UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

Date: Thursday 3rd November 2005
Duration: 35 minutes

INFORMATICS 1A
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 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.
- As an aid to memory, some functions from the standard prelude that you may wish to use are listed on the next page.

PLEASE INSERT YOUR MATRICULATION NUMBER IN THE SPACE BELOW:

MATRICULATION NUMBER

```
div, mod :: Int -> Int -> Int               (&&), (||) :: Bool -> Bool -> Bool
13 ‘div‘ 4 == 3                             True && False == False
13 ‘mod‘ 4 == 1                             True || False == True

intToDigit :: Int -> Char                   digitToInt :: Char -> Int
intToDigit 3 = ’3’                          digitToInt ’3’ = 3

fromIntegral :: Int -> Float
fromIntegral 3 == 3.0

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

(:) :: a -> [a] -> [a]                      (++) :: [a] -> [a] -> [a]
’g’ : "oodbye" == "goodbye"                 "good" ++ "bye" == "goodbye"

head :: [a] -> a                            tail :: [a] -> [a]
head "goodbye" == ’g’                       tail "goodbye" == "oodbye"

take :: Int -> [a] -> [a]                   drop :: Int -> [a] -> [a]
take 4 "goodbye" == "good"                  drop 4 "goodbye" == "bye"

elem :: (Eq a) => a -> [a] -> Bool
elem ’d’ "goodbye" == True

replicate :: Int -> a -> [a]
replicate 5 ’*’ == "*****"

concat :: [[a]] -> [a]
concat ["con","cat","en","ate"] == "concatenate"

zip :: [a] -> [b] -> [(a,b)]
zip [1,2,3,4] [1,4,9] == [(1,1),(2,4),(3,9)]
```

1. Floating point numbers can be used to represent amounts of money in pounds, and discounts as a percentage. Quantities are represented by integers.

```
type Pounds = Float
type Percent = Float
type Quantity = Int
```

(a) Write a function `discount :: Quantity -> Percent` to implement the following rule for discounting orders according to quantity.

- If the quantity is between 1 and 9, there is no discount.
- If the quantity is between 10 and 40, there is a discount of 10%.
- If the quantity is 50 or greater, there is a discount of 20%.

The function may raise an error if the quantity is negative. For example, `discount 5` returns `0.0`, and `discount 20` returns `0.1` and `discount 100` returns `0.2`. The function `fromIntegral :: Int -> Float` can be used to convert an integer to a float (you must convert an integer to a float before using it to multiply another float).

[*6 marks*]

(b) Write a function `total :: [(Quantity,Pounds)] -> Pounds` that given a list of quantities and costs, returns the total cost of all orders, after discounting. Any order with a negative quantity should be excluded. For example,

```
total [(5, 4.00), (-1, 1000.00), (20, 10.00), (100, 2.00)]
```

returns $5 \times 1.0 \times 3.00 + 20 \times 0.9 \times 10.00 + 100 \times 0.8 \times 2.00 = 20.00 + 180.00 + 160.00 = 360.00$. Your definition should use a *list comprehension* and library functions, not recursion.

[*6 marks*]

(c) Write a second definition of `total`, this time using *recursion*, and not a list comprehension or library functions (other than arithmetic).

[*6 marks*]

2. (a) Write a function `changes :: [Int] -> [Int]` that returns all numbers in a list that differ from the following number. For example, `changes [1,2,2,1,1,3,3,3]` and `changes [1,2,1,3]` both return `[1,2,1]`, and `changes [3,3,3]` returns `[]`. Your definition should use a *list comprehension* and library functions, not recursion.

[*6 marks*]

(b) Write a second definition of `changes`, this time using *recursion*, and not a list comprehension or library functions (other than arithmetic).

[*6 marks*]

(c) Using `changes`, write a function `alleq :: [Int] -> Bool` that returns true if all numbers in the input list are equal. For example, `alleq [1,2,2,3,3,3]` returns false and `alleq [3,3,3]` returns true.

[*5 marks*]