Introduction to Programming: End-semester Examination

Date: November 23, 2026 **Time:** 0930 – 1230 hrs **Marks:** 100 **Weightage:** 40%

This paper has two parts.
There are 8 questions in Part A, each worth 5 marks. For this part, provide the solution and a short explanation justifying your answer.
There are 4 questions in Part B, each worth 15 marks. Syntax should more or less correspond to Haskell, but minor inaccuracies like wrong indentation will be ignored. You can assume standard library functions, but if you use something not taught in class, you must explain briefly what it does.

Part A

(8 × 5 = 40 marks)

1. Define the function `unzip :: [(a, b)] → ([a], [b])`, which splits a list of pairs into the list of first components and the list of second components. Then evaluate the following expression:

```
unzip (zip "hello" [1, 2, 3])
```

2. Evaluate the following expression:

```
map fst (filter (odd . snd) (zip "abcde" [1 ..]))
```

3. What is the running time of `rev xs`, as a function of the length n of `xs`?

```
rev :: [a] → [a]
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

4. What is the running time of `h n`, as a function of n? How would you modify the definition so that it runs in time $O(\log n)$? Recall that `div` on `Int`s is a constant time operation.

```
h :: Int → Int
h 0 = 1
h n = h (n `div` 2) + h (n `div` 2)
```

5. What is the most general type of the following function, including any class constraints?

```
f x y = x == y || x < y
```

6. Consider the following declaration.

```
data Direction = North | East | South | West
```

What happens when we evaluate `elem East [North, South]`? What change to the program makes this expression evaluate to **`False`**?

7. Consider the following declarations. List all values of type `T` in increasing order.

```
data Colour = Red | Green | Blue deriving (Eq, Ord)
data T = A | B Colour | C (Maybe Bool) deriving (Eq, Ord)
```

8. Evaluate the following two expressions:

```
scanl (-) 10 [1, 2, 3]
scanr (-) 0 [1, 2, 3]
```

Part B

(4 × 15 = 60 marks)

1. Consider the following definitions.

```
from :: Int → [Int]
from n = n : from (n + 1)

square :: Int → Int
square x = x * x

firstOver :: Int → [Int] → Int
firstOver k (x:xs)
  | x > k     = x
  | otherwise = firstOver k xs
```

Recall that `map f [] = []` and `map f (x:xs) = f x : map f xs`.

(a) Trace the computation of the following under Haskell's lazy evaluation strategy.

```
firstOver 10 (map square (from 1))
```

(b) Explain briefly why this computation terminates even though `from 1` is an infinite list. What would happen under an eager strategy that fully evaluates the arguments of a function before applying it?

(c) What happens when we evaluate the following expression?

```
firstOver 10 (map square [1, 2, 3])
```

Note: *Tracing* means that you write a sequence of rewrites, where you start with the original

expression, and each step rewrites an appropriate subexpression by following the program. Under lazy evaluation, a subexpression is rewritten only when its value is needed to make progress.

2. Consider the following type of binary trees, and its height function.

```
data BTree = Leaf | Node BTree BTree

height :: BTree → Int
height Leaf       = 0
height (Node l r) = 1 + max (height l) (height r)
```

   (a) List all trees `t :: BTree` with `height t ⩽ 2`. How many trees have height at most 3? (You need not list them.)

   (b) Write a function `next :: [BTree] → [BTree]` such that, if `ts` is the list of all trees of height at most n (without duplicates), then `next ts` is the list of all trees of height at most $n + 1$ (without duplicates). Use it to define `upto :: Int → [BTree]`, where `upto n` is the list of all trees of height at most `n`, for `n ⩾ 0`.

   (c) Write a function `f :: Int → [BTree]` such that `f n` is the list of all trees of height *exactly* `n`, without duplicates, for `n ⩾ 0`. *Hint:* You may use `upto` and `height`.

3. Recall that binary search trees are represented using the following type.

```
data BST = Nil | Node BST Int BST
```

   In each tree `Node l x r`, all values in `l` are less than `x` and all values in `r` are greater than `x`.

   (a) Write `insert :: Int → BST → BST` running in time $O(h)$, where h is the height of the tree. Inserting a value already in the tree should leave the tree unchanged.

   (b) Write `inorder :: BST → [Int]` that returns the elements in increasing order, running in time $O(n)$, where n is the number of nodes.

   (c) Write `inRange :: Int → Int → BST → [Int]` s.t. `inRange lo hi t` returns, in increasing order, all elements `x` of `t` with `lo ⩽ x ⩽ hi`. It should run in time $O(h + k)$, where h is the height and k is the number of elements returned.

4. Write a program `main :: IO ()` that repeatedly reads lines from the input. Each line is either an integer or the word `done`.

   - After reading an integer, the program prints the running total of all integers read so far.
   - On reading `done`, the program prints how many integers were read and the largest of them, and then stops. If no integers were read, it prints `No numbers` instead.

   You may assume that the input is well-formed.