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 2017**
B.A. (Mod.) Computer Science & Business
M.S.I.S.S.
Year 3 Annual Examination

Introduction to Functional Programming

Friday, 12th May **RDS Main Hall** **09:30–11:30**

Dr Andrew Butterfield

Instructions to Candidates:

Attempt **three** questions. All questions are scored out of a total of 33 marks.

There is a reference section at the end of the paper (pp6–6).

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

   Returns all but the first element of a list, if it is non-empty, with a runtime error otherwise. [4 marks]

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

   Returns the last element of a list, if it is non-empty, with a runtime error otherwise. [5 marks]

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

   Returns everything but the last element of a list, if it is non-empty, with a runtime error otherwise. [5 marks]

   (d) `splitAt :: Int -> [a] -> ([a],[a])`

   Split a list (its second argument) into two, the first list being the prefix whose length equals the first argument (or the whole list if the list is shorter) while the second list is what remains, if anything. [6 marks]

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

   This function call `replicate n x` returns a list that is `n` copies of `x`. [6 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
```

   [7 marks]

2. Consider the following function definitions:

```
f1 [] = 42
f1 (x:xs) = x * f1 xs
f2 [] = 0
f2 (x:xs) = 99 * f2 xs
f3 [] = 0
f3 (x:xs) = x + f3 xs
f4 [] = []
f4 (x:xs) = x ++ f4 xs
f5 [] = 0
f5 (x:xs) = (x-42) + f5 xs
```

```
hof :: [a] -> (a -> a) -> (a -> a -> a) -> a -> a
hof [] _ _ base = base
hof (x:xs) t f base = (t x) `f` (hof xs t f base)

f1 xs = hof xs (\x -> x) (+) 42
f2 xs = hof xs (\x -> 99) (*) 0
f3 xs = hof xs (\x -> x) (+) 0
f4 xs = hof xs (\x -> x)  (++) []
f5 xs = hof xs (\x -> x-42) (+) 0
```

\ means where

They all have a common pattern of behaviour.

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

[14 marks]

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

[14 marks]

(c) We have a binary tree built from number-string pairs, ordered by the number (acting as key),

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

and one function `search` defined over it:

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

search x Empty = undefined
search x (Single i s)
 | x == i  =  s
search x (Many left i s right)
 | x  < i  =  search x left
```

There is no pattern for a Many tree with x>i, so if you are searching for a node that isn't a leaf, there will be a runtime error if the node's integer is lesser than the integer you are searching for. This means the node's right branches are never traversed.

The only pattern for searching for a leaf node is if the the integer of the leaf node is equal to the integer you are looking for. If you search for any other value it will result in a runtime error.

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

[5 marks]

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

```
data Expr = K Int
          | V String
          | Add Expr Expr
          | Dvd Expr Expr
          | Where Expr String 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 :: Dict -> String -> Maybe Int
```

and one function `eval` defined over expressions:

```
eval :: Dict -> Expr -> Int
eval _ (K i) = i
eval d (V s) = fromJust $ lkp d s
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 (Where e1 v e2) = eval (ins (v, i) d) e1
                       where  i = eval d e2
fromJust (Just x) = x
```

```
eval :: Dict -> Expr -> Maybe Int
eval _ _ = Nothing
eval _ (K i) = Just i
eval d (V s) = fromJust $ lkp s
eval d (Add e1 e2) = eval d e1 'div' eval d e2
eval d (Dvd e1 e2) = eval d e1 'div' eval d e2
eval d (Where e1 v e2) = eval (ins (v, i) d) e1
                       where i = eval d e2
```

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, and your answer should provide the revised type-signature. [18 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 [42,999,0,123,4,35,663]
```

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, `[]`, `[663]`, etc, as appropriate. [15 marks]

4. Given the following definitions:

```
sum [] = 0
sum (n:ns) = n + sum ns

[] ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)

mbr x [] = False
mbr x (y:ys) | x == y = True
             | otherwise  = mbr x ys
ins x ys = x : ys

map f [] = []
map f (x:xs) = (f x) : map f xs

from n = n : from (n+1)
```

(a) Prove the following property

```
sum (ms++ns) == sum ms + sum ns
```

[11 marks]

(b) Consider the following property:

```
mbr x (ins x ys) == True
```

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. [11 marks]

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

```
map (-42) (from 0) == from -42
```

[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])
```