TRINITY COLLEGE DUBLIN
THE UNIVERSITY OF DUBLIN

Faculty of Engineering, Mathematics and Science

School of Computer Science & Statistics

Integrated Computer Science Programme **Trinity Term 2015**
B.A. (Mod.) MSISS
B.A. (Mod.) Computer Science & Business
Junior Sophister Annual Examination

Introduction to Functional Programming

Thursday 14th May 2015 **Goldsmith Hall** **09:30–11:30**

Mr Andrew Anderson

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 (pp6–8).

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 module `Prelude` is imported by default in every Haskell program. The Prelude defines a large number of useful list functions.

   Give a complete implementation of the Prelude functions described below, **including** the implementations of any functions used as helpers, unless stated otherwise.

   (a) `repeat` generates an infinite list of copies of `x`. For example, `repeat 5` generates the list `[5,5,5,5,5...]`.

   ```
   repeat            :: a -> [a]
   ```

   [4 marks]

   (b) `replicate` generates a finite list of exactly `n` copies of `x`. For example, `replicate 3 'A'` generates the list `['A', 'A', 'A']`.

   ```
   replicate         :: Int -> a -> [a]
   ```

   [5 marks]

   (c) `concat` joins a list of lists together into a single list. For example, `concat [[1, 2], [3]]` creates the list `[1, 2, 3]`.

   ```
   concat            :: [[a]] -> [a]
   ```

   [5 marks]

   (d) `zip` creates a new list by joining elements at the same index in two input lists into pairs. `zip [1,2] ['a','b']` creates the list `[(1, 'a'), (2, 'b')]`.

zip [] [] = []
zip (x:xs) (y:ys) = (x,y) : (zip xs ys)

   ```
   zip               :: [a] -> [b] -> [(a,b)]
   ```

   [6 marks]

   (e) `unzip` converts a list of pairs into a pair of lists by sending the first element of each input pair to one list, and the second element to another. For example, `unzip [(1,4), (2,5), (3,6)]` creates the pair `([1,2,3], [4,5,6])`. Unzipping the empty list should result in a pair of empty lists.

   ```
   unzip             :: [(a,b)] -> ([a], [b])
   ```

   [6 marks]

   (f) `minimum` finds the least value in a **non-empty** list. The class constraint `(Ord a)` ensures that the operations of the typeclass `Ord` work on items in the list. The operations of the typeclass `Ord` are listed in the reference — you need not implement them.

   ```
   minimum           :: (Ord a) => [a] -> a
   ```

   [7 marks]

2. Consider the following function definitions:

```
f1 [] _ = []
f1 _ [] = []
f1 (x:xs) (y:ys) = (x * y) : f1 xs ys

f2 [] _ = []
f2 _ [] = []
f2 (x:xs) (y:ys) = (x + y) : f2 xs ys

f3 [] _ = []
f3 _ [] = []
f3 (x:xs) (y:ys) = (x y) : f3 xs ys

f4 [] _ = []
f4 _ [] = []
f4 (x:xs) (y:ys) = (y, x) : f4 xs ys

f5 [] _ = []
f5 _ [] = []
f5 (x:xs) (y:ys) = x : f5 xs ys
```

```
hof :: [a] -> [b] -> (a -> b -> c) -> [c]
hof [] _ _ = []
hof _ [] _ = []
hof (x:xs) (y:ys)  f = (f x y) (hof xs ys f)

f1 xs ys = hof xs ys (\x y -> x*y)
f2 xs ys = hof xs ys (\x y -> x+y)
f3 xs ys = hof xs ys (\x y -> (x,y))
f4 xs ys = hof xs ys (\x y -> (y,x))
f5 xs ys = hof xs ys (\x y -> x)
```

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour **without** using any function from the reference.

[6 marks]

(b) Write the type signature that Haskell will infer for `hof`.

hof :: (a -> b -> c) -> [a] -> [b] -> [c]

[2 marks]

(c) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments. You **may** use any Prelude function from the reference as a helper.

[20 marks]

(d) Is `hof` provided by the Haskell Prelude (under another name)? If so, what is it called?

[5 marks]

yes, it's provided under the name zipWith

3. A **hash table** is a very common data structure which maps keys to values (a dictionary). To speed up the search for the value associated with some key, a hash table organises (key, value) pairs into *buckets*. In each bucket, all the keys have the same value under some **hash function**. To look up the value associated with some key, we first compute the *hash* of the key, then we search *only* the bucket associated with that hash, and not the full table. An example hash function `hash` is given below.

```
type Bucket k v = [(k, v)]
type HashTable k v = [(Int, Bucket k v)]

hash :: String -> Int
hash str = (sum (map ord str)) `mod` 255
```

Given a key, to extract the corresponding value from a hash table, we must first decide what *bucket* it would be in. To do this, we run the **hash function** on the key, which generates the number of the bucket. Once we've found this bucket, we then need to look up the value which corresponds to our key in the bucket.

```
lookup :: String -> (HashTable String String) -> String
lookup str table =
  let hashValue = hash str
      (_, bucket) = head (filter ((== hashValue) . fst) table)
      (_, value)  = head (filter ((== str) . fst) bucket)
  in value
```

(a) Explain the ways in which function `lookup` can fail, with Haskell runtime errors. Don't worry about precise error messages, but *do* state what causes the errors. [5 marks]

(b) Add in error handling for function `lookup` above, using the `Maybe` type, to ensure this function is now total. Note that this will require changing the type of this function. You may assume the existence of a list function `find :: (a -> Bool) -> [a] -> Maybe a`. [14 marks]

(c) Add in error handling for the `lookup` 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 of the function. You may assume the existence of a list function `find :: (a -> Bool) -> [a] -> Maybe a`. The convention when using `Either` is to represent failure with the `(Left errMsg)` constructor, and success with the `(Right someValue)` constructor. [14 marks]

4. (a) Consider the following function definition:

```
diffsq [] = 0
diffsq (x:xs) = x * x - diffsq xs
```

Use the shorthand AST notation to show how the application `diffsq [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:

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

Show the evaluation of `take 2 (zig 20)` 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`, `zig`, and `zag` 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 $\langle root \rangle$`.in` (where $\langle root \rangle$ is the filename less its extension) opens that file, reads its contents, uses the function `hash :: String -> Int` from Question 3 to compute the hash of the **entire file** contents and outputs the result to file $\langle root \rangle$`.chk` (See the Reference, p8 for file input/output functions). [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])
zip          :: [a] -> [b] -> [(a,b)]
unzip        :: [(a,b)] -> ([a], [b])
zipWith      :: (a -> b -> c) -> [a] -> [b] -> [c]
minimum      :: (Ord a) => [a] -> a
```

Other Prelude Functions

```
const      :: a -> b -> a
fst        :: (a, b) -> a
snd        :: (a, b) -> b
($)        :: (a -> b) -> a -> b
(.)        :: (b -> c) -> (a -> b) -> a -> c
show       :: a -> String
```

Prelude Typeclasses

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: 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
```

© THE UNIVERSITY OF DUBLIN 2015