

UNIVERSITY OF DUBLIN TRINITY COLLEGE

Faculty of Engineering, Mathematics and Science

School of Computer Science & Statistics

**Integrated Computer Science
B.A. (Mod.) Computer Science & Business
M.S.I.S.S.**

Trinity Term 2014

Introduction to Functional Programming

Monday, 19th May

Goldsmith Hall

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 (See Reference p6) 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) Returns the list with its first element removed, if it is non-empty, with a runtime error otherwise.

```
tail          :: [a] -> [a]
```

[4 marks]

- (b) Concatenate two lists together.

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

[5 marks]

- (c) returns everything but the last element of a list, if it is non-empty, with a runtime error otherwise.

```
init          :: [a] -> [a]
```

[5 marks]

- (d) Uses a predicate to split a list into two, the first list being the longest prefix that *does not satisfy* the predicate, while the second list is what remains

```
break         :: (a -> Bool) -> [a] -> ([a], [a])
```

[6 marks]

- (e) Reverse its list argument

```
reverse       :: [a] -> [a]
```

[6 marks]

- (f) Compute the maximum of a non-empty list

```
maximum       :: Ord a => [a] -> a
```

[7 marks]

2. Consider the following function definitions:

```
f1 p [] = p
f1 p (x:xs) = f1 (p*x) xs
```

```
f2 ell [] = ell
f2 ell (x:xs) = f2 (ell+1) xs
```

```
f3 s [] = s
f3 s (x:xs) = f3 (s+x) xs
```

```
f4 c [] = c
f4 c (x:xs) = f4 (c++x) xs
```

```
f5 q [] = q
f5 q (x:xs) = f5 (q+x*x) 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 a binary tree built from number-string pairs, ordered by the number (acting as key),

```
data Tree = Empty
          | Single Int String
          | Many Tree Int String Tree
```

and one function search defined over it:

```
search :: Tree -> Int -> String
```

```
search x (Many left i s right)
  | x == i = s
  | x > i = search x right
search x (Single i s)
  | x == i = s
```

- (a) Explain the ways in which function search can fail, with Haskell runtime errors.
[5 marks]
- (b) Add in error handling for function search above, using the Maybe type, to ensure this function is now total. Note that this will require changing the type of the search function. [12 marks]
- (c) Add in error handling for the search 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 search function. [16 marks]

4. (a) Consider the following function definition:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

Use the shorthand Abstract Syntax Tree (AST) notation to show how the application `sum [3,39]` 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:

```
evenup n = n : evenup (n+2)
take 0 xs = []
take n (x:xs) = x : take (n-1) xs
```

Show the evaluation of `take 2 (evenup 2)` 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 `evenup` above, and no others, write expressions, if possible, that:

- i. fail to terminate when evaluated either strictly or lazily
- ii. terminate when evaluated strictly but not when evaluated lazily
- iii. terminate when evaluated lazily but not when evaluated strictly
- iv. terminate when evaluated both strictly and lazily

If such an expression is not possible for any of the above cases, explain why.

[4 marks]

- (d) Write a program that prompts the user for a filename of the form `<Root>.<Extension>`, changes the filename to DOS 8+3 format (All uppercase, root and extension with max length of 8 and 3 respectively), opens that file, reads its contents, maps all its characters to lowercase, and outputs the result to file `<DOSROOT>.OUT` (See the Prelude IO Functions in 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
```