UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08013 INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Tuesday 15 th December 2015

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: C. Johnson

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. (a) Let's regard midnight as belonging to the following day, so "midnight on Monday" is one minute after 23:59 on Sunday.

   Write a function `p :: [Int] -> Int` that takes a list of time durations in hours and calculates what day of the week it is after all those periods of time have passed, *ignoring negative durations*, starting at midnight on Monday. Use numbers to represent the days of the week, with 1 for Monday, 2 for Tuesday, and so on, up to 7 for Sunday. For example:

```
p []                         = 1
p [-30,-20]                  = 1
p [12,-30,7,8,-20]           = 2
p [90,15]                    = 5
p [90,-100,23,-20,54]        = 7
p [90,-100,23,-20,55]        = 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 single occurrences of characters and one of the occurrences of consecutive repeated characters. For example:

```
f "Tennessee"     = "nse"
```

(removing T, e, one occurrence of n, e, one occurrence of s, and one occurrence of e). Some other examples are:

```
f "bookkeeper"    = "oke"
f "llama hooves" = "lo"
f "www.dell.com" = "wwl"
f "ooooh"         = "ooo"
f "nNnone here"  = ""
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 ε 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 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 (Lit 'B')
         (Seq (Lit 'A') (Or (Lit 'C') (Lit 'D')))
                                            -- B(A(C|D))
```

(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 = ["BAC","BAD"]                 -- B(A(C|D))
```

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 `flatten :: Regexp -> Regexp` that converts a regular expression to an equivalent regular expression by use of the following left distributive law:

$$r\,(s|t) \;=\; (r\,s)\,|\,(r\,t)$$

until no further application of this rule is possible. For example:

```
flatten r1 = Or (Seq (Lit 'A') (Lit 'A'))
                (Seq (Lit 'A') (Lit 'A'))
       -- A(A|A) = (AA)|(AA)
flatten r2 = Or (Seq (Or (Lit 'A') Epsilon) (Lit 'A'))
                (Seq (Or (Lit 'A') Epsilon) (Lit 'B'))
       -- (A|e)(A|B) = ((A|e)A) | ((A|e)B)
flatten r3 = Or (Seq (Or (Lit 'A') (Seq Epsilon (Lit 'A'))) (Lit 'A'))
                (Seq (Or (Lit 'A') (Seq Epsilon (Lit 'A'))) (Lit 'B'))
       -- (A|(eA)) (A|B) = ((A|(eA))A) | ((A|(eA))B)
flatten r4 = r4
       -- the above law can't be applied to (A|(eA)) ((A|B)e)
flatten r5 = Seq (Or (Seq (Or (Lit 'A') (Seq Epsilon (Lit 'A')))
                          Epsilon)
                     (Seq (Or (Lit 'A') (Seq Epsilon (Lit 'A')))
                          (Lit 'B')))
                 (Seq (Or (Lit 'A') (Lit 'B')) Epsilon)
       -- ((A|(eA))(e|B)) ((A|B)e) = (((A|(eA))e) | ((A|(eA))B)) ((A|B)e)
flatten r6 = Or (Seq (Lit 'B') (Seq (Lit 'A') (Lit 'C')))
                (Seq (Lit 'B') (Seq (Lit 'A') (Lit 'D')))
       -- B(A(C|D)) = (B(AC)) | (B(AD))
```

(The correct results are provided in the template file, with names `r1'`, `r2'` etc.) Credit may be given for indicating how you have tested your function.

[*16 marks*]