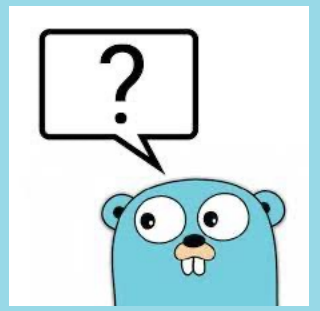
+ fx =



Funktionale Programmierung

Lea Dennhardt

Demo

Agenda

1. **Grundkonzept der funktionalen Programmierung**
   - Überblick
   - FP in Go
   - Ansätze, die noch nicht in Go sind
2. **Funktionale Funktionen in Go**
   - Closures
   - Recursion
   - Lazy vs Eager function
   - Implementation unseres Codes
3. **Vor und Nachteile**

Überblick

OO makes code understandable by encapsullating moving parts.
FP makes code understandable by minimizing moving parts.

- Michael Feathers

Grundkonzept

Imperativ vs deklarativ

```
3   func imperativ() {
4       name := "Max"
5       name = name + "Mustermann"
6
7   }
```

Imperativ

```
4   func deklarativ() {
5       const firstname = "Max"
6       const lastname = "Mustermann"
7       const name = firstname + "" + lastname
8   }
```

Deklarativ

FP in Go

- Keine funktionale Sprache

Language Feature	Support
First-Class Funktionen/Higher Order Funktionen	✅
Closures	✅
Generics	✅
Tail Call Optimization	❌
Currying	❌

Ansätze, die nicht in Go sind

unveränderliche Daten - Strukturen	Typ Parameter für Methoden	Funktion Overloading
Typ Varianz	Higher-Kinded-Types	Tuples sind keine echten Datentypen

2. Funktionale Funktionen

Higher-Order-Functions

- Funktion, die andere Funktionen als Argumente nehmen
  oder eine Funktion als ihr Ergebnis zurückgeben

- In Go:
  - Funktionen sind „first-class citizen“
  - D.h sie können Variablen zugewiesen, als Argumente übergeben und von anderen Funktionen zurückgegeben werden

Higher-Order-Functions

```go
// A higher-order function that takes a function as an argument
func applyOperation(a, b int, operation func(int, int) int) int {
    return operation(a, b)
}

// Define specific operations
func add(x, y int) int {
    return x + y
}

func multiply(x, y int) int {
    return x * y
}

func main() {
    // Use higher-order function with different operations
    fmt.Println(applyOperation(3, 5, add))      // Output: 8
    fmt.Println(applyOperation(3, 5, multiply)) // Output: 15
}
```

Closures

- Funktionswert, der Variablen von außerhalb seines Funktionskörpers referenziert

→ Funktion kann auf die referenzierten Variablen zugreifen

```
5   func adder() func(int) int {
6       sum := 0
7       return func(x int) int {
8           sum += x
9           return sum
10      }
11  }
12
13  func main() {
14      pos, neg := adder(), adder()
15      for i := 0; i < 10; i++ {
16          fmt.Println(
17              pos(i),
18              neg(-2*i),
19          )
20      }
21  }
```

Rekursion

- Technik, bei der ein komplexes Problem in kleinere Teilprobleme zerlegt wird

```
 5  func Factorial(n int) int {
 6      if n <= 1 {
 7          return 1
 8      }
 9      return n * Factorial(n-1)
10  }
11
12  func main() {
13      var n = 5
14      var fac = Factorial(n)
15      fmt.Println("The factorial of", n, "is", fac)
16  }
```

Lazy vs. Eager Evaluation

Eager Evaluation

```go
// ----------------------------------------------------------------------
// Filter Method
// ----------------------------------------------------------------------
func (list *LinkedList[T]) Filter(operation func(T) bool) LinkedList[T] {
    current := list.head
    newList := &LinkedList[T]{}
    for current != nil {
        if operation(current.data) {
            newList.Append(current.data)
        }
        current = current.next
    }
    return *newList
}
```

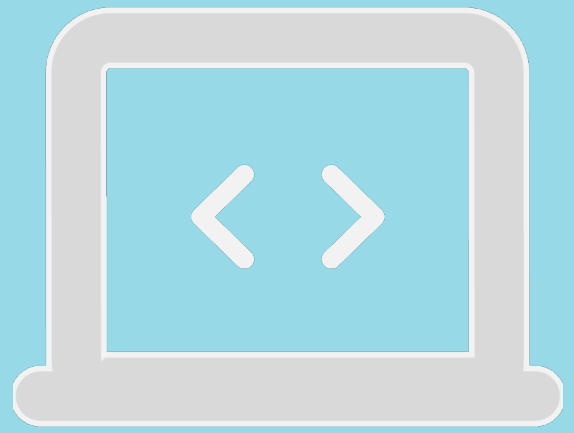
Lazy Evaluation

```go
func (list *LinkedList[T]) LazyFilter(operation func(T) bool) LazyFilterList[T] {
    current := list.head
    lazyOps := []FilterFunc[T]{}
    var datalist []T

    for current != nil {
        value := current.data
        datalist = append(datalist, value)
        lazyOps = append(lazyOps, func() bool {
            return operation(value)
        })
        current = current.next
    }

    return LazyFilterList[T]{
        Operations: lazyOps,
        data:       datalist,
    }
}
```

Implementation von uns



For Each Methode
Filter
Map
Reduce

Vor- und Nachteile unserer Implementation

Vorteil

- First-Class Functions
- Immutable types

💡 Go entwickelt sich ständig weiter

Nachteil

- Begrenzte Unterstützung für funktionale Konzepte
- Performance sinkt
- Fehlende immutable data structures
- Begrenzte Funktionalität der Receiver Funktion
- Begrenzte Typsicherheit
- Lazy Evaluation eingeschränkt

Quellen

- GO Dokumentation
- https://go.dev/doc/
- Basics of Functional Programming in Go
- https://gotz.medium.com/basics-of-functional-programming-in-go-290b5d79fc3e
- Functional Go
- https://medium.com/@geisonfgfg/functional-go-bc116f4c96a4
- Closures are the Generics for Go
- https://medium.com/capital-one-tech/closures-are-the-generics-for-go-cb32021fb5b5#
- Lazy evaluation in go
- https://blog.merovius.de/posts/2015-07-17-lazy-evaluation-in-go/
- Generic Map, Filter and Reduce in Go
- https://erikexplores.substack.com/p/generic-map-filter-and-reduce-in
- Higher-Order Functions in Go
- https://medium.com/@sandakelum/higher-order-functions-in-go-f34db4d8cb20
- Closures are the Generics for Go
- https://medium.com/capital-one-tech/closures-are-the-generics-for-go-cb32021fb5b5

FP in Go

Danke fürs Zuhören!