# Homework Assignment 2

Functional and Logic Programming, 2025

Due date: Tuesday, May 6th, 2025 (06/05/2025)

## 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 `HW2_<id1>_<id2>.zip` where `<id1>` and `<id2>` are the submitters IDs.
  - Or `HW2_<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 `HW2.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 HW2.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 `HW2.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!
- **An exception in this assignment**: the second section—on the `Expr` data type—is probably more involved than the others. This was done to match the order of the tutorials. It is nonetheless solvable without any functionality from the latter sections.
- Try to write elegant code, as taught in class. Use point-free style, $\eta$-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: `Maybe` and `Either` utility functions

This section includes a bunch of simple utility functions for `Maybe` and `Either`. Most of these should be self-evident from the signature alone, but an example usage for each function is detailed below. Hints:

1. Do not confuse `mapMaybe` and `mapEither` with `maybeMap` and `eitherMap` taught in class. For ease of use, these functions have already been implemented for you in the `HW2.hs` file.
2. Most functions (or at least, most *patterns*) should be a single line!
3. Most functions should be obvious from the signature alone, but we've added a few examples.

### 1.1 `Maybe` functions

```
fromMaybe 1 Nothing
1
fromMaybe undefined (Just 2)
2
maybe 1 undefined Nothing
1
maybe undefined length (Just "foo")
3
maybeHead [1,2,3]
Just 1
maybeHead []
Nothing
maybeLast [1,2,3]
Just 3
maybeLast []
Nothing
maybeMaximum [1,2,3]
Just 3
maybeMaximum []
Nothing
maybeMinimum [1,2,3]
Just 1
maybeMinimum []
Nothing
filterMaybe even $ Just 2
Just 2
filterMaybe even $ Just 3
Nothing
sumMaybe (Just 1) (Just 2)
Just 3
sumMaybe (Just 1) Nothing
Nothing
liftMaybe2 (*) (Just 2) (Just 3)
Just 6
liftMaybe2 undefined Nothing (Just 3)
Nothing
mapMaybe (\ e -> if even e then Just $ e * 2 else Nothing) [1,2,3]
[4]
catMaybes [Just 1, Just 2, Nothing, Just 3]
[1,2,3]
```

### 1.2 Either functions

```
fromEither undefined (Right 1)
1
fromEither "foo" (Left 42)
"foo"
catEithers [Right 10, Right 20]
Right [10, 20]
-- If there are any Lefts, returns the first one encountered
catEithers [Right 10, Left "foo", Right 20, Left "bar"]
Left "foo"liftEither2
(*) (Right 2) (Right 3)
mapEither (\x -> if x > 0 then Right $ x * 10 else Left $ x + 5) [1, 2, 3]
Right [10,20,30]
-- Returns the first Left
mapEither (\x -> if x > 0 then Right $ x * 10 else Left $ x + 5) [1, -1, 2, -2]
Left 4
partitionEithers [Right "foo", Left 42, Left 54, Right "bar"]
([42,54],["foo","bar"])
liftEither2 (*) (Right 2) (Right 3)
Right 6
liftEither2 undefined (Right 2) (Left "foo")
Left "foo"
```

## Section 2: Expression trees

In this section, we will revisit the `Expr` data type you saw in the tutorials.

```
data Expr = Iden String | Lit Int
          | Plus Expr Expr | Minus Expr Expr | Mul Expr Expr | Div Expr Expr
          -- This allows for *simple* string conversion and equality checks. We will
          -- learn more about it later in the course.
          deriving (Show, Eq)
```

- A very useful function is being able to convert an expression to a pretty[1] `String`, e.g., for debugging. To avoid ambiguity, we will **always** add parentheses around **sub**-expressions.

  ```
  exprToString $ Lit 1 `Plus` Lit 2
  "1 + 2"
  exprToString $ Iden "x" `Plus` (Lit 2 `Div` Lit 3)
  "x + (2 / 3)"
  ```

- In the tutorial, we implemented the function `evaluate`. The function `partialEvaluate` evaluates an expression **as much as possible**, i.e., it replaces `Idens` if possible, and any sub-expression whose operands could be simplified to `Lits`. Since we don't fail for missing variables, we can replace the `Either` return type with a `Maybe` in case of division by zero.

[1] You can already convert `Expr` to a `String` using the `show` function, or just evaluating an `Expr` in `GHCi`, but it would show as an `Expr` tree, as you can see in the examples below, which isn't as pretty.

```
partialEvaluate [("x", 42)] $
  (Lit 1 `Plus` Iden "x") `Mul` (Iden "y" `Minus` Lit 2)
Just (Mul (Lit 43) (Minus (Iden "y") (Lit 2)))
-- Division by 0 is always bad, even if we don't know the numerator value!
partialEvaluate [("x", 42)] $
  (Lit 1 `Plus` Iden "x") `Mul` (Iden "y" `Div` Lit 0)
Nothing
```

## 2.1 Bonus (25 points)

- The function `exprToString'` is similar to `exprToString`, but also takes into account operator **precedence and associativity**, to avoid unnecessary parentheses. In other words, it will only use parentheses when **necessary**.

```
exprToString' $ (Iden "x" `Plus` Lit 42 `Plus`  Iden "y") `Mul` 2 `Plus` 3
(x + 42 + y) * 2 + 3
exprToString' $ Lit 1 `Minus` (Lit 2 `Minus` Lit 3)
1 - (2 - 3)
exprToString' $ (Lit 1 `Minus` Lit 2) `Minus` Lit 3
1 - 2 - 3
```

  Hints:

  - There are two basic reasons to add parentheses to a sub-expression
    1. Mixing associative operators of different precedence. For example, $x + y * 2$ is the same as $x + (y * 2)$ (with the latter being redundant), but different from $(x + y) * 2$ (therefore, the parentheses are required). If all operators are associative—that is, $+$ and $*$—**and of the same precedence**, then you don't need parentheses. Likewise, you don't parentheses around operators of a stronger precedence (as in $x + y * 2$ is correct, but $x + (y * 2)$ is redundant).
    2. Non-associative operators, i.e., $-$ and $/$. For example, $x - y - z$ is the same as $(x - y) - z$ (redundant) but not the same as $x - (y - z)$ (not redundant). Likewise, $x/y * z$ is the same as $(x/y) * z$ and different from $x/(y * z)$.
  - Therefore, To know if you need parentheses around a sub-expression, you need to take into account both the **operator**, and the **precedence** of the sub-expressions.
  - Define a helper function which both returns the sub-expression's `String` *and* the **lowest** (or weakest) level of any **unparenthesised** operator, to know if you need parentheses or not. For example, if the sub-expression is $x + y * 2$, it has the same precedence as $+$, but $(x + y) * 2$ has the same precedence as $*$.
  - You only need **3** possible precedences for a given `Expr`.

## 2.2 Additional functionality

Even though we only defined 4 operators as **constructors**, we can actually define additional operators using functions! For example, we can double the value of an expression:

```
doubleExpr :: Expr -> Expr
doubleExpr = Mul $ Lit 2
```

Implement the functions `negateExpr`, `powerExpr`, and `modExpr` for the `Expr` data type.

```
partialEvaluate [] $ negateExpr $ Lit 20 `Minus` Lit 62
Just (Lit 42)
partialEvaluate [] $ Lit 42 `powerExpr` Lit 0
Just (Lit 1)
-- For simplicity, we also define 0^0 to be 1.
partialEvaluate [] $ Lit 0 `powerExpr` Lit 0
Just (Lit 1)
partialEvaluate [("x", 2)] $ (Iden "x" `Plus` Lit 4) `powerExpr` 3
Just (Lit 216)
-- If n < 0, the expression should be equivalent to 0.
partialEvaluate [] $ (Lit 2 `Plus` (Lit 3) `powerExpr` (-1)
Just (Lit 0)
-- But division by 0 is still bad!
partialEvaluate [] $ (Lit 2 `Div` Lit 0) `powerExpr` (-1)
Nothing
partialEvaluate [] $ (Lit 2 `Div` Lit 0) `powerExpr` 0
Nothing
partialEvaluate [("x", 2),("y", 3)] $ Lit 42 `modExpr` (Iden "x" `Plus` Iden "y")
Just (Lit 2)
Lit 42 `modExpr` Lit 0
Nothing
```

# Section 3: Lists

Unless impossible or otherwise noted, all list functions should also work on infinite lists. For the most part, this shouldn't be an issue, so long as you take care to pattern match correctly, and not evaluate the entire list unnecessarily. The easiest way to create an infinite list is to use the `(..)` operator[2] without an upper bound.

```
positives = [1..]
negatives = [-1,-2..]
take 10 positives
[1,2,3,4,5,6,7,8,9,10]
take 10 negatives
[-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
```

## 3.1 Zips

When working with lists, `zip`s combine elements from both lists, one element at a time, as if we are **zipping** both lists together. For example:

```
zip [1, 2, 3] "foo"
[(1,'f'),(2,'o'),(3,'o')]
zipWith (+) [1, 2, 3] [4, 5, 6]
[5, 7, 9]
```

[2]Do not confuse with the `(.)` operator.

Usually, when one list is shorter than the other, we simply stop early.

```
zipWith (+) [1, 2] [4, 5 ..]
[5, 7]
```

If we're not interested in stopping early, we have two valid options:

1. Providing a default value.
2. Failing.

```
zipWithDefault 0 'a' [1,2,3] "foobar"
[(1,f),(2,o),(3,o),(0,b),(0,a),(0,r)]
zipWithDefault 0 'a' [1..6] "foo"
[(1,f),(2,o),(3,o),(4,a),(5,a),(6,a)]
take 10 $ zipWithDefault 1 'a' [1..] "foo"
[(1,f),(2,o),(3,o),(4,a),(5,a),(6,a),(7,a),(8,a),(9,a),(10,a)]

zipEither [1, 2] "foobar"
Left ErrorFirst
zipEither [1..] "foobar"
Left ErrorSecond
zipEither [1, 2, 3] "foo"
Right [(1,f),(2,o),(3,o)]
```

`unzip` is the reverse operation of zipping.

```
unzip [(1, 2), (3, 4)]
([1, 3], [2, 4])
unzipFst [(1, 2), (3, 4)]
[1, 3]
unzipSnd [(1, 2), (3, 4)]
[2, 4]
```

Of course, zipping isn't the only way we can combine two lists. We can also use Cartesian products.

```
cartesianWith (*) [10, 20] [3, 4, 5]
-- Notice the order!
[30,40,50,60,80,100,120]
```

Note: the `zip` function is one of the very few places where using tuples is the right approach in Haskell!

## 3.2 Lists functions

In this section, we will implement a few list and `String`[3] functions.

```
take 2 [1, 2, 3]
[1,2]
take 4 [1..]

-- snoc is the opposite of cons!
snoc [2,3] 1
[2,3,1]
take 5 $ snoc [1..] 0
[1,2,3,4]

drop 2 [1, 2, 3]
[3]

takeWhile (< 5) [1..]
[1,2,3,4]
take 5 $ dropWhile (< 5) [1..]
[5,6,7,8,9]

slice 2 5 [1..]
[3,4,5]
take 5 $ takeEvery 3 [1..]
[3,6,9,12,15]
take 5 $ dropEvery 3 [1..]
[1,2,4,5,7]

-- nub Only removes consecutive duplicates
nub [1,1,2,3,2,4]
[1,2,3,2,4]
take 5 $ nub [1..]
[1,2,3,4,5]
infiniteRepeats = [tripleX | x <- [1..], tripleX <- [x,x,x]]
take 15 infiniteRepeats
[1,1,1,2,2,2,3,3,3,4,4,4,5,5,5]
take 15 $ nub infiniteRepeats
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]

-- uniq removes all duplicates, consecutive or not
uniq [1,1,2,3,2,4]
[1,2,3,4]
```

[3]Which, as you may recall, is just `[Char]`

Edges cases:

```
take 4 [1, 2, 3]
[1,2,3]
take 0 undefined
[]
take (-1) undefined
[]

takeWhile (< 1) [1..]
[]

drop 4 [1, 2, 3]
[]
drop 0 [1, 2, 3]
[1,2,3]
drop (-1) [1, 2, 3]
[1,2,3]

take 5 $ dropWhile (< 1) [1..]
[1,2,3,5]
-- Would never halt!
take 5 $ dropWhile (> 0) [1..]

slice 3 3 undefined
[]
slice 5 2 undefined
[]
slice (-2 5) [1..]
[1,2,3,4,5]

takeEvery 4 [1,2,3]
[]
takeEvery 1 [1,2,3]
[1,2,3]
takeEvery 0 [1,2,3]
[1,2,3]
takeEvery (-1) [1,2,3]
[1,2,3]

dropEvery 4 [1,2,3]
[1,2,3]
dropEvery 1 undefined
[]
dropEvery 0 undefined
[]
dropEvery (-1) undefined
[]

take 10 [1,1..]
[1,1,1,1,1,1,1,1,1,1]
take 2 $ nub [1,1..] -- Would never halt!

uniq [1..] -- Would never halt!
```

## 3.3 Base64 encoding

The functions `toBase64` and `fromBase64` convert integers to and from Base64 strings. We will use the basic **RFC 4648** for this[4], with an optional leading `'-'` sign for negative numbers. For ease of use, the numbers 0–63 are mapped to their respective index in the following `String`:

```
base64Chars :: [Char]
base64Chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
```

Since `fromBase64` can fail (e.g., `fromBase64 "!"`), its return type is `Maybe Integer`.

```
toBase64 0
"A"
toBase64 509701384549
"Haskell"
fromBase64 "Haskell"
Just 509701384549
toBase64 $ -509701384549
"-Haskell"
fromBase64 "-Haskell"
Just (-509701384549)
fromBase64 "--Haskell"
Just (509701384549)
fromBase64 "Haskell?"
Nothing
```

Tip: `fromBase64 . toBase64` is equivalent to `Just`!

[4] Do not worry about the padding mentioned in the table. As we convert our integers directly to strings, we do not need it.