Final Exam

Cover Page

Total marks: 200
Reading period: 15 Minutes duration (no use of keyboard or selection of answers).
Writing period: 180 Minutes duration.
Permitted materials: One A4 page with notes on both sides, Unannotated paper-based dictionary.
You may not use any devices in the exam room other than the provided lab computers.

Note that links to the documentation of the `Prelude` and `Data.List` libraries are provided on your desktop.

Questions are not of equal value.
All questions must be completed within Ed Lessons.

This is a closed examination. You may not copy this exam.

Multiple Choice

Question 1

Consider the following expressions in Haskell:

1. `"Hello" ++ "World"`
2. `"Hi" !! 1`
3. `head "Bye"`
4. `length "Haskell"`

Which of the following statements is **TRUE**?

- Expression 1 results in `"Hello World"` (with a space).
- Expression 2 returns `'H'`.
- Expression 3 returns `"B"`.
- Expression 4 returns `7`.

Question 2

Consider the following expressions in Haskell:

1. `5 + 3`
2. `5 / 3`
3. `5 :: Int`
4. `5.0 * 3.0`

Which of the following statements is **TRUE?**

- All four expressions have type `Int`.
- Expression 2 (`5 / 3`) will cause a compile-time type error because `/` only works on fractional types.

- Expression 4 has type `Int` because multiplying two numbers always produces an integer.
- None of the above statements are true

Question 3

Here's a mysterious function in Haskell with no type signature.

```
mystery x y = x * length y
```

Which of the following is a correct type signature for `mystery`?

-
```
mystery :: Int -> [a] -> Int
```
-
```
mystery :: [a] -> Int -> Int
```
-
```
mystery :: Num a => a -> [a] -> a
```
-
```
mystery :: [a] -> [b] -> [c]
```

Question 4

Here's a function on lists:

```
f :: [Int] -> Int
f xs = case xs of
    [x]     -> x
    (x:y:_) -> x + y
    []      -> 0
```

Which of the following functions below is equivalent to the definition above?

-
```
g xs
  | length xs >= 2 = head xs + head (tail xs)
  | length xs >= 1 = head xs
  | otherwise      = 0
```

```
h xs = case xs of
    (x:_:_) -> x + head xs
    [x]     -> x
    []      -> 0
```

```
k xs
  | length xs >= 1 = head xs
  | length xs >= 2 = head xs + head (tail xs)
  | otherwise      = 0
```

```
m xs = head xs + head (tail xs)
```

Question 5

Here is a bad implementation of the function `drop` that has a mistake in it. What is the result of this mistake?

```
dropBad :: Int -> [a] -> [a]
dropBad n xs
  | n <= 0    = xs
  | null xs   = []
  | otherwise = dropBad (n - 1) xs
```

- It will lead to an infinite recursion
- It will take the tail of an empty list and crash
- It will always return the input list regardless of the input given.
- It will always return the empty list regardless of the input given.

Question 6

Here are two functions, `f1` and `f2`:

```
f1 lst = case lst of
  [] -> []
```

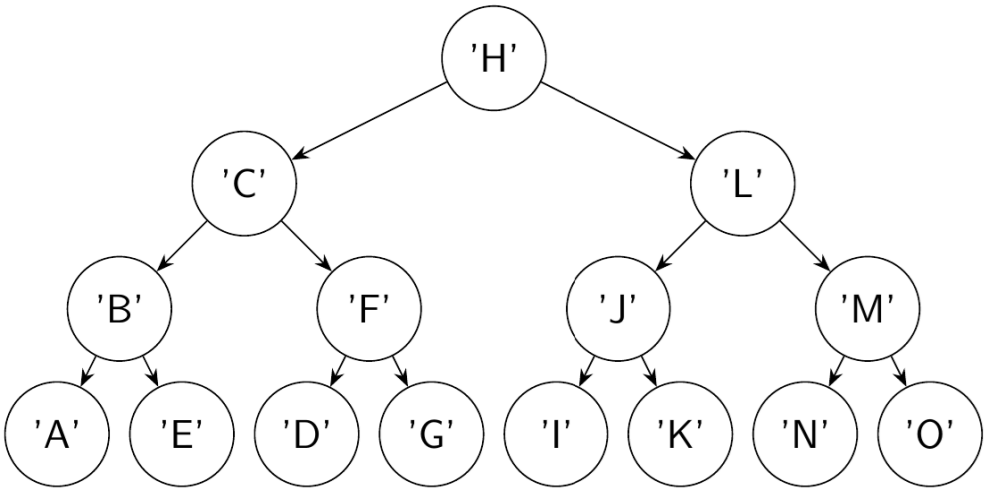
```
  (x:xs)
    | x > 0     -> (x * x) : f1 xs
    | otherwise -> f1 xs

f2 xs = map (\x -> x * x) (filter (>0) xs)
```

Which of the following statements is **TRUE**?

- `f1` and `f2` always give the same result for all inputs.
- `f2` has a more general type than `f1`
- They differ on singleton lists.
- They can return lists of different lengths

Question 7

Consider the following binary tree:



List **all** characters which are the roots of trees in this picture that are **not** correctly defined binary search trees.

- `'B'`, `'C'`, `'F'`, `'H'`, `'J'`, `'L'`, `'M'`
- `'C'`, `'H'`, `'L'`, `'M'`
- `'C'`, `'M'`

- `'H'`, `'L'`, `'M'`
- `'M'` only

Question 8

Suppose we take an empty stack of `Int`s, and apply the following operations, in order from top to bottom:

- `push 1`
- `push 2`
- `push 3`
- `pop`
- `push 4`
- `pop`

What is at the top (the next value to be popped) of the resulting stack?

- `1`
- `2`
- `3`
- `4`
- No answer is correct, because `pop` is a partial function, and will throw an error in this sequence.

Question 9

A 'binary' representation of natural numbers in Haskell might be defined using lists of 'bits' (zeros and ones) as follows:

```
data Bit = Zero | One

type BinaryNumber = [Bit]
```

The intended meaning of the first element of such a list is the number of ones; of the second element

is the number of twos; the third element is the number of fours; then eights; then sixteens; and so on.

- e.g. `[One,Zero,Zero,One,One]` is understood as `(1*1)+(0*2)+(0*4)+(1*8)+(1*16) = 25`.

What is the worst case time complexity of the following function, which doubles (multiplies by two) a binary number? Here `n` should be understood to be the length of the input list.

```
double :: BinaryNumber -> BinaryNumber
double x = Zero : x
```

- O(1)
- (O log n)
- O(n)
- O(n log n)
- $O(n^2)$

Question 10

Consider the following function, which has type `BinaryNumber -> BinaryNumber` and computes the successor (adds one to) a binary number.

```
succ x = case x of
  []        -> [One]
  Zero : xs -> One : xs
  One : xs  -> Zero : succ xs
```

Describe the situation in which this function will exhibit its **best case** time complexity, with respect to 'big O' analysis.

- The input list contains no elements
- The input list has `Zero` as its leftmost element
- The input list has `Zero` as its rightmost element

- The input list has `One` as its leftmost element
- The input list has `One` as its rightmost element

Question 11

What is the **worst case** time complexity of the `succ` function defined above? `n` should again be understood to be the length of the input list.

- O(1)
- O(log n)
- O(n)
- O(n log n)
- O (n^2)

Question 12

Consider the following function, which has type `Int -> BinaryNumber` and maps integers to their binary representation.

```
toEnum x
  | x <= 0    = []
  | even x    = Zero : toEnum (div x 2)
  | otherwise = One : toEnum (div x 2)
```

What is the worst case time complexity of this function? Here `n` should be understood to be the input itself. You may assume that the guards `x <= 0` and `even x` are computed in O(1) time.

- O(1)
- O(log n)
- O(n)

- O(n log n)
- O (n^2)

Programming Q1: Sum of Squares

Implement a function `sumSquares :: Int -> Int -> Int` that takes two integers and returns the sum of their squares.

Example:

```
sumSquares 3 4 -- returns 25
sumSquares 0 5 -- returns 25
```

Programming Q2: Traffic Lights

Define a data type `TrafficLight` with constructors `Red`, `Yellow`, and `Green`. Then write a function `nextLight :: TrafficLight -> TrafficLight` that returns the next traffic light in the (Australian) sequence: Red → Green → Yellow → Red.

Example:

```
nextLight Red    -- returns Green
nextLight Yellow -- returns Red
```

Programming Q3: Digit Sum

Write a recursive function `digitSum :: Int -> Int` that computes the **sum of the digits** of a non-negative integer.

Example:

```
digitSum 123  -- returns 1 + 2 + 3 = 6
digitSum 405  -- returns 4 + 0 + 5 = 9
digitSum 0    -- returns 0
```

Hint:

To get the last digit of a number `x`, use `mod x 10`, and to get the number `x` without its final digit, use `div x 10`.

Note: You will not be tested on negative inputs, and may handle them as you like.

Programming Q4: Double Evens

Write a Haskell function `doubleEvens :: [Int] -> [Int]` that takes a list of integers and returns a new list where all the even numbers are doubled, and the odd numbers remain unchanged.

Examples:

```
doubleEvens [1,2,3,4]  -- returns [1,4,3,8]
doubleEvens [0,5,6]    -- returns [0,5,12]
doubleEvens []         -- returns []
```

Programming Q5: Product of Lengths

Write a recursive function `productOfLengths :: [[a]] -> Int` that takes a list of lists and returns the product of the lengths of each sublist.

Example:

```
productOfLengths [[1,2],[3,4,5],[6]] -- returns 2*3*1 = 6
productOfLengths []                  -- returns 1
```

Programming Q6: Interleaving

Write a recursive function `interleave :: [a] -> [a] -> [a]` that takes two lists and interleaves their elements, starting with the first list, including any remaining elements from the longer list once the shorter list runs out.

Example:

```
interleave [1,3,5] [2,4,6]         -- returns [1,2,3,4,5,6]
interleave ["a","b"] ["x","y","z"] -- returns ["a","x","b","y","z"]
interleave [1,2] []                -- returns [1,2]
interleave [] [3,4]                -- returns [3,4]
```

Programming Q7: Custom Lists and Folds

Complete the function `customFoldl`:

A type of Custom Lists is provided for you. Implement fold-left on this type.

Recall that fold-left takes as inputs

- a combining operation,
- a starting value of an accumulator, and
- a Custom List that we wish to perform recursion on;

and is defined by:

- if the input list is empty, return the accumulator;
- if the input list is non-empty, use the combining operation and the list head to update the accumulator, then continue the recursion on the list tail.

Do **not** use any operation on Haskell's built-in lists to write this function.

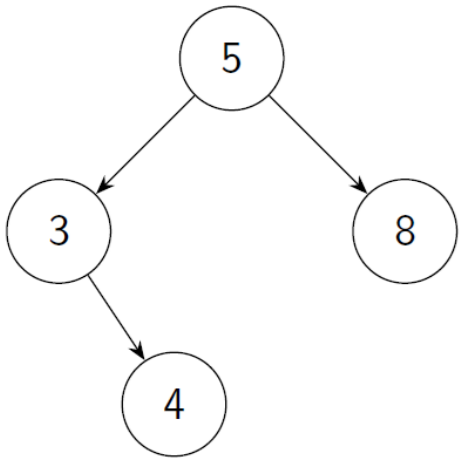
Programming Q8: Binary Trees

Complete the function `atPosition`:

Along with a standard definition of a type of Binary Trees, you are given a type of `Position`s. This is the type of positions in a tree at which a node might appear. Each node might be

- a `Root`, with no parents, or
- a `Leaf`, with no children, or
- an `Internal` node, with one parent and at least one child.

For example, in the picture below the node labelled `5` is the `Root`, the nodes with `8` and `4` are both `Leaf`s, and the node with `3` is the only `Internal`.



Note that in a tree with only one node, that node is both `Root` and `Leaf`.

Given a `Position` and a Binary Tree, return a list (in any order) of all values in the tree appearing in that Position.

Note that this code generates an `import of `Data.List' is redundant` warning. You should ignore this warning.

Programming Q9: Rose Trees

Complete the function `maxBranch`:

Given a Rose Tree, return the largest number of children that any node has.

Programming Q10: Ad Hoc Polymorphic List Functions

There are **two** functions to complete in this file, each worth **12** marks. You do *not* need to have completed the first, to attempt the second.

1, Complete the function `notAtBounds`:

Given a list whose values are in both the `Eq` and `Bounded` typeclasses, return this list in the same order, but with every occurrence of `minBound` or `maxBound` removed.

2, Given a type in the typeclass `Bounded`, make lists of that type an instance of `Bounded`, by completing the function `maxBound`:

The maximum element of this type of lists should be the stream (infinite list) whose every member is `maxBound`.

The minimum element of this type of lists is the empty list. This definition of `minBound` is provided for you and does not need to be edited.

Note that this code generates an `Orphan instance` warning. You should ignore this warning.

Programming Q11 : Queues

Complete the function `queueElem`:

Given a value, and a queue of values of that type, return `True` if and only if that value appears somewhere in the queue.

Your function must work with **any** instance of the `Queue` type constructor.

There are two definitions provided, at the bottom of the file, of `Queue` instances. These are provided for testing purposes only, and there is no need to edit them.