# The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 4 MODULE, AUTUMN SEMESTER 2018–2019

**REAL-WORLD FUNCTIONAL PROGRAMMING**

# ANSWERS

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.*

## Note: ANSWERS

**Knowledge classification:** Following School recommendation, the (sub)questions have been classified as follows, using a subset of Bloom's Taxonomy:

K: Knowledge

C: Comprehension

A: Application

Note that some questions are closely related to the coursework. This is intentional and as advertised to the students; the coursework is a central aspect of the module and as such partly examined under exam conditions.

**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)

  **Answer:** *[K] Many possibilities, including map-reduce, Erlang, data science, financial applications, pharma and biotech start-ups.*

- 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)

  **Answer:** *[K,C] Framework for property-based testing with flexible language for stating properties. Random test cases generated automatically based on type of argument(s) to properties. Counterexample produced when test case fails, and can automatically be shrunk in attempt to find minimal counterexamples. Thanks to automatic test-case generation and ease of customization, quick check helps achieving very good test coverage with comparatively little effort.*

  *To illustrate, one property of the function that reverses a list is that reversing a list twice is the identity on lists. This property can be captured as a QuickCheck property in Haskell as follows:*

```
prop_RevRev :: Eq a => [a] -> Bool
prop_RevRev xs = reverse (reverse xs))
```

*QuickCheck will verify such a property by applying it to a number of randomly generated lists. Should it fail on some list, QuickCheck will automatically try to find a shorter list, starting from the one found, for which the test still fails and present this as a counter example.*

**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)

**Answer:** *[A]*

```
instance Functor S where
    fmap f (a `Fby` as) = (f a) `Fby` (fmap f as)
```

(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)

**Answer:** *[A]*

```
instance Applicative S where
    pure a = as
        where
            as = a `Fby` as

    (f `Fby` fs) <*> (a `Fby` as) = (f a) `Fby` (fs <*> as)
```

(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)

**Answer:** *[A]*

```
instance Num a => Num (S a) where
    xs + ys = pure (+) <*> xs <*> ys

    xs - ys = pure (-) <*> xs <*> ys

    xs * ys = pure (*) <*> xs <*> ys

    abs xs = pure abs <*> xs

    fromInteger x = pure (fromInteger x)
```

(d) Define the stream of natural numbers

```
nat :: S Integer
```

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

**Answer:** *[A]*

```
nat :: S Integer
nat = 0 `Fby` (nat + 1)
```

(e) Define the stream of Fibonacci numbers

```
fib :: S Integer
```

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

**Answer:** *[A]*

```
fib :: S Integer
fib = 1 `Fby` (fib + 0 `Fby` fib)
```

**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)

**Answer:** *[K]*

*The equations that define the solution can be transliterated directly into the language without any need to be explicit about the order in which the subproblems are solved, or indeed exactly which subproblems need to be solved. The demand-driven aspect of lazy evaluation ensures that the solutions are computed as and only if needed, at most once, in an appropriate order. Having the order induced automatically by demand is a major advantage compared with typical solutions in strict languages (imperative or not). However, note that it still is necessary to tabulate the solutions: the "evaluate at most once" aspect of lazy evaluation applies to individual, physical redexes. It does* not *mean that only one of a number of physically distinct but identical redexes are evaluated, and the others automatically updated with the result. That effect can be achieved by* memoization of functions. *Such a facility can be implemented in typical lazy language implementations, but generally would have to be invoked explicitly for functions of interest. It is not the default.*

(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)

**Answer:** *[A]*

```
minCost :: Array (Int,Int) Cost -> Cost
minCost costs = minCosts ! ub
    where
        bs@(lb, ub) = bounds costs

        minCosts = array bs [ (ij, mcAux ij) | ij <- indices costs ]

        mcAux (0,0) = costs ! (0,0)
        mcAux (0,j) = costs ! (0,j) + minCosts ! (0, j - 1)
        mcAux (i,0) = costs ! (i,0) + minCosts ! (i - 1, 0)
        mcAux (i,j) = costs ! (i,j) + min (minCosts ! (i, j - 1))
                                          (minCosts ! (i - 1, j))
```

(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)

**Answer:**

```
cpsOne :: Cost -> CostPaths
cpsOne c = (c, [[c]])


cpsAdd :: Cost -> CostPaths -> CostPaths
```

```
cpsAdd c (cps, ps) = (c + cps, map (c:) ps)


cpsMin :: CostPaths -> CostPaths -> CostPaths
cpsMin cpsps1@(cps1, ps1) cpsps2@(cps2, ps2)
    | cps1 < cps2 = cpsps1
    | cps2 < cps1 = cpsps2
    | otherwise   = (cps1, ps1 ++ ps2)

minCostPaths :: Array (Int,Int) Cost -> CostPaths
minCostPaths costs = minCPss ! ub
    where
        bs@(lb, ub) = bounds costs

        minCPss = array bs [ (ij, mcpsAux ij) | ij <- indices costs ]

        mcpsAux (0,0) = cpsOne (costs ! (0,0))
        mcpsAux (0,j) = cpsAdd (costs ! (0,j)) (minCPss ! (0, j - 1))
        mcpsAux (i,0) = cpsAdd (costs ! (i,0)) (minCPss ! (i - 1, 0))
        mcpsAux (i,j) = cpsAdd (costs ! (i,j))
                               (cpsMin (minCPss ! (i, j - 1))
                                       (minCPss ! (i - 1, j)))
```