Fold:

- A fold takes a binary function, a starting value, called the **accumulator**, and a list to fold up. The binary function itself takes two parameters. The binary function is called with the accumulator and the first or last element and produces a new accumulator. Then, the binary function is called again with the new accumulator and the now new first or last element, and so on. Once we've walked over the whole list, only the accumulator remains, which is what we've reduced the list to.

Foldl:

- Here's the function definition and implementation of foldl:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
-- if the list is empty, the result is the initial value; else
-- we recurse immediately, making the new initial value the result
-- of combining the old initial value with the first element.
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- **Note:** foldl consumes the list left to right and evaluates from left to right.
- With foldl, the binary function has the accumulator as the first parameter and the current value as the second one.
- E.g. foldl (-) 0 [1..10] = -55

  foldl (-) 0 [1..10]
  → foldl (-) (0 - 1) [2..10]
  → foldl (-) ((0-1) - 2) [3..10]
  → foldl (-) (((0-1) - 2) - 3) [4..10]
  → foldl (-) ((((0-1) - 2) - 3) - 4) [5..10]
  → foldl (-) (((((0-1) - 2) - 3) - 4) - 5) [6..10]
  → foldl (-) ((((((0-1) - 2) - 3) - 4) - 5) - 6) [7..10]
  → foldl (-) (((((((0-1) - 2) - 3) - 4) - 5) - 6) - 7) [8..10]
  → foldl (-) ((((((((0-1) - 2) - 3) - 4) - 5) - 6) - 7) - 8) [9..10]
  → foldl (-) (((((((((0-1) - 2) - 3) - 4) - 5) - 6) - 7) - 8) - 9) [10]
  → foldl (-) ((((((((((0-1) - 2) - 3) - 4) - 5) - 6) - 7) - 8) - 9) - 10) []
  → ((((((((((0-1) - 2) - 3) - 4) - 5) - 6) - 7) - 8) - 9) - 10)
  → (((((((((-1 - 2) - 3) - 4) - 5) - 6) - 7) - 8) - 9) - 10)
  → (((((((-3 - 3) - 4) - 5) - 6) - 7) - 8) - 9) - 10)
  → (((((((-6 - 4) - 5) - 6) - 7) - 8) - 9) - 10)
  → ((((((-10 - 5) - 6) - 7) - 8) - 9) - 10)
  → (((((-15 - 6) - 7) - 8) - 9) - 10)
  → ((((-21 - 7) - 8) - 9) - 10)
  → (((-28 - 8) - 9) - 10)
  → ((-36 - 9) - 10)
  → (-45 - 10)
  → -55

Foldr:

- Here's the function definition and implementation of foldr:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
-- if the list is empty, the result is the initial value z; else
-- apply f to the first element and the result of folding the rest
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- **Note:** foldr consumes the list left to right but evaluates from right to left.
- With foldl, the binary function has the current value as the first parameter and the accumulator as the second one.
- E.g. foldr (-) 0 [1..10] = -5

```
foldr (-) 0 [1..10]
→ 1 - (foldr (-) 0 [2..10])
→ 1 - (2 - (foldr (-) 0 [3..10]))
→ 1 - (2 - (3 - (foldr (-) 0 [4..10])))
→ 1 - (2 - (3 - (4 - (foldr (-) 0 [5..10]))))
→ 1 - (2 - (3 - (4 - (5 - (foldr (-) 0 [6..10])))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (foldr (-) 0 [7..10]))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (foldr (-) 0 [8..10])))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (8 - (foldr (-) 0 [9..10]))))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (8 - (9 - (foldr (-) 0 [10])))))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (8 - (9 - (10 - (foldr (-) 0 [])))))))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (8 - (9 - (10)))))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (8 - (-1))))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (9)))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (-2))))))
→ 1 - (2 - (3 - (4 - (5 - (8)))))
→ 1 - (2 - (3 - (4 - (-3))))
→ 1 - (2 - (3 - (7)))
→ 1 - (2 - (-4))
→ 1 - (6)
→ -5
```