Coláiste na Tríonóide, Baile Átha Cliath
Trinity College Dublin
Ollscoil Átha Cliath | The University of Dublin

**Faculty of Engineering, Mathematics and Science**

**School of Computer Science & Statistics**

**Integrated Computer Science** **Trinity Term 2016**
**B.A. (Mod.) Computer Science & Business**
**M.S.I.S.S.**
**Year 3 Annual Examinations**

# Introduction to Functional Programming

**Wednesday, 25th May** **Goldsmith Hall** **14:00–16:00**

**Dr Andrew Butterfield**

**Instructions to Candidates:**

Attempt **three** questions. All questions carry equal marks. Each question is scored out of a total of 100 marks.

There is a reference section at the end of the paper (pp7–8).

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

1. 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) `head :: [a] -> a`

   Returns the first element of a list, if it is non-empty, with a runtime error otherwise. [12 marks]

   (b) `init :: [a] -> [a]`

   Returns everything but the last element of a list, if it is non-empty, with a runtime error otherwise. [15 marks]

   (c) `last :: [a] -> a`

   Returns the last element of a list, if it is non-empty, with a runtime error otherwise. [15 marks]

   (d) `span :: (a -> Bool) -> [a] -> ([a],[a])`

   Uses a predicate to split a list into two, the first list being the longest prefix that *satisfies* the predicate, while the second list is what remains [18 marks]

   (e) `(!!) :: [a] -> Int -> a`

   We can index into a list, starting from zero. So `xs!!n` returns the $(n+1)$th element of `xs`, provided `n` is non-negative and the length of the list is long enough. Otherwise, we get a run-time error. [18 marks]

   (f) `foldl1 :: (a -> a -> a) -> [a] -> a`

   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 left, as illustrated immediately below

   ```
   foldl1 op [x1,x2,...,xn-1,xn]
    = ((...((x1 `op` x2) `op` x3) ... ) `op` xn-1) `op` xn
   ```

   [22 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

[42 marks]

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

[42 marks]

(c) 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
```

Explain the ways in which function `search` can fail with Haskell *runtime* errors.

[16 marks]

3. (a) We have an expression datatype as follows:

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

and a dictionary type with insert (`ins`) and lookup (`lkp`) 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
```

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. [54 marks]

(b) Consider the following function definition:

```
prod [] = 1
prod (0:_) = 0
prod (x:xs) = x * prod xs
```

Use the shorthand AST notation to show how the application
`prod [3,14,0,999]`
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, `[]`, `[999]`, etc, as appropriate. [46 marks]

4. Given the following definitions:

```
prod [] = 1
prod (n:ns) = n * prod ns
[] ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)
mbr x [] = False
mbr x (y:ys) | x == y = True
             | otherwise  = mbr x ys
rem x [] = []
rem x (y:ys) | x == y  = rem x ys
             | otherwise  =  y:(rem x ys)
map f [] = []
map f (x:xs) = (f x) : map f xs
from n = n : from (n+1)
```

(a) Prove the following property

```
prod(ms++ns) == prod ms * prod ns
```

[33 marks]

(b) Consider the following property:

```
mbr x (rem x ys) == False
```

State the base and step proofs to be done in a proof by induction, propose a case split for the step property, and prove one of the cases. [33 marks]

(c) Prove the following property by co-induction.

```
map (+1) (from 0) == from 1
```

[34 marks]

5. Consider the following recursively defined datatype:

```
data Shape = Dot Size Coord
           | Box Size Size Coord
           | SideBySide Shape Shape
           | Stack [Shape]
```

where we assume that Size and Coord are types defined elsewhere.

(a) Show how the definition of Shape can be written as an algebraic expression. [18 marks]

(b) Compute the derivative, w.r.t. Shape, of the algebraic expression given as an answer in (a) above (See Reference at p8). [18 marks]

(c) Define a Haskell "derivative" type that corresponds to the derivative computed in (b) above. [24 marks]

(d) Define a "Zipper" type called ShapeZip for Shape. [12 marks]

(e) Give Haskell code for a zipper function with the following signature that performs a move upwards

```
up :: ShapeZip -> ShapeZip
```

[28 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])
```

**Laws of Differentiation**

$$\frac{dk}{dx} = 0$$

$$\frac{dx^n}{dx} = nx^{n-1}$$

$$\frac{d(f(x)+g(x))}{dx} = \frac{df(x)}{dx} + \frac{dg(x)}{dx}$$

$$\frac{d(f(x)g(x))}{dx} = f(x)\frac{dg(x)}{dx} + g(x)\frac{df(x)}{dx}$$

$$\frac{d(t^*)}{dt} = (t^*)^2$$