The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 4 MODULE, AUTUMN SEMESTER 2018–2019

REAL-WORLD FUNCTIONAL PROGRAMMING

Time allowed ONE AND A HALF hours

Candidates may complete the front cover of their answer book and sign their desk card but must NOT write anything else until the start of the examination period is announced.

Answer ALL THREE questions

No calculators are permitted in this examination.

Dictionaries are not allowed with one exception. Those whose first language is not English may use a standard translation dictionary to translate between that language and English provided that neither language is the subject of this examination. Subject-specific translation directories are not permitted.

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

DO NOT turn examination paper over until instructed to do so

INFORMATION FOR INVIGILATORS: none

Question 1

This question is about functional programming in a real-world context.

- Drawing from what has been covered in this module, and then in particular the guest lectures, give two examples of where functional programming has been used successfully in the "real world". For the purpose of this question, "real world" is to be understood as addressing some problem of general interest that is not intrinsically motivated by functional programming in itself. Your examples can be of a general area or something more specific, and it can be either functional programming as such or functional programming ideas being applied more broadly. For each example, provide enough context and outline the reasons for success so as to make a convincing case. (20)

- Explain what the QuickCheck framework is, the key ideas behind it, and why it is attractive from a real-world perspective. Illustrate with a small, concrete example. (10)

Question 2

The following type is a Haskell representation of streams (infinite sequences):

```
data S a = a `Fby` (S a)
```

The name `Fby` of the stream constructor is short for "followed by".

(a) Provide a `Functor` instance for `S`:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

(4)

(b) Provide an `Applicative` instance for `S`:

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

(8)

(c) Streams of numbers can be seen as numbers themselves. Provide a `Num` instance for streams of numbers to that end, defined in terms of `pure` and `(<*>)` (we are only considering part of the class `Num`):

```
class Num a where
    (+), (-), (*) :: a -> a -> a
    abs :: a -> a
    fromInteger :: Integer -> a

instance Num a => Num (S a) where ...
```

(8)

(d) Define the stream of natural numbers

```
nat :: S Integer
```

using only `Fby` and streams as numbers. (4)

(e) Define the stream of Fibonacci numbers

```
fib :: S Integer
```

using only `Fby` and streams as numbers. (6)

Question 3

Dynamic Programming (DP) is an important method for solving optimization problems. It is applicable whenever a problem exhibits *optimal substructure* (the overall problem can be solved by combining optimal solutions to subproblems) and *overlapping subproblems* (the same subproblem will appear more than once when the problem is decomposed). The central idea is to tabulate the subproblem solutions to ensure each subproblem is solved at most once.

(a) Explain why lazy functional programming is a good fit for DP. (10)

(b) Minimizing the cost of a path through a cost matrix is an example of a problem where DP is applicable:

> Given an $m \times n$ matrix of costs, find the minimum cost to reach the last cell $(m-1, n-1)$ from cell $(0,0)$, moving either right or down; i.e., only cells $(i, j+1)$ and $(i+1, j)$ are reachable from (i, j).

For example, the minimum cost path for the following matrix is 17:

$$\begin{bmatrix} 3 & 1 \\ 5 & 9 \\ 2 & 7 \end{bmatrix}$$

The minimal cost for reaching cell (i, j) in a given matrix $costs$ is defined as follows:

$$minCost(i,j) = \begin{cases} costs(0,0) & \text{if } i = j = 0 \\ costs(0,j) + minCost(0, j-1) & \text{if } i = 0 \\ costs(i,0) + minCost(i-1, 0) & \text{if } j = 0 \\ costs(i,j) + \min(minCost(i, j-1), minCost(i-1, j)) & \text{otherwise} \end{cases}$$

Write a Haskell function `minCost` that finds the minimum cost path demonstrating the use of lazy evaluation for DP:

```
type Cost = Int

minCost :: Array (Int,Int) Cost -> Cost
```

(10)

(c) Show how modify the solution to also return all minimum paths, each represented as a sequence of individual costs:

```
type Path = [Cost]
type CostPaths = (Cost, [Path])

minCostPaths :: Array (Int,Int) Cost -> CostPaths
```

(20)