UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

# INFR08013 INFORMATICS 1 - FUNCTIONAL PROGRAMMING

**Tuesday 15 th December 2015**

**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: D. K. Arvind
External Examiner: C. Johnson

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. (a) Write a function `p :: [Int] -> Int` that takes a list of time durations in minutes and calculates what hour it is after all those periods of time have passed, *ignoring negative durations*, starting at 1 o'clock and using a 12-hour clock. For example:

```
p []                          = 1
p [-30,-20]                   = 1
p [20,-30,30,14,-20]          = 2
p [200,45]                    = 5
p [60,-100,360,-20,240,59] = 12
p [60,-100,360,-20,240,60] = 1
```

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 :: [Int] -> 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 :: [Int] -> 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
```

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

2. (a) Write a function `f :: String -> String` that removes all but one occurrence of consecutive repeated characters. For example:

```
f "Tennessee"   = "Tenese"
f "llama"       = "lama"
f "oooh"        = "oh"
f "none here"   = "none here"
f "nNnor hEere" = "nNnor hEere"
f "A"           = "A"
f ""            = ""
```

Upper/lower case should be taken into account when comparing characters, as these examples show.

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 -> String` 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*]

3. The following data type represents a simplified form of regular expressions which omits the "star" (repetition) operator:

```
data Regexp = Epsilon                  -- empty
            | Lit Char                 -- character literal
            | Seq Regexp Regexp        -- sequence: r s
            | Or Regexp Regexp         -- choice: r | s
```

Recall that every regular expression describes a set of strings (its "language"), where:

- the regular expression $\varepsilon$ describes only the empty string;
- for any character $A$, the regular expression $A$ describes only the string containing the single character $A$;
- the regular expression $r\ s$ describes all strings consisting of a first part that is described by $r$ followed by a second part that is described by $s$; and
- the regular expression $r|s$ describes all strings that are either described by $r$ or by $s$.

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

The template file also contains the following regular expressions for you to use in testing:

```
r1 = Seq (Lit 'A') (Or (Lit 'A') (Lit 'A')) -- A(A|A)
r2 = Seq (Or (Lit 'A') Epsilon)
         (Or (Lit 'A') (Lit 'B'))           -- (A|e)(A|B)
r3 = Seq (Or (Lit 'A') (Seq Epsilon (Lit 'A')))
         (Or (Lit 'A') (Lit 'B'))           -- (A|(eA)) (A|B)
r4 = Seq (Or (Lit 'A') (Seq Epsilon (Lit 'A')))
         (Seq (Or (Lit 'A') (Lit 'B')) Epsilon)
                                            -- (A|(eA)) ((A|B)e)
r5 = Seq (Seq (Or (Lit 'A') (Seq Epsilon (Lit 'A')))
              (Or Epsilon (Lit 'B')))
         (Seq (Or (Lit 'A') (Lit 'B')) Epsilon)
                                            -- ((A|(eA))(e|B)) ((A|B)e)
r6 = Seq (Seq Epsilon Epsilon)
         (Or Epsilon Epsilon)               -- (ee)(e|e)
```

(a) Write a function `language :: Regexp -> [String]` which, given a regular expression, returns its language in the form of a list without duplicates. For example, referring to the test examples above:

```
language r1 = ["AA"]                    -- A(A|A)
language r2 = ["AA","AB","A","B"]       -- (A|e)(A|B)
language r3 = ["AA","AB"]               -- (A|(eA)) (A|B)
language r4 = ["AA","AB"]               -- (A|(eA)) ((A|B)e)
language r5 = ["AA","AB","ABA","ABB"]   -- ((A|(eA))(e|B)) ((A|B)e)
language r6 = [""]                      -- (ee)(e|e)
```

Credit may be given for indicating how you have tested your function. (**Hint:** you will need to test using an equality on lists that disregards order but not repetitions. An appropriate function `equal` is provided in the template file.) [*16 marks*]

(b) Write a function `simplify :: Regexp -> Regexp` that converts a regular expression to an equivalent simpler regular expression by use of the following laws:

$$\begin{aligned} \varepsilon\, r &= r \\ r\, \varepsilon &= r \\ r|r &= r \end{aligned}$$

For example:

```
simplify r1 = Seq (Lit 'A') (Lit 'A')
                        -- A(A|A) = AA
simplify r2 = r2        -- (A|e)(A|B) is already simplified
simplify r3 = Seq (Lit 'A') (Or (Lit 'A') (Lit 'B'))
                        -- (A|(eA)) (A|B) = A(A|B)
simplify r4 = Seq (Lit 'A') (Or (Lit 'A') (Lit 'B'))
                        -- (A|(eA)) ((A|B)e) = A(A|B)
simplify r5 = Seq (Seq (Lit 'A')
                        (Or Epsilon (Lit 'B')))
                  (Or (Lit 'A') (Lit 'B'))
                        -- ((A|(eA))(e|B)) ((A|B)e) = (A(e|B))(A|B)
simplify r6 = Epsilon   -- (ee)(e|e) = e
```

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