Programming in Haskell (PUH)

FER, Zagreb, 2024

The plan

- Why Haskell?
- About the course
- Lecture

Why Haskell?

①

Haskell is the **flagship language** of functional programming.

So, why **functional programming**?

- Employability
- Improved skills in all languages
- Fun and interesting

②

Haskell is the **breeding ground** for bleeding-edge features and constructs.

Haskell is a **language designer**’s favorite language.

Tail-call optimization

Higher-kinded polymorphism

Static typing

List comprehensions

Polymorphic type inference

Metaprogramming

Currying

Non-strict semantics

First-class functions

Sections

Effect systems

Higher order functions

Pattern matching

Lazy evaluation

Algebraic data types

Memory safety

Typeclasses (Ad-hoc polymorphism)

First-class operators

Tail-call optimization

Higher-kinded polymorphism

Static typing

List comprehensions

Polymorphic type inference

Metaprogramming

Non-strict semantics

Currying

First-class functions

Sections

Effect systems

Higher order functions

Pattern matching

Lazy evaluation

Algebraic data types

Memory safety

Typeclasses (Ad-hoc polymorphism)

First-class operators

Tail-call optimization

Higher-kinded polymorphism

Static typing

List comprehensions

Polymorphic type inference

Metaprogramming

Non-strict semantics

Currying

First-class functions

Sections

Effect systems

Higher order functions

Pattern matching

Lazy evaluation

Algebraic data types

Memory safety

Typeclasses (Ad-hoc polymorphism)

First-class operators

Tail-call optimization

List comprehensions

Non-strict semantics

Effect systems

Lazy evaluation

Typeclasses (Ad-hoc polymorphism)

Static typing

Metaprogramming

First-class functions

Higher order functions

Memory safety

Higher-kinded polymorphism

Polymorphic type inference

Currying

Sections

Pattern matching

Algebraic data types

First-class operators

Tail-call optimization

Higher-kinded polymorphism

Static typing

List comprehensions

Polymorphic **type inference**

Metaprogramming

Currying

Non-strict semantics

First-class functions

Sections

Effect systems

Higher order functions

Pattern matching

Lazy evaluation

Algebraic data types

Memory safety

Typeclasses (**Ad-hoc polymorphism**)

First-class operators

Tail-call optimization

Static typing

Higher-kinded polymorphism

List comprehensions

Polymorphic type inference

Metaprogramming

Non-strict semantics

Currying

First-class functions

Sections

Effect systems

Higher order functions

Pattern matching

Lazy evaluation

Algebraic data types

Memory safety

Typeclasses (Ad-hoc polymorphism)

First-class operators

Tail-call optimization

Higher-kinded polymorphism

Static typing

List comprehensions

Polymorphic **type inference**

Metaprogramming

Non-strict semantics

Currying

First-class functions

Sections

Effect systems

Higher order functions

Pattern matching

Lazy evaluation

Algebraic data types

Memory safety

Typeclasses (**Ad-hoc polymorphism**)

First-class operators

Tail-call optimization

Higher-kinded polymorphism

Static typing

List comprehensions

Polymorphic type inference

Metaprogramming

Currying

Non-strict semantics

First-class functions

Sections

Effect systems

Higher order functions

Pattern matching

Lazy evaluation

Algebraic data types

Memory safety

Typeclasses (Ad-hoc polymorphism)

First-class operators

Tail-call optimization

Higher-kinded polymorphism

Static typing

List comprehensions

Polymorphic type inference

Metaprogramming

Non-strict semantics

Currying

First-class functions

Sections

Effect systems

Higher order functions

Pattern matching

Lazy evaluation

Algebraic data types

Memory safety

Typeclasses (**Ad-hoc polymorphism**)

First-class operators

Tail-call optimization

Higher-kinded polymorphism

Static typing

List comprehensions

Polymorphic type inference

Metaprogramming

Currying

Non-strict semantics

First-class functions

Sections

Effect systems

Higher order functions

Pattern matching

Lazy evaluation

Algebraic data types

Memory safety

Typeclasses (Ad-hoc polymorphism)

First-class operators

Tail-call optimization

Higher-kinded polymorphism

Static typing

List comprehensions

Polymorphic type inference

Metaprogramming

Non-strict semantics

Currying

First-class functions

Sections

Effect systems

Higher order functions

Pattern matching

Lazy evaluation

Algebraic data types

Memory safety

Typeclasses (**Ad-hoc polymorphism**)

First-class operators

Tail-call optimization

Higher-kinded polymorphism

Static typing

List comprehensions

Polymorphic type inference

Metaprogramming

Currying

Non-strict semantics

First-class functions

Sections

Effect systems

Higher order functions

Pattern matching

Lazy evaluation

Algebraic data types

Memory safety

Typeclasses (Ad-hoc polymorphism)

First-class operators

Tail-call optimization

Higher-kinded polymorphism

Static typing

List comprehensions

Polymorphic type inference

Metaprogramming

Non-strict semantics

Currying

First-class functions

Sections

Effect systems

Higher order functions

Pattern matching

Lazy evaluation

Algebraic data types

Memory safety

Typeclasses (Ad-hoc polymorphism)

First-class operators

Tail-call optimization

Higher-kinded polymorphism

Static typing

List comprehensions

Polymorphic type inference

Metaprogramming

Currying

Non-strict semantics

First-class functions

Sections

Effect systems

Higher order functions

Pattern matching

Lazy evaluation

Algebraic data types

Memory safety

Typeclasses (Ad-hoc polymorphism)

First-class operators

Tail-call optimization

List comprehensions

Non-strict semantics

Effect systems

Lazy evaluation

Typeclasses (Ad-hoc polymorphism)

Static typing

Metaprogramming

First-class functions

Higher order functions

Memory safety

Higher-kinded polymorphism

Polymorphic type inference

Currying

Sections

Pattern matching

Algebraic data types

First-class operators

Haskell in a nutshell

- **declarative** vs imperative
- **statically-typed** vs dynamically-typed
- **strongly-typed** vs loosely-typed
- **functional** vs procedural vs object-oriented vs …
- **pure** vs allowing side effects
- **lazy** vs eager
- **type inference** vs manifest typing
- **nominal typing** vs structural typing
- **immutable** vs mutable

```haskell
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

```haskell
quicksort [] = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
        lesser = filter (< p) xs
        greater = filter (>= p) xs
```

```haskell
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

```haskell
data PieceType = Pawn | Knight | Bishop | Rook | Queen | King
  deriving (Eq, Enum, Ord, Show)

data Square = Square File Rank
  deriving (Eq, Ord, Show)

data Board = Board [(Piece, Square)]
  deriving (Eq, Show)

initialBoard :: Board
initialBoard =
  Board $
    concat
      [ capitalPieces Black R8,
        pawns Black R7,
        pawns White R2,
        capitalPieces White R1
      ]
  where
    pawns color rank = (\f -> (Piece color Pawn, Square f rank)) <$> [FA .. FH]
    capitalPieces color rank = zip (Piece color <$> capitalPiecesOrder) ((`Square` rank) <$> [FA .. FH])
    capitalPiecesOrder = [Rook, Knight, Bishop, Queen, King, Bishop, Knight, Rook]

getBoard :: Game -> Board
getBoard (Game moves) = foldl' (\board move -> fromEither $ performMoveOnBoard board move) initialBoard moves

isPlayerInCheck :: Color -> Board -> Bool
isPlayerInCheck currentPlayerColor board@(Board pieces) = any isKingUnderAttackByPiece oponnentPieces
  where
    kingsSquare = findKing currentPlayerColor board
    oponnentColor = oppositeColor currentPlayerColor
    oponnentPieces = filter (\(Piece c _, _) -> c == oponnentColor) pieces
    isKingUnderAttackByPiece piece = kingsSquare `S.member` getValidDstSquaresForPiece piece
    getValidDstSquaresForPiece (Piece _ _, pieceSquare) = getMoveDstSquare `S.map` fromEither (getValidSimpleMoves c

performMove :: Game -> MoveOrder -> Either String Game
performMove game@(Game moves) moveOrder = do
  validMove <- makeValidMove game moveOrder
  return $ Game $ validMove : moves
```

③

You will learn **not only** Haskell…

You will also...

- Learn to use **Git** and **GitHub**
- Get professional **code reviews**
- Get a bunch of **learning resources** (Haskell or otherwise)
- Learn to use the **CLI** and other industry-standard tools
- Learn more about **programming languages** in general.
- Get to **talk with us** about anything you want (careers, linux, editor setup...)

About the course...

Lecturers

Ante Kegalj

Luka Hadžiegrić

Filip Sodić

Mihovil Ilakovac

Teaching assistants

Anton Vučinić

Nikola Kraljević

Mislav Đomlija

Donik Vršnak

Janko Vidaković

Miho Hren

Guest Lecturers

Jan Šnajder
(Chief Lecturer)

Martin Šošić
(CTO @ Wasp)

Matija Šošić
(CEO @ Wasp)

How the course works

- Lectures
  - Held in person
  - **Mandatory**, 1 absence allowed
  - Full schedule available on Ferweb (mostly Thursdays)
- Training Exercises
  - Homeworks given **after each lecture** (give or take)
  - Submitted through GitHub
  - All homeworks must pass **unit tests and TA code review**
- Seminar
  - A larger practical project
  - Handed out in the second cycle
  - Must pass an **in-person review** at the end of the semester

To pass, you must:

- Attend lectures
- Submit homeworks on time
- Hand in the seminar

Our **Discord server** is the source of truth for all materials and announcements:

https://discord.gg/xvGb5jp8