MP1 - Basic Haskell

Objectives

We will be using Haskell throughout the course to implement other programming languages. That being said, the focus of this course is *not* on Haskell, but rather on studying programming languages in general. You need to understand basic Haskell before we can proceed with the rest of the course though; this MP will test that understanding.

Goals

- Part 1:
  - Write recursive functions and definitions.
  - Implement some set-theoretic functionality using Haskell lists.
  - Use higher-order functions to compactly represent common code patterns.
- Part 2: Use and write Algebraic Data Types (ADTs) with some associated operators.
  - Manipulate linked lists and simple arithmetic expressions defined using ADTs
  - Define a binary tree ADT and implement a function on binary trees.
  - Define a constant value ADT for a simple programming language and a higher order function to manipulate those values.

Useful Reading

If you are stuck on some problems, perhaps it's time to read some of Learn You a Haskell for Great Good. I would recommend reading the whole book eventually, but if you're crunched for time Chapter 3 on Types and Typeclasses, Chapter 4 on Syntax in Functions, and Chapter 8 on Making Your Own Types and Typeclases seem the most relevant. If you're still stuck on recursion problems, check out Chapter 5 on Recursion.

Getting Started

Relevant Files

Parts 1 and 2 of this MP each have their own skeleton Stack project. In each of these projects, the file `src/Lib.hs` contains all the relevant code for that part. The first line of the file, `module Lib where`, says that what follows (the rest of the file in this case) belongs to the `Lib` module. Haskell has a module system which allows for extensive code re-use. Saying something like `import Data.List` imports the `List` module, for example.

Running Code

To run your code, start GHCi with `stack ghci`:

```
$ stack ghci
 - more output -
Ok, modules loaded: Lib.
*Lib>
```

Testing Your Code

You can run the test-suite as you fill in the solutions for Part 1.

The following ensures the file will type-check, but the corresponding `myFunction` test will fail.

```
myFunction = undefined
```

In Part 2, you must supply the Algebraic Data Type declarations at the end of the problem-set correctly in order to compile the tests. If you are having trouble with one declaration in particular, use the interpreter to define and debug each ADT individually.

In each part, the test suite is compiled and executed by running `stack test` in the project directory. `recursion` and `adt` for parts 1 and 2 respectively.

```
$ stack test
```

It will tell you which test-suites you pass, fail, and have exceptions on. To see an individual test-suite (so you can run the tests yourself by hand to see where the failure happens), look in the file `test/Spec.hs`.

I Can't Do Problem X

You can ask for help on Discord or in office hours. If you're really stuck and don't have time to fix it, you *must* still put in the type declaration and define it as `undefined` so that it still compiles with our autograder. Not doing so will result in loss of extra points from your total score. Remember that course policy is that code which doesn't compile receives a zero.

For example, if you cannot complete problem `app :: [a] -> [a] -> [a]`, make sure you put the following code into your submission:

```
app :: [a] -> [a] -> [a]
app = undefined
```

Problems

Builtins: In general you cannot use the Haskell built-in functions. Especially if we say that a function "should behave exactly like the Haskell built-

in", *you cannot use that Haskell built-in.* **If you modify or remove the following two import statements your submission will not be scored!**

```
import qualified Prelude as P
import Prelude hiding ( take, drop, ... etc ...
```

Pattern-matching: You should try to use pattern-matching whenever possible (it's more efficient and easier to read). If you are using the functions `fst :: (a,b) -> a` or `snd :: (a,b) -> b` (for tuples) `head :: [a] -> a` or `tail :: [a] -> [a]` (for lists) chances are you can do it with pattern matching. We will take off points if you use built-ins when it's possible to pattern-match.

Helpers: You may write your own helper functions (inside or outside a `where` clause). All the restrictions about allowable code (no built-ins/using pattern matching) still apply to your helper functions.

Part 1: Recursion and Higher-Order Functions

include your definitions for the following problems in `mp1a-recursion/src/Lib.hs`

Recursion

For these problems, you *may not* use higher-order functions. Instead, you should use recursion to implement the stated functionality. If you do use higher-order functions (or do not use recursion), you will receive no points.

mytake

Write a function `mytake :: Int -> [a] -> [a]` which takes the first `n` elements of a list, or the whole list if there are not `n` elements. It should behave exactly like the Haskell built-in `take :: Int -> [a] -> [a]`.

```
*Main> mytake 4 [2,4,56]
[2,4,56]
*Main> mytake 3 []
[]
*Main> mytake 1 ["hello", "world"]
["hello"]
*Main> mytake (-3) [1,2,3]
[]
```

mydrop

Write a function `mydrop :: Int -> [a] -> [a]` which drops the first `n` elements of a list, or the whole list if there are not `n` elements. It should behave exactly like the Haskell built-in `drop :: Int -> [a] -> [a]`.

```
*Main> mydrop 3 [2,4,56,7]
[7]
*Main> mydrop 3 []
[]
*Main> mydrop 1 ["hello", "world"]
["world"]
*Main> mydrop (-3) [1,2,3]
[1,2,3]
```

rev

Write a function `rev :: [a] -> [a]` which reverses the input list. To get credit, your solution must run in linear time. If you use the `(++)` list append operator, chances are your solution is running in quadratic time. This function should behave exactly like the Haskell built-in `reverse :: [a] -> [a]`.

```
*Main> rev [1,2,3]
[3,2,1]
*Main> rev []
[]
*Main> rev ["hello", "world"]
["world","hello"]
```

app

Write a function `app :: [a] -> [a] -> [a]` which appends two lists. This function should behave like the Haskell built-in `(++) :: [a] -> [a] -> [a]`, and should run in linear time (in the size of the first list).

```
*Main> app [] [1,2,3]
[1,2,3]
*Main> app [1,2,3] []
[1,2,3]
*Main> app [4,5] [1,2,3]
[4,5,1,2,3]
*Main> app ["hello", "world"] ["and", "goodbye"]
["hello","world","and","goodbye"]
*Main> app "hello" "world"
"helloworld"
```

inclist

Write a function `inclist :: Num a => [a] -> [a]` which adds `1` to each element of the input list.

```
*Main> inclist [1,2,3,4]
[2,3,4,5]
*Main> inclist [-2,4,5,1]
```

```
[-1,5,6,2]
*Main> inclist []
[]
*Main> inclist [2.3, 4.5, 7.6]
[3.3,5.5,8.6]
```

sumlist

Write a function `sumlist :: Num a => [a] -> a` which adds all the elements of the input list.

```
*Main> sumlist []
0
*Main> sumlist [1,2,3]
6
*Main> sumlist [-3,2,5]
4
*Main> sumlist [3.3,2.8,-1.2]
4.8999999999999995
```

myzip

Write a function `myzip :: [a] -> [b] -> [(a,b)]` which zips up the elements of two lists. The resulting list should be the same length as the shorter of the two input lists.

```
*Main> myzip [1,2,3] []
[]
*Main> myzip [] [1,2,3]
[]
*Main> myzip [1,2,3] ["hello", "world"]
[(1,"hello"), (2,"world")]
```

addpairs

Write a function `addpairs :: (Num a) => [a] -> [a] -> [a]` which zips up two lists using the addition operator `(+)`. You *must use* your function `myzip`.

```
*Main> addpairs [1,2,3] []
[]
*Main> addpairs [1,2,3] [4,5,6]
[5,7,9]
*Main> addpairs [1.2,3.4] [-1.2,8.9,7.6]
[0.0,12.3]
```

ones

Write a (constant) function `ones :: [Integer]` which produces an infinite list of the integer `1`.

```
*Main> take 15 ones
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
*Main> take 0 ones
[]
```

nats

Write a (constant) function `nats :: [Integer]` which produces an infinite list of all the natural numbers starting at `0`. It is OK for this function if you do not use recursion.

```
*Main> take 15 nats
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14]
*Main> take 0 nats
[]
```

fib

Write a (constant) function `fib :: [Integer]` which produces an infinite list of the Fibonacci series starting with numbers `0` and `1`. You can (and should) use your `addpairs` function here. This is the one place in the assignment where it really makes sense to use `tail :: [a] -> [a]`.

```
*Main> take 10 fib
[0,1,1,2,3,5,8,13,21,34]
*Main> take 0 fib
[]
```

Set Theory

We will be using Haskell lists to represent abstract mathematical sets. Recall that in a set there are no duplicates, which means that our lists should have that property as well. To make this simpler, we'll be storing *sorted* lists, which means we can only create sets of things that are orderable (notice the `Ord a` *type constraint* in the type declarations).

As long as your set-theoretic interface functions do not create duplicates and always return sorted lists, then all of our nice set-theoretic properties will hold. You may assume that the input sets (Haskell lists) to these functions will also be sorted and will not contain duplicates.

You may use higher-order functions or recursion as you see fit in this section.

add

Write a function `add :: Ord a => a -> [a] -> [a]` which will add an element to the set. Remember that it must add it in the correct place to ensure that the list remains sorted. It should run in linear time (in the size of the list).

```
*Main> add 3 []
[3]
*Main> add 3 [1,2]
[1,2,3]
*Main> add 3 [1,3,5]
[1,3,5]
*Main> add 3 [1,5,8,9]
[1,3,5,8,9]
*Main> add "hello" ["goodbye", "world"]
["goodbye", "hello", "world"]
```

union

Write a function `union :: Ord a => [a] -> [a] -> [a]` which unions two input sets (Haskell lists). This should look similar to the "merge" step of merge-sort, and should run in linear time (in the added sizes of the input sets). You *may* use the `add` function defined above, but if you do it probably will *not* run in linear time, so it's probably better not to.

```
*Main> union [] []
[]
*Main> union [1,2,3] []
[1,2,3]
*Main> union [] [1,2,3]
[1,2,3]
*Main> union [1,2,3] [1,2,3]
[1,2,3]
*Main> union ["goodbye", "world"] ["humans", "smell"]
["goodbye", "humans", "smell", "world"]
```

intersect

Write a function `intersect :: Ord a => [a] -> [a] -> [a]` which intersects two input sets. This should run in linear time (in the size of the input sets).

```
*Main> intersect [] [1,2,3]
[]
*Main> intersect [1,2,3] []
[]
*Main> intersect [1,2,3] [3,4,45,89]
[3]
```

```
*Main> intersect ["cruel", "hello", "world"] ["good", "hello", "world"]
["hello", "world"]
```

powerset

Write a function `powerset :: Ord a => [a] -> [[a]]` which calculates the powerset of the input set. Because the output is also a set, it *must preserve our set properties*, including that there are no duplicate elements and the elements are lexicographically sorted. Using the functions `union` and `add` that you've already defined is useful here.

```
*Main> powerset []
[[]]
*Main> powerset [1,2]
[[],[1],[1,2],[2]]
*Main> powerset ["goodbye", "hello", "world"]
[ [], ["goodbye"], ["goodbye", "hello"], ["goodbye", "hello", "world"]
, ["goodbye", "world"], ["hello"], ["hello", "world"], ["world"]
]
```

Higher Order Functions

For these problems, you *must* use higher-order functions. No recursion allowed!

inclist'

Write a function `inclist' :: Num a => [a] -> [a]` which increments each element of an input list.

```
*Main> inclist' [1,2,3,4]
[2,3,4,5]
*Main> inclist' [-2,4,5,1]
[-1,5,6,2]
*Main> inclist' []
[]
*Main> inclist' [2.3, 4.5, 7.6]
[3.3,5.5,8.6]
```

sumlist'

Write a function `sumlist' :: (Num a) => [a] -> a` which adds all the elements of the input list.

```
*Main> sumlist' []
0
*Main> sumlist' [1,2,3]
6
*Main> sumlist' [-3,2,5]
```

```
4
*Main> sumlist' [3.3,2.8,-1.2]
4.8999999999999995
```

Part 2: Algebraic Data Types

include your definitions for the following problems in ***mp1b-adts/src/Lib.hs***

Algebraic Data Types

If you haven't already you may want to read Chapter 8 of Learn you a Haskell, Making Our Own Types and Typeclasses. If Chapter 8 is a little bit over your head, try out Chapter 3 Types and Typeclasses first.

We won't be making any Typeclasses in this MP, but we will be using and making Types. Specifically, we'll be making Algebraic Data Types, because that's what Haskell supports. Below are two Algebraic Data Types we supply for you to use in this assignment.

```
data List a = Cons a (List a)
            | Nil
  deriving (Show, Eq)

data Exp = IntExp Integer
         | PlusExp [Exp]
         | MultExp [Exp]
  deriving (Show, Eq)
```

The above code declares two new *type constructors*, `List` and `Exp`. `List` is a type constructor that takes one type as an argument (for example, if you said `List Int` that would be a different type than `List String` or `List Double`). `Exp` takes no type arguments.

It also declares several *data constructors*, two for `List a` and three for `Exp`. Their types are given below:

```
-- List data constructors
Cons :: a -> List a -> List a
Nil  :: List a

-- Exp data constructors
IntExp  :: Integer -> Exp
PlusExp :: [Exp] -> Exp
MultExp :: [Exp] -> Exp
```

Notice how the above type declarations are written as if the data constructors are *functions*. In fact, you can think of them as functions! `Cons` is a function that takes two arguments, (an `a` and a `List a`) and constructs a `List a`. If that

doesn't make sense to you, perhaps you need to read Chapters 3 and 8 of Learn You a Haskell as mentioned above.

The nice thing about algebraic data-constructors is that we can *pattern match* on them (see Chapter 4 of Learn You a Haskell for pattern matching). Suppose we wanted to make a function `double :: List Int -> List Int` which doubles each element of the input list. We can just ask ourselves "what should we do for each of the ways that a `List Int` can be constructed?"

```
double :: List Int -> List Int
double Nil         = Nil
double (Cons i l) = Cons (2*i) (double l)
```

Notice here that I've told Haskell "if the list is constructed as a `Nil`, then just produce `Nil` again" (this is the base-case). I've also told Haskell, "if the list is constructed as a `Cons i l` (where `i :: Int`, `l :: List Int`), then multiply the `i` by `2` and `double` the rest of the list" (the recursive case). Because I've *exhausted* all of the data-constructors for `List Int`, I *know* that Haskell will be able to apply `double` to any `List Int`. If you get a "non-exhaustive patterns" error, it means you haven't told Haskell how to handle all of the ways something of your input type can be constructed.

You'll be writing a few functions which manipulate the ADTs given above.

list2cons

Write a function `list2cons :: [a] -> List a` which converts a Haskell list into our `List` type. Do this recursively (not using higher-order functions).

```
*Main> list2cons []
Nil
*Main> list2cons [3,2,5]
Cons 3 (Cons 2 (Cons 5 Nil))
*Main> list2cons ["hello", "world"]
Cons "hello" (Cons "world" Nil)
*Main> list2cons "hello"
Cons 'h' (Cons 'e' (Cons 'l' (Cons 'l' (Cons 'o' Nil))))
```

cons2list

Write a function `cons2list :: List a -> [a]` which converts our `List` type into the Haskell list type. Do this recursively.

```
*Main> cons2list Nil
[]
*Main> cons2list (Cons 3 (Cons 4 Nil))
[3,4]
*Main> cons2list (Cons "goodbye" (Cons "world" Nil))
["goodbye", "world"]
```

eval

Write a function `eval :: Exp -> Integer` which evaluates the integer expression represented by its input. You may use recursion and higher-order functions.

```
*Main> eval (IntExp 3)
3
*Main> eval (PlusExp [])
0
*Main> eval (MultExp [])
1
*Main> eval (PlusExp [MultExp [IntExp 3, IntExp 5], PlusExp [IntExp 3], IntExp 5])
23
*Main> eval (MultExp [IntExp 3, IntExp 45, IntExp (-2), PlusExp [IntExp 2, IntExp 5]])
-1890
```

list2cons'

Write a function `list2cons' :: [a] -> List a` which converts a Haskell list into our `List` type. You are required to use higher-order functions for this, *no recursion*.

```
*Main> list2cons' []
Nil
*Main> list2cons' [3,2,5]
Cons 3 (Cons 2 (Cons 5 Nil))
*Main> list2cons' ["hello", "world"]
Cons "hello" (Cons "world" Nil)
*Main> list2cons' "hello"
Cons 'h' (Cons 'e' (Cons 'l' (Cons 'l' (Cons 'o' Nil))))
```

BinTree

Write an ADT `BinTree a` which represents a binary tree that stores things of type `a` at internal nodes, and stores nothing at the leaves.

You must add `deriving (Show)` to the `data` declaration so that GHCi can print your datatype (as we've done above for `List a` and `Exp`). Not doing so will result in a loss of points.

The data-constructors must have the following types (and names):

```
Node :: a -> BinTree a -> BinTree a -> BinTree a
Leaf :: BinTree a
```

sumTree

Write a function `sumTree :: Num a => BinTree a -> a` which takes a `BinTree a` (where `a` is a `Num`) from the previous problem and sums all the elements of its nodes.

```
*Main> sumTree Leaf
0
*Main> sumTree (Node 3 Leaf (Node 5 (Node 8 Leaf Leaf) Leaf))
16
*Main> sumTree (Node (-4) Leaf Leaf)
-4
```

SimpVal

Write an ADT `SimpVal` which represents the values that a simple programming language can have. We'll have `IntVal` for integers, `BoolVal` for booleans, `StrVal` for strings, and `ExnVal` for exceptions.

You must add `deriving (Show)` to the `data` declaration so that GHCi can print your datatype (as we've done above for `List a` and `Exp`). Not doing so will result in a loss of points.

The data-constructors must have the following types (and names):

```
IntVal  :: Integer -> SimpVal
BoolVal :: Bool    -> SimpVal
StrVal  :: String  -> SimpVal
ExnVal  :: String  -> SimpVal
```

liftIntOp

Write a function `liftIntOp :: (Integer -> Integer -> Integer) -> SimpVal -> SimpVal -> SimpVal` which will take an operator over integers (like `(+) :: Integer -> Integer -> Integer`) and turn it into an operator over `SimpVal`s. If the inputs are not `IntVal`, raise an exception by returning `ExnVal "not an IntVal!"`.

```
*Main> liftIntOp (+) (IntVal 3) (IntVal 4)
IntVal 7
*Main> liftIntOp (*) (IntVal 2) (IntVal (-5))
IntVal (-10)
*Main> liftIntOp (+) (BoolVal True) (IntVal 3)
ExnVal "not an IntVal!"
*Main> liftIntOp (+) (IntVal 5) (StrVal "hello")
ExnVal "not an IntVal!"
*Main> liftIntOp (+) (StrVal "hello") (ExnVal "not an IntVal!")
ExnVal "not an IntVal!"
```