# Functional Programming

## Concepts of Programming Languages

Sebastian Macke
Rosenheim Technical University

# Last lecture

- Exception Handling. Error return codes vs. Exceptions
- Object Oriented Programming without class hierarchy
- Embedding
- Implicit satisfied interfaces with Polymorphism

# What is Functional programming?

- Functional programming is a programming paradigm where functions can not only be defined and applied. Like data, functions can be linked together, used as parameters, and occur as function results.
- In short: Functions are treated as first-class citizens.

# History

Functional programming is old, but some major languages have adapted them quite recently

- LISP in 1950. Introduced functional paradigm in its programming language
- JavaScript 1995 (originally idea was to embed Scheme into HTML, a functional language)
- C# in 2008
- C++ in 2011
- Java 8 in 2014
- Rust 2015
- Go 2009

## Functional programming languages are categorized into two groups

- **Pure** Functional Languages

These types of functional languages support only the functional paradigms and have no state. For example − Haskell.

- **Impure** Functional Languages

These types of functional languages support the functional paradigms and imperative style programming. For example all of the major languages today.

# Impure Functional Languages

6

# Function as variable

- Similar to a normal function, but the function is not accessible outside of main.

```
func main() {
    ADD := func(x, y int) int {
        return x + y
    }
    fmt.Println(ADD(1, 2)) // -> returns 3
    fmt.Println("The type of ADD is", reflect.TypeOf(ADD).Kind())
}
```

- Function variable can be copied.

```
ANOTHERADD := ADD;
ANOTHERADD(1, 2);
```

- The function doesn't need to be bound to a variable (Anonymous Function)

```
func main() {
    fmt.Println(func(x, y int) int { return x + y }(3, 4)) // -> returns 7
}
```

7

# Function as parameter.

- Perform a function x times

```
func do(f func(int), loops int) {
    for i := 0; i < loops; i++ {
        f(i)
    }
}

func main() {
    printvalue := func(i int) { fmt.Println(i) }
    do(printvalue, 5)
}
```

8

# Function as return value.

```
type areaFunc func(int, int) int

func getAreaFunc() areaFunc {
    return func(x, y int) int {
        return x * y
    }
}

func main() {
    areaF := getAreaFunc()
    res := areaF(2, 4)
    fmt.Println(res)
}
```

9

# Why functional programming is better - Odd Number Filter

- With imperative programming. Generate a new array and copy only the odd values.

```
array := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
filteredArray := []int{}
for _, v := range array {
    if v%2 == 0 {
        filteredArray = append(filteredArray, v)
    }
}
fmt.Println(filteredArray)
```

- With functional declarative programming. Filter the array via a stream.

```
array := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
filteredArray := stream.OfSlice(array).
    Filter(func(n int) bool { return n%2 == 0 }).
    ToSlice()
fmt.Println(filteredArray)
```

10

# Why functional programming is better - Sorting

- A sorting algorithm needs to be able to compare two elements. This is done via a comparator function.
- Before functional programming was a thing in Java, we had to provide a Comparator implementation via inheritance.

Java (unlearn this):

```
class IntComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer i1, Integer i2) {
        return i1.compareTo(i2);
    }
}

class SortDemo {
    public static void main(String[] args) {
        Integer[] array = { 3, 2, 1, 5, 8, 6 };
        Arrays.sort(array, new IntComparator());
        System.out.println(Arrays.toString(array));
    }
}
```

# Why functional programming is better - Sorting

- With functional programming, we can use a lambda expression to provide a comparator function.

```
array := []int{10, 5, 3, 7, 1, 0, 4, 6}
sortedArray := stream.OfSlice(array).
    Sorted(func(a, b int) int { return a - b }).
    ToSlice()
fmt.Println(sortedArray)
```

12

# Closures

- Function gets access to its context (e. g. variables) at creation time.

```
// intSeq returns another function, which we define anonymously in the body of intSeq.
// The returned function closes over the variable i to form a closure.
func intSeq() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}
func main() {
    // We call intSeq, assigning the result (a function) to nextInt.
    // This function value captures its own i value, which will be updated each time we call nextInt.
    nextInt := intSeq()
    // See the effect of the closure by calling nextInt a few times.
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    // To confirm that the state is unique to that particular function, create and test a new one.
    newInts := intSeq()
    fmt.Println(newInts())
}
```

# Closures II

## Access array outside of the function for sorting

```
func main() {
    data := []int{27, 15, 8, 9, 12, 4, 17, 19, 21, 23, 25}
    sort.SliceStable(data, func(i, j int) bool {
        return data[i] < data[j]
    })
    fmt.Println(data)
}
```

## In functional programming, closures encapsulate data, allowing functions to access necessary values without relying on global state or explicit access control.

# Exercise 5.1 - Warm Up

This is a Bubble Sort algorithm for ints.

```
func BubbleSort(data []int) {
    for i := 0; i < len(data); i++ {
        for j := 0; j < len(data)-1; j++ {
            if data[j] > data[j+1] {
                data[j], data[j+1] = data[j+1], data[j] // Swap the values

            }
        }
    }
}

func main() {
    data := []int{27, 15, 8, 9, 12, 4, 17, 19, 21, 23, 25}
    BubbleSort(data)
    fmt.Println(data)
}
```

Rewrite it, so that the Bubble Sort algorithm takes a comparison function as input

```
BubbleSort(data, func(i, j int) bool {return data[i] > data[j]})
```

# Closures in JavaScript

```
const object1 = {
    prop: 42,

    func: function () {
        return this.prop;
   },
};

console.log(object1.prop)  // 42
console.log(object1.func())  // 42  // At creation time, this points to test.prop

const object2 = {
    prop: 43,
    func: object1.func  // take the function defined in object1.
}

console.log(object2.prop)  // 43
console.log(object2.func())  // 42 or 43?
```

// can be bound to object1 through object1.func.bind(object1)

//prints 43

16

# Pure Functions

## What is a pure function?

- Its return value always the same for the same arguments
- The evaluation has no side effects (no mutation of data outside of the function)
- In other words: After calling a pure function, the rest of the program will be in the same state it was before calling

Examples: exp, sin, cos, max, min.

Advantages

- Reading and understanding is simpler
- Easier to test
- Are less prone to error in general

# Every Function can be made pure

- This one is impure

```
var count = 0
func increment() {
    count++
}
```

- This one is pure

```
func incrementPure(count int) int {
    return count + 1 // This function is pure because it does not modify external state
}
```

- Often functions are "pure enough" in the practical sense.

```
var array []int
newarray := append(array, 1)
```

- Question: How to make a pseudo random() function pure?
- Question: How to make a real random() function pure?

19

# Pure Functional Languages

20

# Functional Programming – Characteristics

The most prominent characteristics of functional programming are as follows

- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- Functional programming languages don't support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls. Hence, the declarative approach is prevalence rather than the imperative approach.
- Like OOP, functional programming languages can support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism

## Functional programming offers the following advantages

- Bugs-Free Code

Functional programming does not support state, so there are no side-effect results and we can write error-free codes.

- Efficiency

Functional programs consist of independent units that can **run concurrently**. As a result, such programs can be more efficient.

- Lazy Evaluation

Functional programming supports **lazy evaluation** like Lazy Lists, Lazy Maps, etc.

- Distribution

Functional programming supports distributed computing

# Many Functional Languages only support Single Argument Functions

- Currying : Converting a function with n arguments in n functions with one argument

```
// ADD with 2 parameters
ADD := func(x, y int) int {
    return x + y
}
```

```
ADD(1,2) -> 3
```

```
// Curried ADD
ADDC := func(x int) func(int) int {
    return func(y int) int {
        return x + y
    }
}
```

```
ADDC(1)(2) -> 3
```

# Currying allows for Partial Application

```
// Curried ADD
ADDC := func(x int) func(int) int {
    return func(y int) int {
        return x + y
    }
}
```

```
partialApplicatedFunction := ADDC(1)
.... Do something else
partialApplicatedFunction(3) -> 4
```

# Functional Composition

## Functions can be composed to new functions
## g(f(x)) -> (g ◦ f)(x)

```
// Function f()
f := func(x int) int {
    return x * x
}

// Function g()
g := func(x int) int {
    return x + 1
}

// Functional Composition: (g◦f)(x)
gf := func(x int) int {
    return g(f(x))
}

fmt.Printf("%v\n", gf(2)) // --> 5
```

25

# Functional Composition (2)

## Functions can be composed with functions as parameters

```
package main

import "fmt"

type function func(any) any

func main() {
    compose := func(g, f function) function {
        return func(x any) any {
            return g(f(x))
        }
    }

    square := func(x any) any { return x.(int) * x.(int) }

    fmt.Printf("%v\n", compose(square, square)(2)) // --> 4*4 = 16
    fmt.Printf("%v\n", compose(compose(square, square), square)(2))
}
```

# Famous Functional Languages

- Haskell (Introduction (https://www.youtube.com/watch?v=1jZ7j21g028)) ← Next lecture
- ML
- Clojure
- F#
- Scala

27

# Lambda Calculus - Programming with Nothing

28

# Motivation

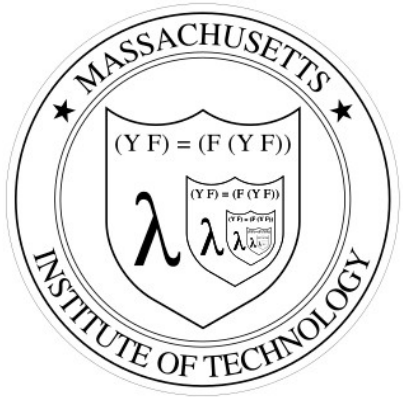
In the 1930s, mathematicians were trying to formalize the foundations of mathematics to describe all of math in purely logical terms.

This was part of a movement called **Hilbert's program**, which aimed to make mathematics fully rigorous and mechanically provable.

**Alonzo Church** developed lambda calculus as part of this effort — he wanted a simple formal system that could
precisely define what a "function" is and how functions can be applied.

Church's goal was to:
- Create a minimal, formal language that captures the essence of computation through functions and function application.
"What does it mean for something to be computable?" 29

# History: The Lambda Calculus




- What is it?
- Why is it useful?
- Where did it came from?

Professor Graham Hutton explains the Lambda Calculus (Cool Stuff :-) (https://www.youtube.com/watch?v=eis11j_iGMs)

30

# The Lambda Calculus

- Lambda calculus describes computation as the application of functions to arguments.
- No lists, integers, strings, loop, if, switch, ...
- Just anonymous functions

## Format

λ input . output

is equivalent to

```
function (input) {
    return  output
}
```

## That's all

31

# Lambda calculus

- Lambda calculus contains 3 elements

## Variables

*x*

- A variable can be itself a function
- There are no datatypes (number, logical values)

## Functions

*λx.x* (here the identity function as an example)

- Functions have no internal state

## Applications

*(λx.x) a = a*

- here the application of the identity function to a returns a
- also a can be a function
- Parenthesis help to avoid ambiguity.



see it as: x(input).x(output)

e.g.: x.x a= ax.a b=ax.y

T := a.b.aF := a.b.bN

# Lambda calculus to the extreme

github.com/woodrush/lambda-8cc (https://github.com/woodrush/lambda-8cc)

33

# Lambda Calculus in Go - All functional languages are measured by their correspondence to the lambda calculus

34

# Lambda Calculus in Go: Combinators

```
// This is the key: A recursive function definition for all functions!!!
type fnf func(fnf) fnf

func main() {
    // Boolean TRUE as function: λx.λy.x
    TRUE := fnf(func(x fnf) fnf {
        return func(y fnf) fnf {
            return x
        }
    })

    // Boolean FALSE as function: λx.λy.y
    FALSE := fnf(func(x fnf) fnf {
        return func(y fnf) fnf {
            return y
        }
    })

    fmt.Println(TRUE.ToBool())
    fmt.Println(FALSE.ToBool())
}
```

35

# Lambda Calculus in Go: Helper for printing

```
// ToBool Returns an actual bool for a Church Boolean
func (f fnf) ToBool() bool {
    // λx.x is a function which returns itself (the ID)
    ID := func(x fnf) fnf { return x }
    var ret bool
    f(func(f fnf) fnf { ret = true; return f })(func(f fnf) fnf { ret = false; return f })(ID)
    return ret
}
```

- OOP style on a function definition!
- Executes the function and provides two functions. If the first is executed it is true, if the second it is false

# Lambda Calculus in Go: Boolean Logic

```
    // Boolean Logic
    NOT := func(p fnf) fnf { return p(FALSE)(TRUE) } // λb.b False True
    AND := func(p fnf) fnf { return func(q fnf) fnf { return p(q)(p) } }
    OR := func(p fnf) fnf { return func(q fnf) fnf { return p(p)(q) } }
    EQ := func(p fnf) fnf { return func(q fnf) fnf { return p(q)(NOT(q)) } }

    fmt.Println(NOT(FALSE).ToBool())
    fmt.Println(NOT(TRUE).ToBool())

    fmt.Println(AND(FALSE)(TRUE).ToBool())
    fmt.Println(OR(FALSE)(FALSE).ToBool())
    fmt.Println(EQ(TRUE)(TRUE).ToBool())
}
```

# Lambda Calculus in JavaScript

```
TRUE = a => b => a;
FALSE = a => b => b;
NOT = f => a => b => f(b)(a);

ToBool = f => f(() => true)(() => false)()

console.log( ToBool(TRUE) ) // -> true
console.log( ToBool(FALSE) ) // -> false

console.log( ToBool(NOT(TRUE)) )  // -> false
console.log( ToBool(NOT(FALSE)) ) // -> true
```

Fundamentals of Lambda Calculus & Functional Programming in JavaScript (https://www.youtube.com/watch?v=3VQ382QG-y4)

38

# Exercise

# Functions as First Class Citizens in Go

- Go supports functions as 1st Class Citizens: Closures und Lambdas
- Functions can be assigned to variables
- Functions can be used as function parameters and return values (High Order Functions)
- Functions can be created inside functions
- The Go standard library uses functional constructs

# Sample from the Go Standard Library

- strings.map

```
// Map returns a copy of the string s with all its characters modified
// according to the mapping function. If mapping returns a negative value, the character is
// dropped from the string with no replacement.
func Map(mapping func(rune) rune, s string) string
```

- Usage

```
s := "Hello, world!"
s = strings.Map(func(r rune) rune {
    return r + 1
}, s)
fmt.Println(s) // --> Ifmmp-!xpsme"
```

# Go does not have an API similar to Java Streams

- It is possible to build such an API in Go

```
// array of generic interfaces.
stringSlice := []any{"a", "b", "c", "1", "D"}

// Map/Reduce
result := ToStream(stringSlice).
    Map(toUpperCase).
    Filter(notDigit).
    Reduce(concat).(string)

if result != "A,B,C,D" {
    t.Error(fmt.Sprintf("Result should be 'A,B,C,D' but is: %v", result))
}
// lambda (inline)
```

42

# Exercise 5.2 - Map / Filter / Reduce

Exercise 5.2 (https://github.com/s-macke/concepts-of-programming-languages/blob/master/docs/exercises/Exercise5.md#exercise-53--map--filter--reduce)

Map/Reduce is a famous functional construct implemented in many parallel and distributed collection frameworks like Hadoop, Apache Spark, Java Streams (not distributed but parallel), C# Linq

- Implement a custom M/R API with the following interface:

```
type Stream interface {
    Map(m Mapper) Stream
    Filter(p Predicate) Stream
    Reduce(a Accumulator) Any
}
```

- What is the type of Mapper, Predicate and Accumulator?
- How can you make the types generic, so they work for any type, not only for string?

# Generic Mapper, Predicate and Accumulator

```
// Predicate function returns true if a given element should be filtered.
type Predicate func(any) bool

// Mapper function maps a value to another value.
type Mapper func(o1 any) any

// Accumulator function returns a combined element.
type Accumulator func(any, any) any
```

44

## Exercise 5.2 - Word Count (WC)

Word Count is a famous algorithm for demonstrating the power of distributed collections and functional programming. Word Count counts how often a word (string) occurs in a collection. It is easy to address that problem with shared state (a map), but this solution does not scale well. The question here is how to use a pure functional algorithm to enable parallel and distributed execution.

After running Word Count, you should get the following result:

```
INPUT:  []Any{"a", "a", "b", "b", "D", "a"}
OUTPUT: "a:3, b:2, D:1, "
```

Questions

- How can you implement the problem with the already built Map()/Filter()/Reduce() functions?
- Write an Unit Test to prove that your solution works as expected!

# Classic Word Count Sample

```
// Classic wordcount sample
// ========================
func TestWordCount(t *testing.T) {
    strings := []any{"a", "a", "b", "b", "D", "a"}

    // Map/Reduce
    result := ToStream(strings).
        Map(func(o any) any {
            result := []Pair{Pair{o, 1}}
            return result
        }).
        Reduce(sumInts).([]Pair)

    for _, e := range result {
        fmt.Printf("%v:%v, ", e.k, e.v) // "a:3, b:2, D:1, "
    }
}
```

46

# Questions

- How can you implement parallel execution for our API?
- How can you implement distributed execution for our API?

# Thank you

Tags: go, programming, master (#ZgotmplZ)

Sebastian Macke
Rosenheim Technical University
Sebastian.Macke@th-rosenheim.de (mailto:Sebastian.Macke@th-rosenheim.de)
https://www.qaware.de (https://www.qaware.de)