PF3 Homework Assignment

Winter 2025

This year, the homework assignment consists of 3 different problems that you may solve in any order during a longer period of time (slightly over 1 month). You will be given a "framework" which provides you with a basic space to develop your solutions, and which also serves as a simple preliminary test of whether your solutions meet the expectations.

Grading. The work is organized into several tasks. Completing all tasks may provide you a total 30% of the final score from the course. The bonus tasks count as additional points, but only up to the maximum score of the homework assignment (they are there to allow the student to cover up for other minor mistakes).

Please note that the tasks are not evaluated solely by correctness on the test cases. In fact, the supplied test cases in the testing examples provided are deliberately minimal and serve only as a guideline to flag completely incorrect solutions. Your code will also be tested on a more comprehensive set of tests, and additional factors such as code succinctness, clarity, and solution performance may affect the final score.

Solutions. Your solution should be filled into the places marked with `undefined` in the test framework. You may use any code style that you like, but please ensure that the code is easily understandable by humans when viewed as-is, without additional tools.

Your solution must be separated into 3 files for problems 1, 2 and 3, named `Olympics.hs`, `Evernight.hs` and `Pipeline.hs`, as provided in the testing framework, each defining the appropriate Haskell module that contains your solutions of the tasks in that Problem. E.g., `Olympics.hs` should look like this:

```
module Olympics where

stepanalysis n = ...

...
```

In the solution, tou may rely on any functions that are present in the `base`[1] package of GHC; in particular it is recommended to utilize suitable functions from `Prelude`[2] and modules such as `Data.List` to keep the code terse. You may also use code from other libraries (such as `extra`[3]) — in such cases, copypaste the source code of the functions that you want to use directly into your solutions and properly mark its origin with code comments. In many cases, you may help yourself a lot by defining various little helper functions that simplify the task, these are generally welcome and may contribute positively to your score.

Submission. Some time after the announcement of the homework, you will be provided with an on-line platform to evaluate your solution on blinded tests. This should give you additional feedback regarding the correctness and performance of your solutions.

Regardless of your use of the testing platform, your solution must be submitted in Moodle to be graded. The deadlines for submission will be also marked in Moodle.

Feedback. In case of questions, we recommend to ask for possible additional details and clarifications via Discord (see Moodle for link). In particular, if you are not sure what the solution should answer for a given input, you may post the input in Discord channel — we will try to give you an elaborated explanation or provide the expected output.

Please avoid posting any of your solution code in Discord and generally any other communication channels.

[1] https://hackage.haskell.org/package/base
[2] https://hackage.haskell.org/package/base-4.21.0.0/docs/Prelude.html
[3] https://hackage.haskell.org/package/extra

Problem 1

The Math Olympic Games 🏅

The Math Olympic Games have started! All numbers are competing against one another to prove which is the best number of them all. After many races, they arrive at the final winner. Or do they?

For the final race, the competition is to see who can reach the number **1** the fastest, while respecting the following rules:

1. Each **even number** is divided by 2.
2. Each **odd number** transforms itself into $3n + 1$ equivalent, where n is the original odd number.
3. These steps repeat until the number becomes 1.

Now *you* must help the numbers find out which one is the fastest by solving the following tasks:

Task 1.1 (`reaching1`)**.** *The numbers first need you to define a function `reaching1` that calculates how many operations it takes for a given number to reach* 1.

For example, evaluating `reaching1 5` should return `5` because it takes 5 steps as follows:

1. $3 \times 5 + 1 = 16$
2. $16/2 = 8$
3. $8/2 = 4$
4. $4/2 = 2$
5. $2/2 = 1$

Task 1.2 (Function `stepAnalysis`)**.** *After defining `reaching1`, the numbers are still curious! They want to see how they transform after each step. Define a function `stepAnalysis` that returns a list of* 2*-element tuples, where:*

- *the first element is the step number,*
- *the second element is the new value after the operation.*

For example:

```
stepAnalysis 5 == [(1,16), (2,8), (3,4), (4,2), (5,1)]
```

In short, each pair shows the step and the value after that step, until the value reaches 1.

Ranking ties with GCD

Even with all this information, the numbers still cannot decide who is "better", so they team up in *pairs of two.* For this competition, the goal is to see which pair has the *lowest GCD.*

Most of the numbers had no idea what this "GCD" was, so the referee decided to explain it in simple terms:

The **Greatest Common Divisor (GCD)** of two numbers a and b is the biggest number that divides both of them exactly, without leaving a remainder. This, we write $\gcd(a, b)$ as the largest number dividing both a and b.

For example, $\gcd(12, 8) = 4$ because 4 divides both 12 and 8 perfectly.

The GCD is computed using Euclidean algorithm. To compute GCD of 20 and 6, one computes:

$$\begin{aligned}\gcd(20, 6) &= \gcd(6,\ 20 \bmod 6)\\ &= \gcd(6, 2)\\ &= \gcd(2,\ 6 \bmod 2)\\ &= \gcd(2, 0)\\ &= 2\end{aligned}$$

The actual ranking of numbers is performed as follows:

1. First, the GCD is computed for every pair of competing numbers. The pairs with smaller gcd are considered better.
2. In case two or more pairs have the same GCD, the pair with the smallest sum of both numbers is considered better.

Task 1.3 (Final ranking)**.** *Write a function* `finalRanking` *which takes a list of pairs of positive integers, and returns them ordered from* first place *(best) to* last place *(worst) according to the rule above:*

```
finalRanking :: [(Int, Int)] -> [(Int, Int)]
```

If there are ties, the function should preserve the order from the input list (i.e., it can be viewed as a stable sort).

For example:

```
finalRanking [(20, 3), (15, 30), (6, 19)]
  == [(20,3), (6,19), (15,30)]
```

Battle of the best

A small group of numbers thought that the final ranking is still not sufficiently great, and decided to revolt. Calling themselves a 'cyclic group', they decided to choose the *Most Final Best* number using the following procedure:

1. All numbers from the final ranking form up a cycle (by standing in a circle). The number that was the first in the final ranking list takes the hold of a *token*.
   (E.g., if the numbers in the final ranking were `[5, 2, 3]`, number 5 gets the token.)
2. The number with a token uses the token to smash the number next to it (the closest one standing in the direction of the original list). The impact cancels the number out, and removes it from the cycle.
   (Continuing the example from above, number 5 cancels out number 2.)
3. The token is passed to the next number in line.
   (Now number 3 holds the token.)
4. The procedure repeats from start until there is only a single number left. That number is The Most Final Best Number.
   (In the above example, number 3 will immediately cancel out number 5, and becomes the best number.)

Task 1.4 (Cyclic rank)**.** *Write a function that simulates the above procedure on a list of numbers, returning the best one:*

```
cyclicBest :: [Int] -> Int
```

(Upon receiving an empty list, the function may simply crash or return an error.)

Battle of the fastest

The decision procedure naturally attracted the curiosity of bookmakers, who wanted to quickly predict which number would win. It turned out that simulating the number canceling-procedure with all numbers is elegant, but not fast enough!

Luckily, there is a faster method: Since the position of the winning number is uniquely determined by the count of the numbers in the list, we can calculate the result of the procedure without actually simulating it, using only integer computations.

Bonus task 1.5. *Implement a function:*

```
bestIndex :: Int -> Int
```

that takes the count of best numbers (length of the list) and calculates the position of The Most Final Best number without simulating the individual operations. The positions should be numbered as list indices*, i.e., if the first number in the list would win, the returned value is* `0`*.*

Your implementation of `bestIndex` should be compatible with `cyclicBest`; in particular the function below should give the same result as `cyclicBest`:

```
fasterCyclicBest :: [Int] -> Int
fasterCyclicBest xs = xs !! bestIndex (length xs)
```

Hint: Run `cyclicBest [1..n]` for several values of `n`, and compare the *binary representation* of each result with the binary representation of the corresponding `n`.

Problem 2

The Evernight Tree 🍃

Five explorers step onto a world that shouldn't exist. The trees hum with distant voices, the rivers dream in reverse, and the sky wears a crown of stars that never blink. She—the fifth explorer—vanishes. Weeks pass. Then, in a dreamscape, they find her again. She no longer answers to her old name. She whispers a new one: Evernight.

To understand what she became, the team models her fractured memory and shifting sense of time as a special tree, a variation of a binary tree. The nodes of the tree contain memories, and are labeled with either of two possible states of the given memory — the Known, and the Forgotten. Your mission is to manipulate this *Evernight tree* to help bring her memories back.

The memory in the Evernight world is not a given — the world desires void, and the ability to remember facts is a subject to a Disconnection Rule of the world's physics. Viewed in the team's binary tree model, the rule states:

No directly connected memories shall be remembered together.

In particular, the tree must not contain two adjacent Known nodes. Thus, all children of a Known node must be Forgotten.

Task 2.1 (Is this Evernight?)**.** *Given types:*

```
data NodeState = Known | Forgotten deriving (Show, Eq)
data Evernight a = Leaf | Node NodeState (Evernight a) a (Evernight a) deriving Show
```

...write a predicate function:

```
isEvernight :: Evernight a -> Bool
```

...that checks whether a given Evernight tree respects the Disconnection Rule.

An example valid Evernight tree (displayed nicely in fig. 2.1 on the left) may be encoded as follows:

```
demoTree :: Evernight Int
demoTree =
  Node Forgotten
    (Node Known
       (Node Forgotten Leaf 3 Leaf)
       2
       (Node Forgotten Leaf 4 Leaf))
    1
    (Node Forgotten
       (Node Known Leaf 6 Leaf)
       5
       (Node Known
          (Node Forgotten Leaf 8 Leaf)
          7
          (Node Forgotten Leaf 9 Leaf)))
```

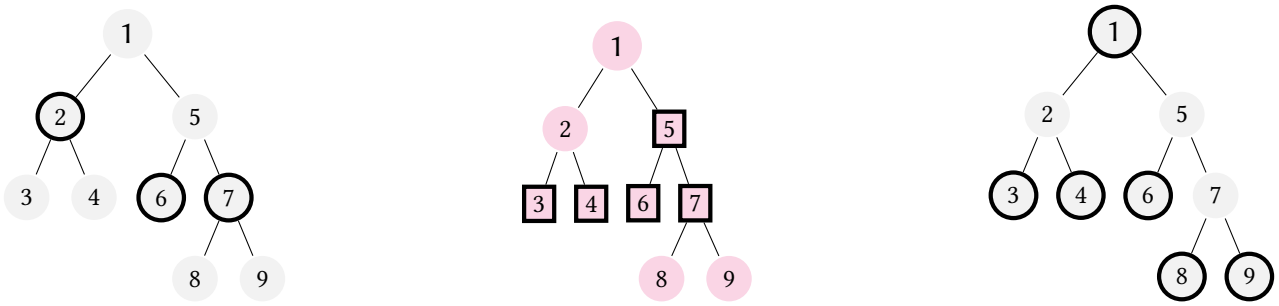

Figure 2.1: **On the left:** Example valid Evernight tree with integer memories, the encircled nodes are KNOWN, others are FORGOTTEN. Here, function `knownMemories` should return `[2,6,7]`. **Middle:** The same Evernight tree, with all 'unforgotten' nodes labeled with squares (all descendants of node 1 got "flipped" to the other state, and all descendants of node 5 got "flipped back" to the original state). Function `unforgottenMemories` should thus return `[3,4,6,5,7]`. **Right:** Optimal assignment of KNOWN nodes in the tree on the left (function `optimalEvernight` should return this).

Finding the memories

Task 2.2 (Memory scan)**.** *Write a function that extracts the contents of all KNOWN nodes into a list:*

```
knownMemories :: Evernight a -> [a]
```

The items in the list should follow the left-to-right traversal order; i.e., the item from any given node is preceded by all items from the left subtree of the node, and succeeded by all items from the right subtree.

In a dark corner of the Evernight world where double negation is real and nothing is impossible, forgetting that you forgot something brings back the memory! In the tree view, the memories are "viewed" from the root, and the whole path from root to the given memory determines whether the memory is known or not: If there is a single FORGOTTEN node on the whole path (including the root and the examined node itself), the memory is considered to be forgotten. If there are two, the memory is considered known (because the FORGOTTEN marks cancel out); if there is three, the memory is forgotten again; if four the memory is known again, etc. (See fig. 2.1 for an illustration.) After regarding all the confusing possibilities of this self-canceling forgetfulness, the team concludes to arm themselves with a helper function to resolve the situation for them, and moves on to spend no more time in this forsaken place.

Task 2.3 (Double-negated memory scan)**.** *Write a function:*

```
unforgottenMemories :: Evernight a -> [a]
```

that recovers the memories that are considered known (or "un-forgotten") as described above, in a left-to-right tree traversal order. In particular, `unforgottenMemories` *should return the memories from all nodes of the tree whose path from the root node (including the node itself) contains an* even number *of FORGOTTEN nodes. Note that the Disconnection Rule does not apply to such un-forgotten memories.*

The team, happily returning to a more traditional part of Evernight, aims to recover as much of the consciousness as possible. They notice that by whispering the right words, they may switch the Evernight tree nodes between the KNOWN and FORGOTTEN states as they please. (See fig. 2.1 for an example.) Unfortunately, it seems difficult to decide which nodes they should recover first: Recovering a KNOWN memory makes the universe immediately forget all neighboring memories!

Task 2.4 (Optimal recovery)**.** *Implement a function that changes the node states in a given Evernight tree in a way that maximizes the number of KNOWN nodes, while respecting the Disconnection Rule of the Evernight universe:*

```
optimalEvernight :: Evernight a -> (Int, Evernight a)
```

The function should return a tuple that contains the count of KNOWN nodes, together with the modified tree. If there are multiple optimal solutions, the function may return any of them.

Hint: A brute-force solution that generates all possible Evernight trees and picks the best one will "work", but it will also take exponential time, and is thus too slow for the amount of memories the team aims to recover. Instead, consider a dynamic-programming algorithm that works from the leaves to the root as follows:

- For a node, it computes the best possible subtree where this node (a "local root") is KNOWN, and the best possible subtree where the node is FORGOTTEN:

  - For leaves, this is trivial (both possibilities are the same).
  - For nodes with children, the algorithm recurses, and receives in total 4 possibilities of arranging the states in the children nodes. From these, it can easily reconstruct the best possibility where it is itself marked KNOWN (all children must be FORGOTTEN), and the best possibility where it is marked FORGOTTEN (the children are chosen either with KNOWN or FORGOTTEN in their roots, depending on which is best).

- Both computed possibilities are returned to the parent node. The parent is then able to choose the optimal combination from the best possibilities it received, and continues with the recursion.

Note that the Disconnection Rule implies that KNOWN nodes must have FORGOTTEN children, whereas FORGOTTEN nodes may have KNOWN and FORGOTTEN children. Additionally, when a FORGOTTEN node is choosing between the possible subtrees of equal "score" (same amount of KNOWN nodes), either subtree may be selected without affecting the parent — the validity of parent's label only depends on the FORGOTTEN label of the node, as the Rule does not "reach" the subtrees. In turn, there may be many different Evernight trees with the same score.

You may choose to materialize this algorithm via a helper function that returns the two possibilities, with a type signature such as:

```
optimalPossibilities :: Evernight a -> ((Int, Evernight a), (Int, Evernight a))
```

Shards of the Olympics

Suddenly, she remembers numbers — lots of them, striving to win a pointless competition! And she likes them, especially the greatest of values, hiding much entropy and inspiring to know even more. The team jumps up in excitement and arranges to recover the memories of the largest total of all numbers.

Task 2.5 (Olympic recovery)**.** *Write a function similar to* `optimalEvernight` *that, instead of maximizing the count of Known memories, attempts to maximize a total sum of the numbers in the Known memories. Note that as in the previous task, we aim to find a optimal configuration of an Evernight tree (satisfying the Disconnection Rule), but now the goal is to maximize the sum of all Known labels.*

```
maximizeEvernight :: Evernight Int -> (Int, Evernight Int)
```

Since the team is full of Haskell programmers, they are not happy at all having two different functions around doing almost the same thing. Some explorers start to whimper about "static checking" and "correctness by construction", others exclaim "this could break anytime" and "parseltongue!" The following two tasks are here to bring happiness to these explorers.

Task 2.6 (Functional olympic recovery)**.** *Write a generic function for optimizing the Evernight trees:*

```
genericOptimalEvernight ::
  (Num b, Ord b) => (a -> b) -> Evernight a -> (b, Evernight a)
```

The function allows one to convert the contents of the Evernight tree (of type `a`*) to a numeric type of score (type* `b`*, such as* `Int`*), and optimizes the score.*

For example, with the above at hand, one could implement a function that recovers the longest possible total amount of text from a memory of Strings as follows:

```
longestTextMemories :: Evernight String -> [String]
longestTextMemories = knownMemories . snd . genericOptimalEvernight length
```

To scrutinize your solution, you may also choose to also re-implement `optimalEvernight` and `maximizeEvernight` as one-liners that use `genericOptimalEvernight`.

Bonus task 2.7 (Strictly typed Evernight)**.** *Since invalid trees should never be admitted to the universe (or the program code), implement a data type* `ValidEvernight` *(possibly together with several helper data types) that may encode* only valid *Evernight trees. In particular, attempts at encoding Evernight trees that would break the Disconnection Rule must result in a type error. Implement the corresponding conversion function:*

```
validEvernight :: Evernight a -> Maybe (ValidEvernight a)
```

For additional bonus points, implement a function `validKnownMemories` *that works just like the* `knownMemories` *atop the* `ValidEvernight` *type.*

Hint: Use the following scheme of mutually recursive data types:

```
type ValidEvernight a = ... -- type alias for the type of the "root" node
data AnyEvernight a = ... -- three data constructors for Known, Forgotten and leaf nodes
data ForgottenEvernight a = ... -- two data constructors for Forgotten node and leaf
```

Problem 3

The Pipeline 🏭

Think of a computation as an assembly line in a factory: raw data enter through the input layer and pass through a cascade of machines, each transforming and relaying its output to the next. The entire network pulses with purpose, turning information into results, all according to the design of its builders. What does it manufacture? Perhaps simple sums, perhaps meaning, or perhaps it dreams in a language of its own, about worlds beyond our knowing.

Formally, we shall model the factory structure as a layered network of nodes:

- The nodes are partitioned into layers $L_{\text{in}}, L_0, L_1, \ldots, L_n$. The nodes in each layer are indexed with consecutive integers starting from 0. $L_0, L_1, \ldots, L_n$ are also referred to as internal layers.
- The nodes remember the indices of the nodes in the preceding layer that are used as their inputs. For example, a node in layer L_0 may remember index 3, which means that it processes the input in layer L_{in} on the index 3.

Figure 3.1 shows such a pipeline: Layer L_{in} contains input data (boxes), and layers L_0 to L_2 contain machines (circles). The data flow from L_{in} to L_2 along the direction of the arrows.

Types of pipelines

Each node from internal layers $(L_0, \ldots, L_n)$ represents a machine which carries an executable function. A node may have arity 0, 1, or 2 (at most two inputs), and stores references (indices) into the preceding layer:

```haskell
import Text.Show.Functions  -- enables printing out function types

type Layer a = [a]

data Machine a = Nullary a
               | Unary (a -> a) Int
               | Binary (a -> a -> a) Int Int
  deriving Show

data Pipeline a = Pipeline (Layer a) [Layer (Machine a)]
  deriving Show
```

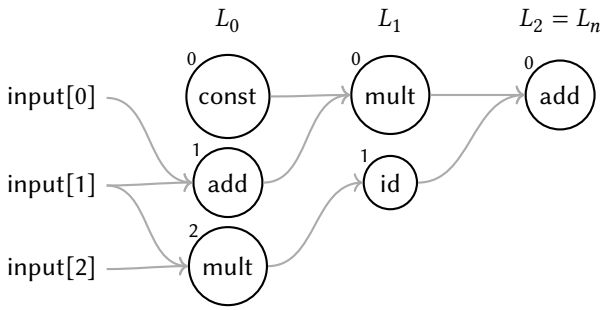

Figure 3.1: An example pipeline network. Indices of the nodes in layers are marked with the small number labels.

Here the outer list in `Pipeline` enumerates internal layers, each inner list enumerates the nodes within that layer and thus induces their indices $(0, 1, 2, \dots)$. The input layer L_{in} is an explicit list of values of type a stored in the `Pipeline`. For example, the pipeline in Figure 3.1 can be encoded as follows:

```
layerIn = [x0, x1, x2]

layer0 =
  [ Nullary 0
  , Binary (+) 0 1
  , Binary (*) 1 2
  ]

layer1 =
  [ Binary (*) 0 1
  , Unary id 2
  ]

layer2 =
  [ Binary (+) 0 1
  ]

internalLayers = [layer0, layer1, layer2]

pipeline = Pipeline layerIn internalLayers
```

Well-formed pipelines

We say that a list of internal layers is well-formed with respect to a non-negative bound m (the size of the preceding layer) iff every reference r appearing in the current layer satisfies $0 \leq r < m$. Intuitively, references must point only to existing nodes in the preceding layer. The list of internal layers may be empty (a degenerate pipeline that returns its inputs as outputs).

Task 3.1 (References used by a node/layer). *Implement functions*

```
machineRefs :: Machine a -> [Int]
layerRefs :: Layer (Machine a) -> [Int]
```

...that extract the references used by a single `Machine` and by a whole `Layer`. The result for a layer should be deduplicated and sorted in ascending order.

Task 3.2 (Checking the invariants layer-by-layer). *Implement a function*

```
checkLayers :: Int -> [Layer (Machine a)] -> Maybe [Layer (Machine a)]
```

...that checks well-formedness starting from the given bound (initially the size of L_{in}). It should return `Just` the original list when all references in each layer are strictly below the current bound (and, implicitly, non-negative), and `Nothing` otherwise. (On success, the bound for the recursive step becomes the size of the current layer.)

Task 3.3 (Building a pipeline with validation). *Implement a function*

```
constructPipeline :: Layer a -> [Layer (Machine a)] -> Maybe (Pipeline a)
```

...that checks the internal layers and, if valid, constructs the corresponding `Pipeline`. Otherwise return `Nothing`.

Evaluation of pipelines

Evaluation proceeds layer-by-layer, caching each layer's results for the subsequent layer.

Task 3.4 (Adapting inputs to a fixed-width pipeline). *Sometimes one wants to reuse a pipeline shape with a different list of inputs. Implement a function*

```
withInputs :: [a] -> a -> Pipeline a -> Pipeline a
```

...that replaces the input layer of a pipeline with the given list, trimming the list if it is too long, or padding it with the provided default value if it is too short, so that its length matches the input length of the original pipeline.

Task 3.5 (Evaluating a single machine)**.** *Implement a helper function*

```
evalMachine :: [a] -> Machine a -> a
```

...that, given the cached results (or inputs) of the previous layer, evaluates a single `Machine`*. Assume that all reference indices in the Machine are valid for the given input cache.*

Task 3.6 (Evaluating all layers)**.** *Implement a helper function*

```
evalLayers :: [a] -> [Layer (Machine a)] -> [a]
```

...that evaluates all internal layers left-to-right, using the output of each layer as the input cache for the next one, and finally returns the output layer's values. Assume that in each step, all reference indices used in every layer are valid for the given input cache.

Task 3.7 (Evaluating a whole pipeline)**.** *Implement a function*

```
eval :: Maybe (Pipeline a) -> [a]
```

...that evaluates a `Pipeline` *starting from its input layer. The input is supposed to arrive from* `constructPipeline`*, so the function may assume that all indices and references in the provided pipeline are valid. If there is no pipeline in the input, the function should return an empty list.*

For example, running:

```
eval $ constructPipeline [0,1,2] internalLayers
```

...should return `[2]`.