Durham University

Examination Paper

Examination Session:	Year:	Exam Code:
Model Exam	XXXX	COMP2221-WE01

Title: Programming Paradigms – Submodule Functional Programming

Time Allowed:	1 hour
Additional Material provided:	None
Materials Permitted:	None
Calculators Permitted:	No
Visiting Students may use dictionaries:	Yes

Instructions to Candidates:	Answer ALL questions. Please answer each section in a separate answer booklet.

Section A Functional Programming (Laura Morgenstern)

Except where otherwise stated, any code you write in this section must be in Haskell.

Question 1

(a) Haskell uses **lazy evaluation**. Describe how this differs from **eager evaluation** (as seen in C# or C) with reference to functions and their arguments. **[4 Marks]**

Solution: *Comprehension, Knowledge*
In eager evaluation, the arguments to functions are always fully evaluated before the function is applied [2 marks]. In contrast, in lazy evaluation, the function is applied first, before its arguments are evaluated [2 marks].

(b) Consider an operation `scan` which computes the prefix sum on lists of arbitrarily large integers. When given a list $[x_0, x_1, x_2, \ldots, x_{n-1}]$, `scan` should return the list $[y_0, y_1, y_2, \ldots, y_{n-1}]$ where $y_0 = x_0$, $y_1 = x_0 + x_1$, and generally $y_j = \sum_{i=0}^{j} x_j$. Implement `scan` recursively and make use of pattern matching in your answer. **[10 Marks]**

Solution: *Knowledge, Application*

```
scan :: [Integer] -> [Integer]
scan [] = []
scan [x] = [x]
scan (x:y:xs) = x : scan (x+y:xs)
```

- `scan :: [Integer] -> [Integer]` [2 marks]. Only [1 mark] if `Int`.
- `scan [] = []` [1 mark]
- `scan [x] = [x]` [1 mark]
- `scan (x:y:xs)` [2 marks] for the pattern match `x:y:xs`, [1 mark] for the brackets.
- `= x : scan (x+y:xs)` [1 mark] for the concatenation, [1 mark] for the recursive call, [1 mark] for application to `(x+y:xs)`.

(c) Now turn `scan` into a polymorphic, higher-order function. Rewrite `scan` as a new function, `scanf`, which accepts an additional argument that can be any binary operator. Ensure that `scanf` continues to yield the results of `scan` if you pass in `(+)` as the higher-order argument. **[4 Marks]**

Solution: *Comprehension, Application*

```
scanf :: (a -> a -> a) -> [a] -> [a]
scanf _ [] = []
scanf _ [x] = [x]
scanf f (x:y:xs) = x : scanf f (x `f` y : xs)
```

- Function argument is binary (three arguments) [1 mark]
- Generic type variable a for all arguments [1 mark]
- Extra parameter `f` (or similar) [1 mark]
- Replacing `x+y` with ``x `f` y`` or similar [1 mark]

Question 2

(a) Provide type annotations for the following functions. Include type constraints where required.

- A function `func1` that takes a string as input and returns the string reversed
- A function `func2` that takes two arbitrarily large integers and returns both as a pair
- A function `func3` that takes a list of arbitrary numerical parameters and returns their sum

[8 Marks]

Solution: *Knowledge,Application*

```
func1 :: [Char] -> [Char]
func2 :: Integer -> Integer -> (Integer, Integer)
func3 => Num a :: [a] -> a
```

continued

(b) Consider the following data type `Letter` that represents either a lowercase letter `Minuscule` or an uppercase letter `Majuscule`.

```
data Letter = Minuscule Char | Majuscule Char
```

Write a function `isLowercase` which returns `True` if the letter is a `Minuscule` and `False` otherwise. **[3 Marks]**

Solution: *Application, Comprehension*

```
isLowercase :: Letter -> Bool
isLowercase (Minuscule _) = True
isLowercase (Majuscule _) = False
```

(c) Explain the concept of functors in Haskell and write a `Functor` instance for the `Maybe` data type:

```
data Maybe a = Just a | Nothing
```

[6 Marks]

Solution: *Comprehension, Application*
A functor is a container that can be mapped over to transform its elements while the structure of the container remains unchanged. In Haskell, a `Functor` has to implement the generic mapping function `fmap` which must fulfill the `Functor` laws.

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

Question 3

(a) Reduce the λ-expression $(\lambda x.(xx))((\lambda y.(ay))b)$ to normal form. **[4 Marks]**

Solution: *Application,Comprehension*

Apply β-reduction, e.g. as follows:

$(\lambda x.(xx))((\lambda y.(ay))b)|[b:y]$

$(\lambda x.(xx))(ab)$

$(\lambda x.(xx))(ab)|[(ab):x]$

$(ab)(ab)$

(b) In the λ-calculus, functions take exactly one argument. In practice, however, we often require higher-order functions with multiple parameters. Write down a λ-expression that can model a higher-order function with two input parameters. How is this technique called? **[4 Marks]**

Solution: *Comprehension, Application*

$(((\lambda x.(\lambda y.(xy)))a)b) \rightarrow$ Currying

END OF PAPER