UNIVERSITY OF DUBLIN
TRINITY COLLEGE

Faculty of Engineering, Mathematics and Science

School of Computer Science & Statistics

Integrated Computer Science, Year 3 **Trinity Term 2013**
B.A. (Mod.) Computer Science & Business
Junior Sophister Annual Examination

Introduction to Functional Programming

Wednesday, 8th May **Luce Hall Lower** **09:30–11:30**

Dr Andrew Butterfield

Instructions to Candidates:

Attempt **three** questions. All questions carry equal marks. Each question is scored out of a total of 33 marks.

There is a reference section at the end of the paper (pp6–7).

You may not start this examination until you are instructed to do so by the Invigilator.

Materials permitted for this examination:

None

1. The Haskell Prelude defines a large number of list functions that are loaded by default when a Haskell program is interpreted or compiled.

   Give a complete implementation of the Prelude functions described below. By "complete" is meant that any other functions used to help implement those below must also have their implementations given.

   (a) Return `True` if the list is empty, `False` otherwise.

   ```
   null              :: [a] -> Bool
   ```

   [4 marks]

   (b) Concatenate two lists together.

   ```
   (++) :: [a] -> [a] -> [a]
   ```

   [5 marks]

   (c) Return the last element of a non-empty list.

   ```
   last              :: [a] -> a
   ```

   [5 marks]

   (d) Call `dropwhile p xs` scans the elements of `xs` until `p` returns `False`. It then returns the list from that point onwards.

   ```
   dropWhile                 :: (a -> Bool) -> [a] -> [a]
   ```

   [6 marks]

   (e) Call filter p xs returns a list containing only those elements of `xs` that return `True` when `p` is applied to them.

   ```
   filter :: (a -> Bool) -> [a] -> [a]
   ```

   [6 marks]

   (f) Take a binary function and a non-empty list of elements and use the function to reduce the list down to one value with nesting to the right.

   ```
   foldr1 op [x1,x2,...,xn-1,xn]
    = x1 `op` (x2 `op` ... (xn-1 `op` xn)...)

   foldr1            :: (a -> a -> a) -> [a] -> a
   ```

   [7 marks]

2. Consider the following function definitions:

```
f1 [] = 1
f1 (x:xs) = x * f1 xs
f2 [] = 0
f2 (x:xs) = 1 + f2 xs
f3 [] = 0
f3 (x:xs) = x + f3 xs
f4 [] = []
f4 (x:xs) = x ++ f4 xs
f5 [] = 0
f5 (x:xs) = (x*x) + f5 xs
```

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called? [5 marks]

3. We have an expression datatype as follows:

```
data Expr = K Int
          | V String
          | Add Expr Expr
          | Dvd Expr Expr
          | Let String Expr Expr
```

a dictionary type with insert and lookup functions (full code not given):

```
type Dict = [(String,Int)]
ins :: String -> Int -> Dict -> Dict
lkp :: String -> Dict -> Maybe Int
```

and one function eval defined over expressions:

```
eval :: Dict -> Expr -> Int
eval _ (K i) = i
eval d (V s) = fromJust $ lkp s d
eval d (Add e1 e2) = eval d e1 + eval d e2
eval d (Dvd e1 e2) = eval d e1 `div` eval d e2
eval d (Let v e1 e2)
 = eval (ins v i d) e2
 where  i = eval d e1

fromJust (Just x) = x
```

(a) Explain the ways in which function eval can fail, with Haskell runtime errors. [5 marks]

(b) Add in error handling for function eval above, using the Maybe type, to ensure this function is now total. Note that this will require changing the type of this function. [14 marks]

(c) Add in error handling for the eval function above, using the Either type, ensuring it is now total, and giving back a useful error message. Note that this will also require changing the type (again) of the function. [14 marks]

4. (a) Consider the following function definition:

```
sumsq [] = 0
sumsq (x:xs) = x * x + sumsq xs
```

Use the shorthand AST notation to show how the application `sumsq [2,3]` is evaluated, indicating clearly where copying takes place. You need not draw the full AST (with cons-nodes) for the lists but just show any list instead as a single node, `[]`, `[6]`, etc, as appropriate. [10 marks]

(b) Consider the following function definitions:

```
down n = n : down (n-1)
take 0 xs = []
take n (x:xs) = x : take (n-1) xs
```

Show the evaluation of `take 2 (down 42)` using both *Strict* Evaluation and *Lazy* Evaluation. Show enough evaluation steps to either indicate the final result, or to illustrate why no such result will emerge. [8 marks]

(c) Using explicit numbers, lists and either or both functions `take` and `down` above, *and no others*, write expressions, *if possible*, that:

i. terminate when evaluated both strictly and lazily

ii. terminate when evaluated strictly but not when evaluated lazily

iii. terminate when evaluated lazily but not when evaluated strictly

iv. fail to terminate when evaluated either strictly or lazily

[4 marks]

(d) Write a program that prompts the user for a filename of the form ⟨*root*⟩`.in` (where ⟨*root*⟩ is the filename less its extension) opens that file, reads its contents, maps all its characters to uppercase, and outputs the result to file ⟨*root*⟩`.out` (See the Reference, p7). [11 marks]

Reference

Prelude List Functions

```
map :: (a -> b) -> [a] -> [b]
(++) :: [a] -> [a] -> [a]
filter :: (a -> Bool) -> [a] -> [a]
concat :: [[a]] -> [a]
head             :: [a] -> a
tail             :: [a] -> [a]
last             :: [a] -> a
init             :: [a] -> [a]
null             :: [a] -> Bool
length           :: [a] -> Int
(!!)                :: [a] -> Int -> a
foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl1           :: (a -> a -> a) -> [a] -> a
scanl            :: (a -> b -> a) -> a -> [b] -> [a]
scanl1           :: (a -> a -> a) -> [a] -> [a]
foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr1           :: (a -> a -> a) -> [a] -> a
scanr             :: (a -> b -> b) -> b -> [a] -> [b]
scanr1          :: (a -> a -> a) -> [a] -> [a]
iterate          :: (a -> a) -> a -> [a]
repeat           :: a -> [a]
replicate        :: Int -> a -> [a]
cycle            :: [a] -> [a]
take                  :: Int -> [a] -> [a]
drop                  :: Int -> [a] -> [a]
splitAt                 :: Int -> [a] -> ([a],[a])
takeWhile              :: (a -> Bool) -> [a] -> [a]
dropWhile              :: (a -> Bool) -> [a] -> [a]
span, break            :: (a -> Bool) -> [a] -> ([a],[a])
```

Prelude IO Functions

```
type  FilePath = String

putChar     :: Char -> IO ()
putStr      :: String -> IO ()
putStrLn    :: String -> IO ()
print       :: Show a => a -> IO ()
getChar     :: IO Char
getLine     :: IO String
getContents :: IO String
readFile    :: FilePath -> IO String
writeFile   :: FilePath -> String -> IO ()
```

Data.Char Functions

```
isControl :: Char -> Bool
isSpace :: Char -> Bool
isLower :: Char -> Bool
isUpper :: Char -> Bool
isAlpha :: Char -> Bool
isAlphaNum :: Char -> Bool
isPrint :: Char -> Bool
isDigit :: Char -> Bool
toUpper :: Char -> Char
toLower :: Char -> Char
toTitle :: Char -> Char
digitToInt :: Char -> Int
intToDigit :: Int -> Char
ord :: Char -> Int
chr :: Int -> Char
```