UNIVERSITY OF EDINBURGH
COLLEGE OF SCIENCE AND ENGINEERING
SCHOOL OF INFORMATICS

Date: 1–5 November 2004

INFORMATICS 1A

Examiners R. Irving (External)

M. Jerrum (Chair)

INSTRUCTIONS TO CANDIDATES

1. Candidates in the third or later years for the degree of M.A. (Arts), B. Comm, B.Sc (Social Sciences) or LL.B should put a cross (X) in the box on the front cover of **both** script books.
2. Answer Part A and Part B in SEPARATE SCRIPT BOOKS. Mark the question number clearly in the space provided on the front of the book.
3. **Note that ALL QUESTIONS ARE COMPULSORY.**
4. **DIFFERENT QUESTIONS MAY HAVE DIFFERENT NUMBERS OF TOTAL MARKS.** Take note of this in allocating time to questions.

Write as legibly as possible.

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

Part A COMPUTATION AND LOGIC

This sample exam does not cover Computation and Logic.

Part B FUNCTIONAL PROGRAMMING

Unless otherwise stated, you may use either list comprehension or recursion in your answers (or both).

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 operators and functions from the standard prelude and from the libraries Char, List, and Maybe. You need not write import declarations.

As an aid to memory, types of relevant functions are listed on the next page. You will not need all of the functions listed.

```
data Maybe a                  =  Nothing | Just a
                              deriving (Eq,Ord,Show)

(<), (<=), (>=), (>)          :: (Ord a) => a -> a -> Bool
(==), (/=)                    :: (Eq a) => a -> a -> Bool
min, max                      :: (Ord a) => a -> a -> a
(+), (-), (*), (/)            :: (Num a) => a -> a -> a
div, mod                      :: (Integral a) => a -> a -> a
(&&), (||)                    :: Bool -> Bool -> Bool
(++)                          :: [a] -> [a] -> [a]

otherwise                     :: Bool

sum                           :: (Num a) => [a] -> a
minimum, maximum              :: (Ord a) => [a] -> a
show                          :: (Show a) => a -> String

elem                          :: (Eq a) => a -> [a] -> Bool
lookup                        :: (Eq a) => a -> [(a,b)] -> Maybe b
fromMaybe                     :: a -> Maybe a -> a

isAlpha, isUpper, isLower     :: Char -> Bool
toUpper, toLower              :: Char -> Char

words, lines                  :: String -> [String]
unwords, unlines              :: [String] -> String

head                          :: [a] -> a
tail                          :: [a] -> [a]
zip                           :: [a] -> [b] -> [(a,b)]
take, drop                    :: Int -> a -> [a]

map                           :: (a -> b) -> [a] -> [b]
zipWith                       :: (a -> b -> c) -> [a] -> [b] -> [c]
filter, takeWhile, dropWhile  :: (a -> Bool) -> [a] -> [a]
iterate                       :: (a -> a) -> a -> [a]

foldr                         :: (a -> b -> b) -> b -> [a] -> b
foldl                         :: (b -> a -> b) -> b -> [a] -> b
foldr1, foldl1                :: (a -> a -> a) -> a -> [a] -> a

sort                          :: (Ord a) => [a] -> [a]
nub                           :: (Eq a) => [a] -> [a]
group                         :: (Eq a) => [a] -> [[a]]
```

1. Inventory for a warehouse consists of a list of entries, where each entry contains an item name, a quantity, and a price (in pence).

```
type Item = String
type Quantity = Int
type Price = Int
type Entry = (Item, Quantity, Price)
```

   (a) Write a function `important :: Entry -> Bool` that returns true if the stock of that item is considered important. An item is *important* if one of the following holds:

   - There are more than 100 items is stock,
   - the price of a single item is more than 1,000 pence,
   - the value is greater than 50,000, where the value is the product of the quantity and the price.

   [5%]

   (b) Write a function `showEntry :: Entry -> String` that converts an entry to a string. The name of the item should be followed by a star if it is important. Thus,

```
Main*> showEntry ("Paper clip", 5000, 10)
"Paper clip(*): 5000@0.10"
Main*> showEntry ("Notebook", 3, 900)
"Notebook: 3@9.00"
```

   [5%]

   (c) Write a function `showEntries :: [Entry] -> String` that prints out a list of entries. For example,

```
Main*> let entries = [("Paper clip",5000,10),("Notebook",3,900)]
Main*> putStr (showEntries entries)
Paper clip(*): 5000@0.10
Notebook: 3@9.00
```

   [5%]

2. Floating point numbers can be used to represent amounts of money in pounds.

```
type Pounds = Float
```

(a) Write a function `tax :: Pounds -> Pounds` that given an income returns the corresponding amount of tax. It should signal an error if given an income less than zero. The formula for computing tax in 2004–5 is as follows.

- Income in the range £0–£2,010 is charged at the *starting* rate of 10%.
- Income in the range £2,010–£31,400 is charged at the *basic* rate of 22%.
- Income above £31,400 is charged at the *higher* rate of 40%.

Thus, an income of £20,000 attracts a tax of $£4,158.80 = 0.10 \times 2010 + 0.20 \times (20,000 - 2,010)$, and an income of £40,000 attracts a tax of $£10,106.80 = 2,010 \times 0.10 + (31,400 - 2,010) \times 0.22 + (40,000 - 31,400) \times 0.40$.

[5%]

(b) Write a function `highTax :: [Pounds] -> Pounds` that given a list of incomes, returns the total tax to be paid by all individuals who pay tax at the higher rate. For example, given the list [40000, 20000, 40000] this function will return $£20213.6 = 10106.8 + 10106.8$. Your definition should use a *list comprehension* and library functions, not recursion.

[5%]

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

[5%]

3. Pascal's triangle looks like this.

```
     1
    1 1
   1 2 1
  1 3 3 1
 1 4 6 4 1
1 5 10 10 5 1
```

The number in each row is the sum of the two numbers above it.

(a) Write a function `nextRow :: [Int] -> [Int]` that given a list of integers, forms a new list containing the sum of each adjacent pair of integers in the original, treating the first number as if it were preceded by 0 and the last as if it were followed by zero. Thus, `nextRow [1,3,3,1] = [1,4,6,4,1]` since $1 = 0+1, 4 = 1+3, 6 = 3+3, 4 = 3+1, 1 = 1+0$.

[5%]

(b) Write a function `pascal :: Int -> [[Int]]` that generates the given number of rows of Pascal's triangle. Thus,

```
*Main> pascal 6
[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1],[1,5,10,10,5,1]]
```

[5%]

(c) Write a function `showLine :: Int -> [Int] -> String` that will display a given row of Pascal's triangle, centred to occupy a line of the given width. For example,

```
*Main> showLine 20 [1,4,6,4,1]
"     1 4 6 4 1"
```

Each number should be separated by one space. The numbers occupy 9 positions, and there are 5 spaces at the beginning of the line, leaving 6 at the end to occupy a total width of 20.

[5%]

(d) Write a function `showPascal :: Int -> Int -> String` that will display the first `n` rows of Pascal's triangle, centred to occupy a line of the given width. For example,

```
*Main> putStr (showPascal 20 6)
         1
        1 1
       1 2 1
      1 3 3 1
     1 4 6 4 1
   1 5 10 10 5 1
```

[5%]