

**UNIVERSITY OF EDINBURGH
COLLEGE OF SCIENCE AND ENGINEERING
SCHOOL OF INFORMATICS**

**Date: Tuesday 21st October 2014
Duration: 35 minutes**

**INFORMATICS 1 — FUNCTIONAL PROGRAMMING
CLASS TEST**

INSTRUCTIONS TO CANDIDATES

- **Note that ALL QUESTIONS ARE COMPULSORY.**
- **DIFFERENT QUESTIONS MAY HAVE DIFFERENT NUMBERS OF TOTAL MARKS.** Take note of this in allocating time to questions.
- **WRITE YOUR ANSWERS ON THE EXAM PAPER ITSELF.** Write as legibly as possible.
- In the answer to any part of any question, you may use any function specified in an earlier part of that question. You may do this whether or not you actually provided a definition for the earlier part; nor will you be penalized in a later part if your answer to an earlier part is incorrect.
- Unless otherwise stated, you may use any function from the standard prelude, including the libraries Char, List, and Maybe. You need not write import declarations.
- As an aid to memory, some functions from the standard prelude that you may wish to use are listed on the next page. You need not use all the functions.

**PLEASE INSERT YOUR NAME AND MATRICULATION NUMBER IN
THE SPACE BELOW:**

MATRICULATION NUMBER	NAME

```

div, mod :: Integral a => a -> a -> a
even, odd :: Integral a => a -> Bool
(+), (*), (-), (/) :: Num a => a -> a -> a
(<), (<=), (>), (>=) :: Ord a => a -> a -> Bool
(==), (/=) :: Eq a => a -> a -> Bool
(&&), (||) :: Bool -> Bool -> Bool
not :: Bool -> Bool
max, min :: Ord a => a -> a -> a
isAlpha, isAlphaNum, isLower, isUpper, isDigit :: Char -> Bool
toLower, toUpper :: Char -> Char
ord :: Char -> Int
chr :: Int -> Char

```

Figure 1: Basic functions

sum, product :: (Num a) => [a] -> a	and, or :: [Bool] -> Bool
sum [1.0,2.0,3.0] = 6.0	and [True,False,True] = False
product [1,2,3,4] = 24	or [True,False,True] = True
maximum, minimum :: (Ord a) => [a] -> a	reverse :: [a] -> [a]
maximum [3,1,4,2] = 4	reverse "goodbye" = "eybdoog"
minimum [3,1,4,2] = 1	
concat :: [[a]] -> [a]	(++) :: [a] -> [a] -> [a]
concat ["go","od","bye"] = "goodbye"	"good" ++ "bye" = "goodbye"
(!!) :: [a] -> Int -> a	length :: [a] -> Int
[9,7,5] !! 1 = 7	length [9,7,5] = 3
head :: [a] -> a	tail :: [a] -> [a]
head "goodbye" = 'g'	tail "goodbye" = "oodbye"
init :: [a] -> [a]	last :: [a] -> a
init "goodbye" = "goodby"	last "goodbye" = 'e'
takeWhile :: (a->Bool) -> [a] -> [a]	take :: Int -> [a] -> [a]
takeWhile isLower "goodBye" = "good"	take 4 "goodbye" = "good"
dropWhile :: (a->Bool) -> [a] -> [a]	drop :: Int -> [a] -> [a]
dropWhile isLower "goodBye" = "Bye"	drop 4 "goodbye" = "bye"
elem :: (Eq a) => a -> [a] -> Bool	replicate :: Int -> a -> [a]
elem 'd' "goodbye" = True	replicate 5 '*' = "*****"
zip :: [a] -> [b] -> [(a,b)]	
zip [1,2,3,4] [1,4,9] = [(1,1),(2,4),(3,9)]	

Figure 2: Library functions

1. (a) In typography, a *descender* is the portion of a character that extends below the line, for example the “tail” in the letter y. Write a function `f :: Char -> Bool` that returns `True` for letters with descenders and `False` otherwise. For example:

```
f 'a' == False      f 'p' == True       f 'A' == False
f 'P' == False      f '3' == False
```

Which letters have descenders is dependent on font, but for this question assume that no upper-case letters have descenders and that only the following lower-case letters have descenders: g, j, p, q, y.

[15 marks]

- (b) Using `f`, define a function `g :: String -> Int` that given a string returns the number of letters in the string that have descenders. For example:

```
g "prig" == 2        g "minimum" == 0        g "" == 0
g "42NATly" == 1     g "Jiggle" == 2
```

Your definition may use basic functions, *list comprehension*, and *library functions*, but not recursion.

[20 marks]

- (c) Again using `f`, define a function, `h :: String -> Int` that behaves identically to `g`, this time using basic functions and *recursion*, but not list comprehension or library functions.

[20 marks]

2. (a) Write a function `c :: String -> String` that converts all characters in positions 0, 2, 4, ... to upper case, numbering from 0. For example:

```
c "haskell" == "HaSkElL"      c "" == ""  
c "Edinburgh" == "EdInBuRgH"  c "83wing" == "83WiNg"
```

Your definition may use *basic functions*, *list comprehension*, and *library functions*, but not recursion.

[20 marks]

- (b) Define a second function `d :: String -> String` that behaves identically to `c`, this time using *basic functions* and *recursion*, but not list comprehension or other library functions.

[20 marks]

- (c) Write a QuickCheck property `prop_cd` to confirm that `c` and `d` behave identically. Give the type signature of `prop_cd` and its definition.

[5 marks]