [-3,6,1,-3,9,18]      =  False
p [5,-2,-6,3]           =  False
```

The function should give an error if applied to an empty list or to a list whose first element is zero.

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 `q :: [Int] -> 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. [*16 marks*]

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

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

Do not use recursion or list comprehension. Credit may be given for indicating how you have tested your function. [*12 marks*]

3. The following data type represents arithmetic expressions over a single variable:

```
data Expr = X                        -- variable
          | Const Int                -- integer constant
          | Neg Expr                 -- negation
          | Expr :+: Expr            -- addition
          | Expr :*: Expr            -- multiplication
```

The template file includes code that enables QuickCheck to generate arbitary values of type `Expr`, to aid testing.

Note: In this question, you will need to convert integers to and from strings: `show 234 = "234"` and `read "234" :: Int = 234`.

(a) *Reverse Polish Notation* (RPN) is a parenthesis-free way of writing arithmetic expressions. Operators follow all of their operands, like so:

$\mathtt{X} * 3$	in RPN is	`X 3 *`
$-(\mathtt{X} * 3)$	in RPN is	`X 3 * -`
$(5 + -\mathtt{X}) * 17$	in RPN is	`5 X - + 17 *`
$(15 + -(7 * (\mathtt{X} + 1))) * 3$	in RPN is	`15 7 X 1 + * - + 3 *`

Write a function `rpn :: Expr -> [String]` which converts an expression to its equivalent in RPN. We will represent RPN expressions as lists of strings. For instance, for the third example above:

```
rpn ((Const 5 :+: Neg X) :*: Const 17)
```
should produce
```
["5", "X", "-", "+", "17", "*"]
```

Credit may be given for indicating how you have tested your function. [*16 marks*]

(b) The algorithm for evaluating an RPN expression uses a list for storing intermediate results. Starting with an empty list, and given a value for the variable `X`, we scan the expression from left to right until it is exhausted:

- If the next item is the variable `X`, add its value to the front of the list.
- If the next item is a constant, add it to the front of the list.
- If the next item is an operator with n arguments, then remove the first n items from the front of the list, apply the corresponding operation to them, and add the result to the front of the list. If there were fewer than n items on the list, then the original expression was ill-formed.
- If the next item is something else, then the original expression was ill-formed.

At this point, there should be one item on the list, and that is the result; otherwise the original expression was ill-formed.

Implement this as a function `evalrpn :: [String] -> Int -> Int`, where the first argument is the RPN expression and the second argument is the value of `X`. For example:

```
                 evalrpn ["X", "3", "*"] 10  =  30
            evalrpn ["X", "3", "*", "-"] 20  =  -60
evalrpn ["5", "X", "-", "+", "17", "*"] 10  =  -85
evalrpn ["15", "7", "X", "1", "+", "*", "-", "+", "3", "*"] 2
                                             =  -18
```

`evalrpn ["X", "3", "*", "-", "+"] 20` should produce an error, because there are not enough items on the list of intermediate values to perform the final addition.

Credit may be given for indicating how you have tested your function. The template file includes a function `evalExpr :: Expr -> Int -> Int` to evaluate expressions, where `evalExpr` e n produces the value of e when the variable `X` has value n. Evaluation of an expression using `evalExpr` should produce the same result as evaluation of its RPN version using `evalrpn`. [*16 marks*]