9️⃣

Week 9: Solving Problems

Problem 1 - Folding From the Left

Similar to foldr, **foldl** helps you fold the lists from the left.

Code

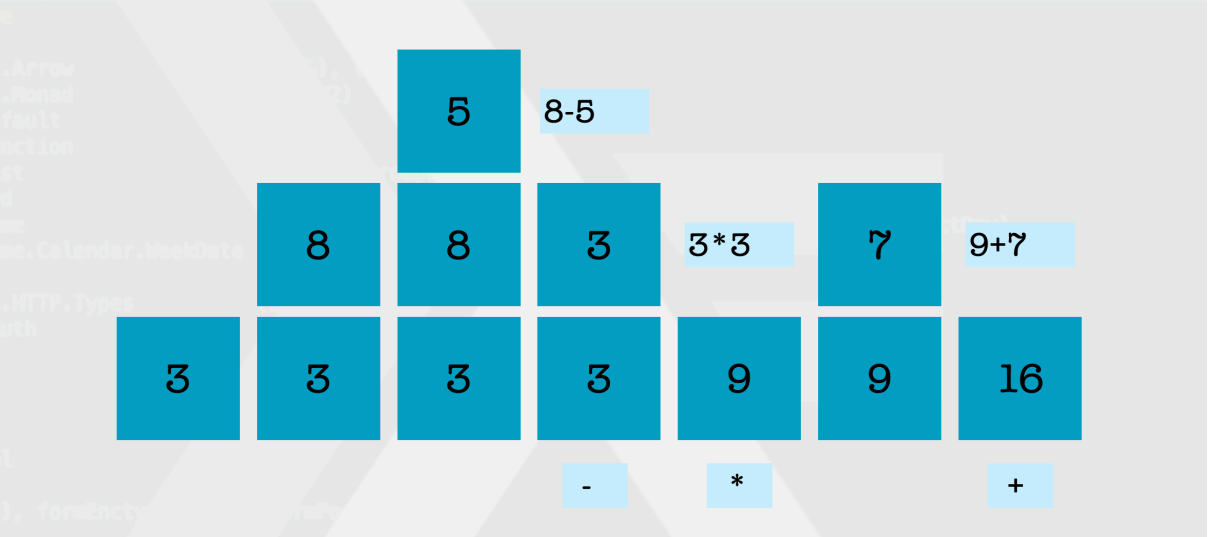
```
-- foldr Version
foldr (op) z [ ] = z
foldr (op) z (x : xs) = x op (foldr (op) z xs)

-- foldl Version
foldl (op) z [ ] = z
foldl (op) z (x : xs) = foldl (op) (z op x) xs
```

Problem 2 - Reverse Polish Notation

Doing calculations using a Stack data structure.

ℹ️ > 3 * (8 - 5) + 7
> 3 8 5 - * 7 +



Code

```
-- 1)
evalRPN :: (Num a) ⇒ String → a -- Type

-- 2)
"3 8 5 - * 7 +" -- Turns into ⟹
[“3”, “8”, “5”, “-”, “*”, “7”, “+”] -- words :: String → [String]

-- 3)
-- STACK IMPLEMENTATION
```

Stack Implementation

```
import Data.List
evalRPN :: (Num a, Read a) ⇒ String → a
evalRPN = head . foldl procStack [ ] . words -- From Data.List
```

procStack does to a list of numbers and a string essentially what we did in the stack implementation

```
procStack :: (Num a, Read a) ⇒ [a] → String → [a]
procStack (x : y : ys) “*” = (y*x) : ys
procStack (x : y : ys) “+” = (y+x) : ys
```

```
procStack (x : y : ys) "-" = (y-x) : ys
procStack xs numString = read numString : xs
```

Problem 3 - Matrices

Represent a matrix as a list of lists

```
type Matrix = [[Int]]
```

Is a list of lists of Int a matrix?

Yes, IF...

1. Every list in the list has the same length.
    - Do map length over list
    - Check every number is the same
2. There is at least one row and one column.
    - Check if the list is non-empty

Code

Every element of a list satisfies a predicate

```
-- all is a library function
all :: (a → Bool) → [a] → Bool
all p xs = foldr (&&) True (map p xs)

-- Another version using function composition
all :: (a → Bool) → [a] → Bool
all p = foldr (&&) True . map p
```

Every element of a list of Int is the same.

```
uniform :: [Int] → Bool
uniform [ ] = True -- Vacuously true
uniform xs = all (== head xs) (tail xs)
```

Check the two properties.

```
valid :: Matrix → Bool
valid [ ] = False
valid (x : xs) = not (null x) && uniform (map length (x : xs))
```

Matrix Addition

> ❗ The matrices have to be the same width and same height.

Creating a list as tuples.

```
-- zipWith is a library function
zipWith' :: (a → b → c) → [a] → [b] → [c]
zipWith' f xs ys = [f x y | (x, y) ← zip xs ys]
```

Calculating the width.

```
matrixWidth :: Matrix → Int
matrixWidth xss = length (head xss)
```

Calculating the height.

```
matrixHeight :: Matrix → Int
```

```
matrixHeight xss = length xss
```

Adding the matrices.

```
plusM :: Matrix → Matrix → Matrix
plusM m n | ok = zipWith (zipWith (+)) m n
   where ok = valid m && valid n
                    && matrixWidth m == matrixWidth n
                    && matrixHeight m == matrixHeight n
```