

# Functional Programming

## Introduction

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2024/25

# Coordinates

- **Course hours:** Tu 14-16
- **Course room:** SR 106-04-007 Videokonferenz
- **Staff:** Prof. Dr. Peter Thiemann  
Gebäude 079, Raum 00-015  
Telefon: 0761 203 -8051/-8247  
E-mail: [thiemann@cs.uni-freiburg.de](mailto:thiemann@cs.uni-freiburg.de)  
Web: <http://www.informatik.uni-freiburg.de/~thiemann>
- **Staff:** Hannes Saffrich  
E-mail: [saffrich@cs.uni-freiburg.de](mailto:saffrich@cs.uni-freiburg.de)
- **Homepage:**  
<https://github.com/proglang/FunctionalProgramming>  
<https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/>

# Administrivia

## Lecture ~90 minutes/week

- lecture videos will be made available on the webpage
- presence with live recording (zoom)

## Exercise ~90 minutes exercise/week

- Friday 14-16, SR 00 031, building 051
- exercise questions modality TBA
  - ▶ discussed in next available exercise session
  - ▶ no need to hand in exercises during the semester

## Final exam

- written exam in the computer pool
- comprising theory questions and small programming tasks

# Contents

- Basics of functional programming using Haskell
- Theoretical background
- Writing Haskell programs
- Using Haskell libraries and development tools
- Your first Haskell project

# What is Haskell?

*In September of 1987 a meeting was held at the conference on Functional Programming Languages and Computer Architecture in Portland, Oregon, to discuss an unfortunate situation in the functional programming community: there had come into being more than a dozen non-strict, purely functional programming languages, all similar in expressive power and semantic underpinnings. There was a strong consensus at this meeting that more widespread use of this class of functional languages was being hampered by the lack of a common language. It was decided that a committee should be formed to design such a language, providing faster communication of new ideas, a stable foundation for real applications development, and a vehicle through which others would be encouraged to use functional languages.*

From "History of Haskell"

# What is Functional Programming?

A different approach to programming

Functions and values

rather than

Assignments and pointers

# What is Functional Programming?

A different approach to programming

Functions and values

rather than

Assignments and pointers

It will make you a better programmer

# Why Haskell?

- Haskell is a very high-level language  
(many details taken care of automatically).
- Haskell is expressive and concise  
(can achieve a lot with a little effort).
- Haskell is good at handling complex data and combining components.
- Haskell is a high-productivity language  
(prioritizes programmer-time over computer-time)



# Functional vs Imperative Programming: Variables

## Functional (Haskell)

```
x :: Int
```

```
x = 5
```

Variable `x` has value 5 forever

# Functional vs Imperative Programming: Variables

## Functional (Haskell)

```
x :: Int  
x = 5
```

Variable `x` has value 5 forever

## Imperative (Java / C)

```
int x = 5;  
...  
x = x+1;
```

Variable `x` can change its content over time

# Functional vs Imperative Programming: Functions

## Functional (Haskell)

```
f :: Int -> Int -> Int
```

```
f x y = 2*x + y
```

```
f 42 16 // always 100
```

Return value of a function **only** depends on its inputs

# Functional vs Imperative Programming: Functions

## Functional (Haskell)

```
f :: Int -> Int -> Int
```

```
f x y = 2*x + y
```

```
f 42 16 // always 100
```

Return value of a function **only** depends on its inputs

## Imperative (Java)

```
boolean flag;
```

```
static int f (int x, int y) {  
    return flag ? 2*x + y , 2*x - y;  
}
```

```
int z = f (42, 16); // who knows?
```

Return value depends on non-local variable `flag`

# Equational Reasoning and Referential Transparency

## Haskell

- 1 Variables have a fixed value
- 2 Functions have a fixed meaning that does not depend on context (Actually, the same as item 1!)

## Consequence: Equational Reasoning

- Every (sub-) expression in a program can be replaced with its value and
- every function call can be replaced with its definition
- ... without without changing the meaning of the program!
- (may change the running time, though)

# Functional vs Imperative Programming: Laziness

## Haskell

```
x = expensiveComputation  
g anotherExpensiveComputation
```

- The expensive computation will only happen if `x` is ever used.
- Another expensive computation will only happen if `g` uses its argument.

# Functional vs Imperative Programming: Laziness

## Haskell

```
x = expensiveComputation  
g anotherExpensiveComputation
```

- The expensive computation will only happen if `x` is ever used.
- Another expensive computation will only happen if `g` uses its argument.

## Java

```
int x = expensiveComputation;  
g (anotherExpensiveComputation)
```

- Both expensive computations will happen anyway.
- Laziness can be simulated, but it's complex!

# Many more features that make programs more concise

- Static typing
- Algebraic datatypes
- Polymorphic types
- Parametric overloading
- Type inference
- Monads & friends (for IO, concurrency, ...)
- Comprehensions
- Concurrency
- Metaprogramming
- ...



# References

- Paper by the original developers of Haskell in the conference on History of Programming Languages (HOPL III):  
[A History of Haskell: Being Lazy with Class](#)
- The Haskell home page: <http://www.haskell.org>
- Haskell libraries repository: <https://hackage.haskell.org/>
- Haskell Tool Stack:  
<https://docs.haskellstack.org/en/stable/README/>

