Homework Assignment 4

Functional and Logic Programming, 2025

Due date: Tuesday, June 3rd, 2025 (03/06/2025)

Bureaucracy

- The staff member in charge of this assignment is Or Tzadik (or.tzadik@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: `MultiSet` Implementation

In this section, you will implement an abstract data type `MultiSet`, which is similar to a set but allows multiple occurrences of each element. Since this is an abstract data type, it resides in its own module, aptly named `MultiSet.hs`. While you can add helper functions to this module, including exported ones, you should of course never export the constructors of the data type itself!

- We will implement `MultiSet` using a `Set (Arg a Int)`, where `Arg a Int` pairs an element of type `a` with its count (multiplicity). The `Ord` and `Eq` instances for `Arg` compare only on the element (the first field), so the `Set` treats each element as a unique key regardless of its count.
- The standard `Set` API does not include a direct lookup-by-key function. Instead, `Data.Set` provides the function Set.lookupGE which returns the smallest element $\geq$ the given key.
- **Hint:** For internal use, you may wish to implement a function `lookupS` directly for the `MultiSet`:

```
lookupS :: Ord a => a -> Set.Set a -> Maybe a
```

  Be aware that an extra comparison is required to ensure the `lookupGE` found the right element.

- `insert`ing an element increases its count by 1 if it already exists in the multiset (or adds it with count 1 if not).
- `remove` removes one occurrence of an element. If that was the last occurrence, the element is removed entirely from the multiset. Removing an element that is not in the multiset has no effect (i.e., returns an equivalent multiset).
- You should also provide functions to convert between multisets and lists (including duplicates), i.e. `fromList` and `toList`.
- Regarding time complexity, `empty` should run in $O(1)$, `member`, `insert`, `remove` in $O(\log(n))$, `count` in $O(n)$, `toList` in $O(m)$ and `fromList` in $O(m \log(n))$ (where n is the number of distinct elements, and m is the total number of elements).
- Lastly, you should implement the following basic **`instance`**s:

  1. `Eq`, such that two multisets are equal if they contain the same elements with the same counts (regardless of order).
  2. `Show`, which should display the multiset contents in any order using {} braces. Duplicates should appear multiple times. For example:

```
insert 1 $ insert 1 $ insert 2 empty
-- Not necessarily in this order:
{1,1,2}
```

     You **may** use the functions intercalate and Set.toAscList to help with this implementation.

  3. `Semigroup`: defined as the union of two multisets. In the result, the count of each element should be the sum of its counts in the two inputs.

```
ms1 = fromList [1,1,2]
ms2 = fromList [1,3]
ms1 <> ms2
-- Again, not necessarily in this order:
{1,1,1,2,3}
```

  4. From the above definitions, you should be able to infer the correct implementation for `Monoid`.

Section 2: The `Jsonable` type class and instances

In this section, we will implement conversions to and from a JSON representation for various types. First, consider the JSON data type and the `Jsonable` type class[1]:

```
-- We define JString to avoid collisions with Jsonable [a] instance
newtype JString = JString String deriving (Show, Eq)

data Json
  = JsonNull
  | JsonBool Bool
  | JsonString JString
  | JsonInt Integer
  | JsonDouble Double
  | JsonArray [Json]
  | JsonObject (Map String Json)
  deriving (Show, Eq)

class Jsonable a where
  toJson   :: a -> Json
  -- Fails if the input JSON is a not a valid Json
  fromJson :: Json -> Maybe a
```

- The primary requirement for any `Jsonable` instance is that

  `fromJson (toJson x) == Just x`

  for any value `x`. In other words, converting a value to JSON and back should return an **equivalent** (as determined by `(==)`) value.

- You may not assume anything else about the values beyond the constraints of the **`instance`** itself. In particular, your JSON encoding should work for *any* value of the given type.

- Use simpler **`instance`**s to build more complex ones. For example, if a type contains a value of another `Jsonable` type, you should use `toJson` and `fromJson` of the inner type as helpers.

- Implement **`instance`**s of `Jsonable` for the following types: `Bool`, `JString`, `Int`, `Tuple`, `Triple`, `Maybe a`, `Either l r`, `[a]`, `Ord a => MultiSet a`, `Matrix a`, `SparseMatrix a`, and `Tree a`.

- To check yourself, you will need in many cases to explicitly use type annotations to prevent ambiguity:

```
fromJson (toJson (JString "abc", 3 :: Int)) :: Maybe (JString, Int)
Just (JString "abc",3)

-- Better yet (No need for type annotations):
thereAndBackAgain :: Jsonable a => a -> Maybe a
thereAndBackAgain = fromJson . toJson

thereAndBackAgain 1
Just 1

thereAndBackAgain $ Matrix [[1,2],[3,4]]
Just (Matrix [[1,2],[3,4]])
```

[1] In proper JSON, all numbers are doubles, but that's boring!

Section 3: The `Num` type class and `instances`

`Num` instances

The `Num` class must satisfy the following laws.

- The type `Bool` can be endowed with the structure of the finite field $\mathbb{F}_2$, the field with exactly two elements. Implement the `Num` instance for `Bool`.

```
True + True
False
True - False
True
9 :: Bool
False
42 :: Bool
True
```

- In the tutorials and previous assignments, we saw the `Expression` data type for arithmetic expressions with integers. In this section, we generalized `Expression` to support more types.

```
data Expression a =
    Iden String
  | Lit a -- Main change
  | Plus Expression Expression
  | Minus Expression Expression
  | Mult Expression Expression
  | Div Expression Expression
  | Signum Expression -- Needed for Num instance
  deriving (Eq)
```

  Implement a `Num` instance for `Expression`.

- Recall the **`newtype`** `Sum a` and **`newtype`** `Product a` from the lecture for defining alternative `Num a` **`instance`**s. We will apply the same idea to `Matrix a` and `SparseMatrix a`:

```
newtype MatrixSum a = MatrixSum {getMS :: Matrix a}
newtype MatrixMult a = MatrixMult {getMM :: Matrix a}
newtype SparseMatrixSum a = SparseMatrixSum {getSMS :: SparseMatrix a}
newtype SparseMatrixMult a = SparseMatrixMult {getSMM :: SparseMatrix a}
```

  Implement `Semigroup` for each of these **`newtype`**s

  - Matrix sums are component-wise addition.
  - Standard matrix multiplication. If the first matrix is $n \times m$ and the second is $m \times p$, the result is an $n \times p$ matrix.
    * You are encouraged to use transpose from `Data.List` (already imported).
  - In all instances, you may assume both matrices have compatible dimensions (if dimensions do not agree you may raise an error).
  - Don't forget to remove any zero entries after performing operations on `SparseMatrix`.

```
let ms = MatrixSum $ Matrix $ [[1,2],[3,4]]
ms <> ms
MatrixSum {getMS = Matrix [[2,4],[6,8]]}

let msb = MatrixSum $ Matrix $ [[True,False],[False,True]]
msb <> msb
MatrixSum {getMS = Matrix [[False,False],[False,False]]}

let mm1 = MatrixMult $ Matrix $ [[1,0],[2,1],[0,1]]
let mm2 = MatrixMult $ Matrix $ [[1,1,2],[0,2,1]]
mm1 <> mm2
MatrixMult {getMM = Matrix [[1,1,2],[2,4,5],[0,2,1]]}

(MatrixMult $ Matrix [[Lit 1]]) <> (MatrixMult $ Matrix [[Iden "x"]])
MatrixMult {getMM = Matrix [[Plus (Lit 0) (Mult (Lit 1) (Iden "x"))]]}

let sms1 =
  SparseMatrixSum $ SparseMatrix 3 3 $ Map.fromList [((0,0),1),((1,1),2)]
let sms2 =
  SparseMatrixSum $ SparseMatrix 3 3 $ Map.fromList [((0,0),3),((2,2),4)]
sms1 <> sms2
SparseMatrixSum {getSMS = SparseMatrix { rows = 3, cols = 3,
    entries = fromList [((0,0),4),((1,1),2),((2,2),4)]}}

let smm1 = SparseMatrixMult $ SparseMatrix 2 2 $
   Map.fromList [((0,0),1),((0,1),2),((1,0),3),((1,1),4)]
let smm2 = SparseMatrixMult $ SparseMatrix 2 2 $
   Map.fromList [((0,0),5),((0,1),6),((1,0),7),((1,1),8)]
smm1 <> smm2
SparseMatrixMult {getSMM = SparseMatrix { rows = 2, cols = 2,
    entries = fromList [((0,0),19),((0,1),22),((1,0),43),((1,1),50)]}}
```

General functions

- Given coefficients $[a_0, a_1, \ldots, a_k]$ for a polynomial $a_0 + a_1 x + \cdots + a_k x^k$, evaluate it at x. Since `Num` has no concept of division, you may assume the exponent is a non-negative integer. You may use the `(^)` operator for exponentiation, since it works for any `Num` base.

```
evalPoly [1,2,3] 2  -- (1*2^0 + 2*2^1 + 3*2^2)
17
evalPoly [] 5
0
evalPoly [1,0,0,3] 2 -- (1*2^0 + 3*2^3)
25
evalPoly [True, False] True
True
evalPoly [False] True
False
evalPoly [Lit 2, Lit 3] (Lit 5)
Plus (Mult (Lit 2) (Lit 1)) (Plus (Mult (Lit 3) (Mult (Lit 5) (Lit 1))) (Lit 0))
```

- In many real-world problems (transportation, social networks, chemistry), we want to count the number of distinct walks of a given length between two nodes. Consider the directed graph $G = (V, E)$ representing four airports and the direct flights between them:

$$V = \{A, B, C, D\}, \quad \text{(airport identifiers)}$$
$$E = \{\, A \to B,\ A \to C,\ B \to C,\ B \to D,\ C \to D\} \quad \text{(direct flights)}.$$

  We encode G as an adjacency matrix of 0s and 1s (rows and columns ordered A, B, C, D):

```
let adj =
      [ [0,1,1,0]   -- A->B, A->C
      , [0,0,1,1]   -- B->C, B->D
      , [0,0,0,1]   -- C->D
      , [0,0,0,0]   -- D has no outgoing flights
      ]
```

Given an adjacency matrix m of a directed graph, multiplying the matrix by itself k times returns a matrix where the entry $(m^k)_{ij}$ counts the number of distinct paths of length k from node i to node j.

Implement the function `pathsOfLengthK k i j` that returns the number of distinct paths of length k from node i to node j.

```
pathsOfLengthK 1 1 2 adj
1
pathsOfLengthK 2 0 3 adj
2
```

- Lastly, implement the function `hasPath` that checks if there is a path (of **any** length) from node i to node j in the graph represented by the adjacency matrix m.

```
hasPath 0 3 adj
True
hasPath 3 0 adj
False
```

Section 4: Expression Simplification and Inlining

In this section, we will add functionality to simplify `Expression`s.

- Write a function `simplify` that recursively simplifies an expression according to algebraic rules. For each operation, consider simplifying its sub-expressions first, then apply these simplifications:
  - $e + 0 = e$ and $0 + e = e$.
  - $e - 0 = e$.
  - $e * 1 = e$ and $1 * e = e$.
  - $e/1 = e$.
  - If both operands of an arithmetic operation are literals, perform the operation and replace with a single `Lit` result (for division, only do this when the divisor is non-zero).
  - `signum (Lit n)` = `Lit (signum n)`.
  - `signum`$(a * b)$ = `signum` a `*` `signum` b.
  - `signum`(a/b) = `signum` a `/` `signum` b.

  Make sure your `simplify` function applies these rules wherever possible, and leaves the expression unchanged when no rule applies. For example:

```
simplify $ Plus (Lit 0) (Iden "x")
Iden "x"

simplify $ Minus (Iden "y") (Lit 0)
Iden "y"

simplify $ Mult (Lit 1) (Plus (Lit 3) (Iden "z"))
Plus (Lit 3) (Iden "z")

simplify (Div (Lit 10) (Lit 5))
Lit 2
simplify (Div (Lit 10) (Lit 0))
Div (Lit 10) (Lit 0)

simplify (Signum (Lit (-7)))
Lit (-1)
simplify $ Signum (Mult (Lit (-3)) (Div (Lit (-5)) (Iden "w")))
Mult (Lit (-1)) (Div (Lit (-1)) (Signum (Iden "w")))
```

Note that non-constant subexpressions remain in symbolic form. In some cases, when new `Expression`s are introduced, an additional call to `simplify` may be required.

Bonus (20 points): `inlineExpressions`

Implement a function `inlineExpressions` which should works in two phases:

1. *Simplify phase:* apply your `simplify` function to every expression in the input list.
2. *Inlining phase:* traverse the list of simplified (`Expression`, `String`) pairs from first to last, and in each expression replace any sub-expression that exactly matches a previously seen expression by its associated name. Perform this replacement in a top-down, left-to-right order.
   - The replacements should be done only with previously introduced names for replacement. In other words, for each pair $(e_i, name_i)$ in the input list (from first to last): replace any sub-expression of e_i that exactly matches some e_j from an earlier pair (with $j < i$) by `Iden` $name_j$. Return the list of new pairs.

```
inlineExpressions [(Plus (Lit 2) (Lit 3),"a")]
[(Lit 5,"a")]

let x = Iden "x"; y = Iden "y"; z = Iden "z"; w = Iden "w";

inlineExpressions [(Plus x (Lit 0),"e1"),(Mult (Plus x (Lit 0)) (Lit 2),"e2")]
[(Iden "x","e1"),(Mult (Iden "e1") (Lit 2),"e2")]

inlineExpressions [(Plus (Iden "x") (Iden "y"),"p"),(Plus (Iden "z") (Iden "w"),"q"),
    (Mult (Plus (Iden "x") (Iden "y")) (Plus (Iden "z") (Iden "w")),"r")]
[(Plus (Iden "x") (Iden "y"),"p"),(Plus (Iden "z") (Iden "w"),"q"),
     (Mult (Iden "p") (Iden "q"),"r")]
```

Here, p is inlined into the definition of r (replacing $x + y$), and q is inlined into r as well (replacing $z + w$).