Homework Assignment 3

Functional and Logic Programming, 2025

Due date: Tuesday, May 20th, 2025 (20/05/2025)

Bureaucracy

- The staff member in charge of this assignment is Gal Lalouche (gal.lalouche@post.runi.ac.il).
- Submission is in pairs, but solo submission is also allowed. We very much suggest you pair-up, as solving this exercises with another person greatly enhances your learning (as well as being more fun!).
- To submit, create a zip file named `HW3_<id1>_<id2>.zip` where `<id1>` and `<id2>` are the submitters IDs.
  - Or `HW3_<id>.zip` if submitting alone.
  - You do not need special permission to submit alone.
- The zip file should contain **a single, top-level file (no folders!)** named `HW3.hs`!
- Make sure your submission compiles successfully. Submissions which do not compile will not receive a grade!
  - We will be using the following command to compile the file: `ghc -Wall -Werror HW3.hs`.
- You may submit the assignment after the due date even without approval (e.g., excluding reserve duty, serious illness, or other cases covered by the student administration).
  - You will be penalized for **5 points for every late day**.
  - The **maximum** extension allowed by this is **3 days**.
- If you don't know how to implement some function, **do not remove it**! Use `undefined` in the implementation, otherwise your entire submission will fail compilation, which will also result in no grade.
  - This is especially true to for the bonus section!

General notes

- The instructions for this exercise are split between this file and `HW3.hs`. This file offers a more high-level overview of the exercise, as well as offering a few hints and examples. The Haskell file details all the required functionality for this assignment.
- You may **not** modify the `import` statement at the top of the file, nor add new `import`s.
  - N.B. HLS has the habit of adding **unnecessary imports** when it doesn't recognize an identifier, **so please double-check this before submitting**!
  - If you are unsure what some function does, you can ask HLS or Hoogle.
  - Hoogle also supports module lookup, e.g., `Prelude.not`.
  - Do be aware however, that some functions from the standard library are more general than what we have learned so far in class.
    - And in some cases their definition may not be entire clear just yet!
- The exercises within each section are defined in a linear fashion. That is, it is a good idea to use previously defined functions (either from the same section or ones before it). It is also a good idea to use functions you saw in class; some of them are already imported, and some of them you would have to define yourself.
  - Do not be alarmed by the large amount of functions! Many of them are simple one-liners, and were designed to aid you in solving the more complex functions.
  - In general, you may define as many helper functions as you wish.
  - If you're feeling fancy, you might even implement some functions using later ones!
- Try to write elegant code, as taught in class. Use point-free style, η-reductions, and function composition to make your code shorter and more declarative. Although it is not *strictly required* in the homework assignments, elegant code *will* receive a higher-grade in the test, so this is a good exercise. HLS and hlint can be very helpful in this.
  - Do note that in some cases, hlint may suggest functions which are not imported!
- If possible, please ask your questions first in Piazza, as this will allow all students to take part in the discussion.
- You can get a grade **greater than 100** if you do the bonus!
  - Assuming you have a passing grade in both the homework and the exam, the final grade formula is:

$$0.7 ExamGrade + 0.3 \sum_{i=1}^{6} \frac{HW_i}{6} + AttendanceBonus$$

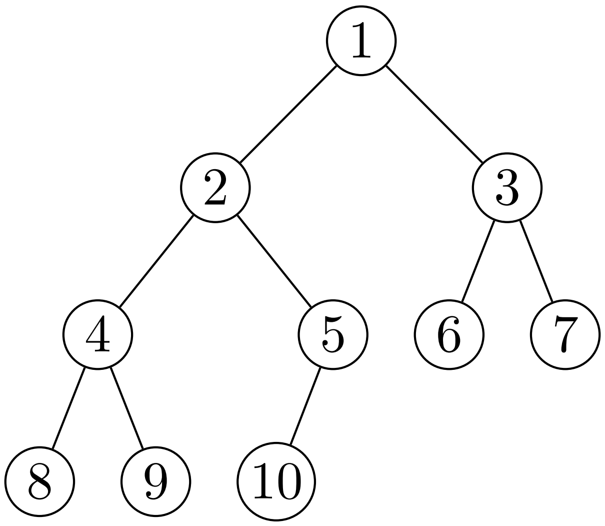
    so a grade greater than 100 can increase your final grade (or help ensure you get a passing grade in the homework).
  - Yes, you can also get bonus marks for the exam!
  - The final grade is still capped at 100, unfortunately.

A note on testing

- The suggested way of testing yourselves is either via `GHCi` or using `-- >>>` comments, as taught in the tutorials.
- Although we do not cover unit tests in class, adventurous students can use the internet or even a generative AI to generate a template file for writing Haskell tests.
  - Personally, we recommend looking into HUnit (see tutorial here), or a wrapper for HUnit called Tasty.
  - N.B. A **very** important part of programming in general is writing your own unit tests, so try not to offload *too* much work to your friendly neighborhood chatbot.

Section 1: Trees

In this section we will implement functions related to `Trees`. As mentioned in the lecture, these trees are not necessarily binary **search** trees, that is, there is no assumed ordering of the elements. An example of a non-binary search tree:



The above tree is defined in Haskell as:

```haskell
-- As defined in the lecture
data Tree a = Empty | Tree (Tree a) a (Tree a) deriving (Show, Eq)

-- Useful utility
singleTree :: a -> Tree a
singleTree x = Tree Empty x Empty

tree :: Tree Int
tree =
  Tree
    (Tree
      (Tree
        (singleTree 8)
        4
        (singleTree 9))
      2
      (Tree
        (singleTree 10)
        5
        Empty))
    1
    (Tree
      (singleTree 6)
      3
      (singleTree 7))
```

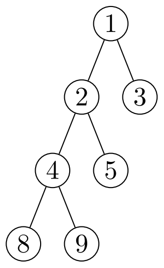
- `treeSize` and `treeHeight` measure the number of elements and the height of a tree, respectively.

```
treeSize tree
10
treeHeight Empty
0
treeHeight tree
4
```

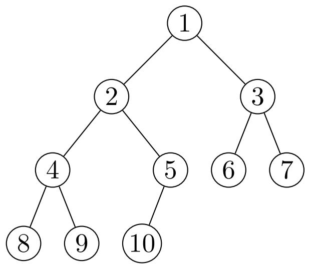
- As you know, there are multiple ways of traversing a tree. Since we have no side effects, one way of defining tree traversal is by converting a tree to a list.

```
preOrder tree
[1,2,4,8,9,5,10,3,6,7]
inOrder tree
[8,4,9,2,10,5,1,6,3,7]
postOrder tree
[8,9,4,10,5,2,6,7,3,1]
```

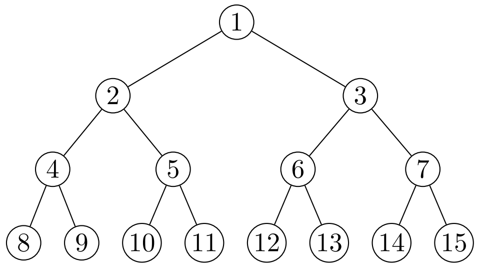
- We can define several taxonomies of binary trees:
  - **Full.** Every node has either 0 or 2 children.



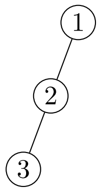
  - **Complete.** Every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible



  - **Perfect.** Every node has 0 or 2 children, and all leaves are at the same level.

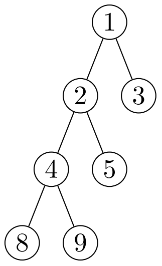


- **Degenerate.** every node has at most 1 child.

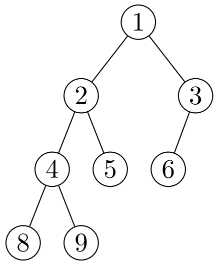


The function `classify` returns the tree classification. Note that the classification `FullAndComplete` should be returned if and only if the tree is both full and complete, but **not** perfect. Likewise, `Perfect` should be returned if and only if the tree is perfect, since every perfect tree is also full and complete. A tree of size **1 or less** is considered perfect, despite also being degenerate.

- A tree is balanced if the height of the left and right subtrees differ by at most 1, and both subtrees are balanced. An empty tree is balanced. The function `isBalanced` checks if a tree is balanced. Hint: There exists a solution in $O(n)$ time complexity, where n is the number of nodes in the tree. While our automatic tests won't cover this, in the test, more efficient solutions (in terms of asymptotic complexity classes) will receive a higher grade!



An unbalanced tree



A balanced tree

Section 2: Infinite lists

In this section we will study infinite lists in more depth. Unlike regular lists `[a]`, which may or may not be infinite, we can define lists which are **necessarily** infinite.

```
-- (:>) is the constructor in this case
data InfiniteList a = a :> InfiniteList a
infixr 5 :>
```

- Since regular lists can also be infinite, it's possible to convert an infinite list to a regular list. Which gives us access to all the existing list functions!

```
sampleN :: Int -> InfiniteList a -> [a]
sampleN n = take n . itoList
sample :: InfiniteList a -> [a]
sample = sampleN 10
smallSample :: InfiniteList a -> [a]
smallSample = sampleN 5

sample $ irepeat 3
[3,3,3,3,3,3,3,3,3,3]
```

Some examples of the functions you should implement:

```
smallSample $ irepeat 1
[1,1,1,1,1]
smallSample $ iiterate (\ x -> x * x + x) 1
[1,2,6,42,1806]
sample naturals
[0,1,2,3,4,5,6,7,8,9]
sample $ imap (* 3) naturals
[0,3,6,9,12,15,18,21,24,27]
```

- `iconcat` can result in a finite number of elements (or even no elements at all!) if only a finite number of the input lists are non-empty. Since the type itself is still an `InfiniteList`, this can cause programs to never halt:

```
sample $ iconcat $ iiterate (map (+1)) [1,2,3]
[1,2,3,2,3,4,3,4,5,4]
smallConcat = iconcat $ [1] :> [2] :> [3] :> [4] :> [5] :> irepeat []
smallSample smallConcat
[1,2,3,4,5]
sampleN 6 smallConcat -- will never halt
```

- `grouped` groups every n elements of the list into a sublist.

```
smallSample $ grouped 3 naturals
[[0,1,2],[3,4,5],[6,7,8],[9,10,11],[12,13,14]]
```

- `reverseN` reverses every n elements of the list.

```
sampleN 12 $ reverseN 3 naturals
[2,1,0,5,4,3,8,7,6,11,10,9]
```

2.1 Approximations

In the tutorial, you saw a way of approximating the square root of a number using the `Newton-Raphson` method. Specifically, you saw a way of generating an infinite list of approximations.

```haskell
-- Reminder:
sqrtSeries :: Double -> [Double]
sqrtSeries x = iterate (\ current -> (current + x / current) / 2) x
```

In this part, we will replace `Double` with `Rational`, to allow arbitrary precision. Don't worry, the formula is identical, as the `Rational` type supports all the basic arithmetic operations as `Double`![1]

```haskell
sqrtSeriesRatio :: Ratioal -> [Rational]
sqrtSeriesRatio x = iterate (\ current -> (current + x / current) / 2) x
```

Functions for working with `Rational`:

```haskell
-- Creating Rationals
half = 1 % 2
third = 1 % 3
numerator half
1
denominator half
2
84 % 2 -- Rational already deals with reduced fractions
42 % 1
```

- Since we're dealing in this section with `InfiniteList`, implement the function `sqrtInf` which returns an `InfiniteList Rational` of approximations.

```haskell
smallSample $ sqrtApprox 2
[2 % 1,3 % 2,17 % 12,577 % 408,665857 % 470832]
```

- It can be helpful to convert a rational number to a (possibly infinite) decimal string[2]. We can define an **infinite** `String` similarly to `String` using `InfiniteList`:

```haskell
type InfiniteString = InfiniteList Char
```

To makes things simpler, we will treat all decimal representations of a `Rational` number as always infinite, even if they can be represented finitely, by padding zeros. The function `longDivision` takes a `Rational` number and returns its decimal representation as an `InfiniteString` using the long division algorithm.

```haskell
sample $ longDivision $ 2 % 7
"0.28571428"
smallSample $ longDivision $ 1 % 3
"0.333"
smallSample $ longDivision $ 1 % 2
"0.500"
smallSample $ longDivision $ 3 % 2
"1.500"
smallSample $ longDivision $ (-3) % 2
"-1.50"
```

[1]In fact, we can generalize this function to work for any numeric type! But for that, we need the `Num` type class, which isn't covered in this assignment.

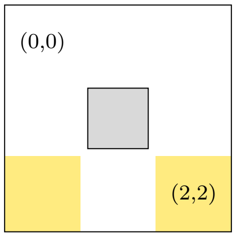
[2]Of course, the decimal representation is finite only if the (reduced) denominator prime factorization contains only powers of 2 and 5.

Lastly, implement the function `sqrtStrings`, which creates an infinite list of infinite decimal representations for each approximation.

```
smallSample $ imap sample $ sqrtStrings 2
["2.00000000","1.50000000","1.41666666","1.41421568","1.41421356"]
```

Section 3: Solving mazes

In this section we will implement a simple maze solver. The maze is represented as a list of lists of cells, where every cell can be open, blocked, or contain a treasure.



A 3×3 maze; there is an obstacle at the middle and a treasure at bottom right and bottom left corners. The top left corner is at row and column 0, and the bottom right corner is at row and column 2.

The above maze is represented in Haskell as[3]:

```
data Maze = Maze {width :: Int, height :: Int, layout :: [[Cell]]} deriving Show
data Cell = Open | Blocked | Treasure deriving (Show, Eq)
maze :: Maze
maze
  = Maze
  { width = 3
  , height = 3
  , layout =
    [ [Open, Open, Open]
    , [Open, Blocked, Open]
    , [Treasure, Open, Treasure]
    ]
  }
```

[3]We can get ever more better asymptotic performance by using a `Map` instead of a list of lists, but as those will only be taught next week, we won't require it for this assignment.

- The function **`cellAt`** returns the cell at a given coordinate. If the coordinate is out of bounds, return `Nothing`.

```
data CellPosition =
  CellPosition {row :: Int, col :: Int} deriving (Show, Eq, Ord)
cellAt maze $ CellPosition {row=0,col=0}
Just Open
cellAt maze $ CellPosition {row=1,col=1}
Just Blocked
cellAt maze $ CellPosition {row=2,col=2}
Just Treasure
cellAt maze $ CellPosition {row=2,col=0}
Just Treasure
cellAt maze $ CellPosition {row=0,col=2}
Just Open
cellAt maze $ CellPosition {row=3,col=0}
Nothing
cellAt maze $ CellPosition {row=0,col=(-1)}
Nothing
```

- The function **`getAvailableMoves`** returns the cells which are reachable in a single move from a given cell. You can only move up, down, left, or right, and only if the target cell is not blocked. If the input is out of bounds, return `Left OutOfBounds`. If the input cell is blocked, return `Left InvalidCell`. Otherwise, return `Right [CellPosition]`.

```
getAvailableMoves maze $ CellPosition {row=0,col=0}
Right [CellPosition {row=0,col=1},CellPosition {row=1,col=0}]
getAvailableMoves maze $ CellPosition {row=0,col=1}
Right [CellPosition {row=0,col=0},CellPosition {row=0,col=2}]
getAvailableMoves maze $ CellPosition {row=1,col=1}
Left InvalidCell
getAvailableMoves maze $ CellPosition {row=(-1),col=42}
Left OutOfBounds
```

- With the basic functions out of the way, we can now implement the maze solver. The *required* functionality is `shortestPath`, which returns the shortest path from a source cell to a target cell. The shortest path **shouldn't** include the source or target cells.
  - If there is no path between the two coordinates, return `Left NoPath`.
  - If either the input or target coordinates are obstacles, return `Left InvalidCell`.
  - If either the input or target coordinates are out of bounds, return `Left OutOfBounds`[4].

[4]Usually, it's better design to have different error types for different functions, if a given function can't actually return all the error types! We've opted for a simpler solution in the assignment.

```
shortestPath maze CellPosition {row=0,col=1} CellPosition {row=2,col=2}
Right [CellPosition {row=0,col=2},CellPosition {row=2,col=1}]
shortestPath maze CellPosition {row=0,col=1} CellPosition {row=0,col=0}
Right []
shortestPath maze CellPosition {row=0,col=1} CellPosition {row=1,col=1}
Left InvalidCell
shortestPath maze CellPosition {row=0,col=1} CellPosition {row=3,col=2}
Left OutOfBounds
blockedMaze
  = Maze
  { width = 3
  , height = 3
  , layout =
    [ [Open, Open, Open]
    , [Blocked, Blocked, Blocked]
    , [Open, Open, Open]
    ]
  }
shortestPath blockedMaze CellPosition {row=0,col=0} CellPosition {row=2,col=2}
Left NoPath
```

Algorithmic guidance: the simplest way of finding the shortest path between two positions is to use the breadth-first search (BFS) algorithm. BFS requires two data structures: a queue to store the cells to be visited, and a set[5] to store the cells that have already been visited[6]. We've added the `Queue` data structure we saw in the lectures to the file, and also imported the `Set` data structure from the `containers` package. Useful functions for `Set`:

```
import qualified Data.Set as Set
setOf123 :: Set Int
setOf123 = Set.fromList [1,2,3]
iterativeSet :: Set String
iterativeSet = "foo" `Set.insert` ("bar" `Set.insert` ("moo" `Set.insert` Set.empty))
cellPosSet :: Set CellPosition
cellPosSet = Set.singleton CellPosition {row=0,col=0}
CellPosition {row=0,col=0} `Set.member` cellPosSet
True
CellPosition {row=0,col=1} `Set.member` cellPosSet
False
```

3.1 Bonus: Treasure hunt (15 points)

As a bonus, implement the function `treasureHunt`, which returns a path that collects **all the treasures** in the maze. Since this problem is NP-hard, we will not require an optimal solution. Like `shortestPath`, the function should return `Left` if the starting position is blocked or out of bounds, or if there is no path from the starting position to any treasure.

```
-- An example solution. This isn't the most optimal path, but that's OK.
treasureHunt maze CellPosition {row=0,col=0}
Right [CellPosition {row=0,col=1},CellPosition {row=0,col=2}
      ,CellPosition {row=1,col=2},CellPosition {row=2,col=2}
      ,CellPosition {row=2,col=1},CellPosition {row=2,col=0}]
```

[5]In the standard library, `Set`s (and `Map`s) are **abstract** data types based on balanced trees.
[6]The classic algorithm simply marks visited nodes in the same mutable structure used to represent the graph.