

Informatics 1 – Functional Programming
Mock Programming Exam, 27 Nov – 1 Dec 2017

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**

1. Suppose the playing cards in a standard deck are represented by characters in the following way: '2' through '9' plus '0' (zero) stand for the number cards, with '0' representing the 10, while the 'A', 'K', 'Q' and 'J' stand for the face cards, i.e. the ace, king, queen and jack, respectively. Let's call these the 'card characters'. The other characters, including the lowercase letters 'a', 'k', 'q', and 'j', and the digit '1', are not used to represent cards.

(a) Write a function `f :: String -> Bool` to test whether all card characters in a string represent face cards. For example:

```
f "ABCDE" = True      f "none here" = True      f "4 Aces" = False
f "01234" = False     f "" = True                f "1 Ace" = True
```

Your function can 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 `g :: String -> Bool` that behaves like `f`, this time using *basic functions* and *recursion*, but not library functions or list comprehension. Credit may be given for indicating how you have tested your function.

[12 marks]

(c) Write a function `h :: String -> Bool` that behaves like `f` using one or more of the following higher-order functions:

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

You can also use *basic functions*, but not other library functions, recursion or list comprehension. Credit may be given for indicating how you have tested your function.

[12 marks]

2. (a) Write a function $\tau :: [a] \rightarrow [a]$ that duplicates every other item in a list. The result should contain the first item once, the second twice, the third once, the fourth twice, and so on. For example,

```
 $\tau$  "abcdefg" = "abbcdeffg"  
 $\tau$  [1,2,3,4] = [1,2,2,3,4,4]  
 $\tau$  ""       = ""
```

Your definition may 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 $u :: [a] \rightarrow [a]$ that behaves like τ , 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 datatype represents propositions.

```
data Proposition = Var String
                | F
                | T
                | Not Proposition
                | Proposition :|: Proposition
                | Proposition :&: Proposition
                deriving (Eq, Ord, Show)
```

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

- (a) Write a function `isNorm :: Proposition -> Bool` that returns true when a proposition is in negation normal form. A proposition is in negation normal form if the only occurrences of logical negation (`Not`) are applied to variables. For example,

```
isNorm (Var "p" :&: Not (Var "q"))      = True
isNorm (Not (Var "p" :|: Var "q"))      = False
isNorm (Not (Not (Var "p"))) :|: Not T  = False
isNorm (Not (Var "p" :&: Not (Var "q"))) = False
```

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

- (b) Write a function `norm :: Proposition -> Proposition` that converts a proposition to an equivalent proposition in negation normal form. A proposition may be converted to normal form by repeated application of the following equivalences:

```
Not F          <-> T
Not T          <-> F
Not (Not p)    <-> p
Not (p :|: q)  <-> Not p :&: Not q
Not (p :&: q)  <-> Not p :|: Not q
```

For example,

```
norm (Var "p" :&: Not (Var "q"))
  = (Var "p" :&: Not (Var "q"))
norm (Not (Var "p" :|: Var "q"))
  = Not (Var "p") :&: Not (Var "q")
norm (Not (Not (Var "p"))) :|: Not T
  = (Var "p" :|: F)
norm (Not (Var "p" :&: Not (Var "q")))
  = Not (Var "p") :|: Var "q"
```

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