Homework Assignment 4

Functional and Logic Programming, 2025

Due date: Tuesday, June 17th, 2025 (17/06/2025)

Bureaucracy

- The staff member in charge of this assignment is Shmuel Hanoch (shmuel.hanoch@post.runi.ac.il).
- Submission is in pairs, but solo submission is also allowed. We very much suggest you pair-up, as solving these exercises with another person greatly enhances your learning (as well as being more fun!).
- To submit, create a zip file named `HW4_<id1>_<id2>.zip` where `<id1>` and `<id2>` are the submitters' IDs.
  - Or `HW4_<id>.zip` if submitting alone.
  - You do not need special permission to submit alone.
- The zip file should contain **two (2!), top-level files** (no folders!) named `HW4.hs` and `MultiSet.hs`!
  - The contents of these files will be explained later.
- 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 MultiSet.hs HW4.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 **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 for the bonus section!

General notes

- The instructions for this exercise are split between this file and `HW4.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 lookups, 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 entirely 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 number 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:
    $$0.7 \times \text{ExamGrade} + 0.3 \times \frac{\sum_{i=1}^{6} HW_i}{6} + \text{AttendanceBonus}$$
    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 the 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: Foldable Functions

In this section you will implement a few functions we saw for lists for `Foldable`, plus a few extra ones. Note that functions which can early exit—`elem`, `null`, etc.—should support it.

Here are example usages of the new functions:

```
getSum $ fold $ map Sum [1, 2, 3]
6
toList $ Just 4
[4]
toList $ Nothing
[]
-- Not part of HW4.hs.
-- You can take the implementation and instances from the lectures.
single a = Tree Empty a Empty
toList Tree (single 1) 2 (single 3)
[1,2,3]
maxBy length ["foo", "bar", "bazz"]
Just "bazz"
minBy length ["bar", "bazz"]
Just "bar"
```

Hint: The `Arg` type from the previous homework can also be useful here (it is already pre-imported from `Data.Semigroup`).
Tip: Since you already saw how to implement a few of these using `foldr`, it is **a very good exercise** to implement them using `foldMap`!

Section 2: Composing folds

In this section, we'll implement a small utility library to ease the usage of folds. Suppose you want to implement the function `average` for a list of numbers. We already have implementations of `sum` and `length`, so we can define something like

```
average :: Fractional a => [a] -> a
average xs = sum xs / fromIntegral (length xs)
```

This implementation is nice and elegant, but not the most efficient one: since `sum` and `length` are both implement using fold, we'll need two folds, which are two different passes on the list. We could solve it by writing a clever implementation which traverse the list only once, but then it would not be as readable and nice. Ideally, we would want folding operations to be more composable, just like regular functions, in a way that is also efficiently implemented.

- To achieve this, we'll define the data type

  ```
  data Fold a b c = Fold (b -> a -> b) b (b -> c)
  ```

  This type represent a left fold operation from a list of `a`'s to some `c`. It has a step function, an initial `b` b, and a "finalize" function that takes the accumulator of type `b` and create the actual result from it, of type `c`.

- implement `combine` and `combineWith`, as the way to combine simple Folds into more complex ones. Use them to implement 'averageF'.

- Finally, we also need to "execute" the fold operation. `runFold` will take a `Fold`, a foldable `t a`, and runs it as a **strict left fold** (i.e., `foldl'`).
- Implement some basic `Folds`, to be used as building blocks: `lengthF`, `sumF`, `nullF`, `toListF`, `findF` and `topKF`.
- Now implement the composed aggregates, `averageF` and (if you want as a bonus) varianceF.

Note: This is a particular case of a powerful technique in functional programming, the idea of **internal domain specific languages (DSLs)**. We separate the description of computations using a dedicated "language", and finally "compile" the language into an actual result.

Section 3: `Functor` functions

In this section you will implement a few functions on `Functor`s. Here are example usages of the new functions:

```
fmapToFst length ["foo", "bar"]
[(3,"foo"),(3,"bar")]
fmapToSnd length  Just "foo"
Just ("foo",3)
strengthenL 42 $ Right "foo"
Right (42,"foo")
strengthenR "x" [1, 2, 3]
[(1,"x"),(2,"x"),(3,"x")]
unzip $ Just (1,2)
(Just1,Just2)
coUnzip (Right [1,2,3] :: Either String [Int])
[Right 1,Right 2,Right 3]
coUnzip (Left "foo" :: Either String [Int])
[Left 'f',Left 'o',Left 'o']
```

Section 4: `MultiSet Foldable instances`

In this section we'll implement additional instances for `MultiSet` from exercise 4.

- Implement a `Foldable instance`. The `instance` should respect multiplicity: if an element appears n times, it should participate in the fold operation n times.

```
toList $ insert 2 $ insert 1 $ insert 2 mempty
[1,2,2]
-- Could be a different order, but the multiplicity should be
   respected.
```

- Sometimes, it's also useful to fold over the **unique** set of the elements in the multiset. Define an additional `Foldable` instance which will allow you to do so.
  - The instance should be defined outside the `MultiSet` module. Use the `FoldOccur` wrapper to avoid an orphan instance.
  - `instance`s defined outside the module can use only the public `MultiSet` API. The API from exercise 4 does not contain an efficient way to traverse each element once, so we will add one. Implement the new function `foldOccur`, and also use it in the unique instance of `Foldable` implementation.

```
toList $ FoldOccur $ insert 2 $ insert 1 $ insert 2 mempty
[1,2]
-- Or a different order, but every item appears exactly once.
```

- `MinToMax` **ensures** that the elements are iterated in **ascending** order

```
toList $ MinToMax $ insert 2 $ insert 1 $ insert 2 mempty
[1,2,2]
-- This exact order!
```

- `MaxToMin` **ensures** that the elements are iterated in **descending** order

```
toList $ MaxToMin $ insert 2 $ insert 1 $ insert 2 mempty
[2,2,1]
-- This exact order!
```

  - You may use the `sort` function from `Data.List` to implement both of the above **`instance`**s, or may define helper functions similar to `foldOccur` in `MultiSet.hs`.
  - You may add additional functions to the export list of `MultiSet.hs`, but since it is an **abstract data type**, you **must not** export its constructor!

Bonus: `Ziplists` (10 points)

Implement a `Semigroup` and `Monoid` **`instance`** for `ZipList`, such that `(<>)` is applied as a dot product.

```
map getsum $ getziplist $ ziplist (map sum [1, 2, 3]) <> ziplist (map sum
    [4, 5])
[5,7]
take 5 $ map getproduct $ getziplist $
    ziplist (map product [1..]) <> ziplist (map product [0..])
[0,2,6,12,20]
```

Note: remember the `Monoid` laws: `mempty <> a == a` and `a <> mempty == a` for all `a` (including infinite lists)! What would be the correct implementation of `mempty` for `ziplists`?

Bonus: `A complex aggregate: varianceF` (5 points)

Implement a `varianceF` Fold. Use the formula

$$E[X^2] - E[X]^2$$

and `combineWith`, `map` and alike as needed to accumulate multiple quantities at the same time and combining them.