

# Functional Programming? Haskell?

## Functional Programming



Universiteit Utrecht

[Faculty of Science  
Information and Computing  
Sciences]

```
import Data.Char(toUpper)
```

```
mkWelcome stylize year n = concat  
  [ stylize "Welcome"  
    , " to INFOFP in " ++ show year ++ "!\n\n"  
    , "We have " ++ show n ++ " students.\n\n"  
    , "So I will have to grade " ++  
      show (numExams n) ++ " exams...."  
    ]
```

```
where numExams m = 2 * m
```

```
main = putStrLn welcomeMsg
```

```
where capitalize = map toUpper
```

```
welcomeMsg = mkWelcome capitalize 2021 351
```



# What is Functional Programming?



Universiteit Utrecht

[Faculty of Science  
Information and Computing  
Sciences]

# What is Functional Programming?

- ▶ A way of thinking about problems:

Define what something **is** rather than **how** to compute it.



# Imperative (C#) vs. Functional (Haskell)

```
int sumUpTo(int n) {  
    int total = 0;  
    for (int i = n; n > 0; i--)  
        total += i;  
    return total;  
}
```

sumUpTo 0 = 0

sumUpTo n = n + sumUpTo (n-1)



# Our aim is to

Teach you **functional programming** techniques

- ▶ Using functions as first-class values
- ▶ Separating pure and impure computations
- ▶ Reasoning about your programs
- ▶ Use strong types
- ▶ ...



# Our aim is to

Teach you **functional programming** techniques

- ▶ Using functions as first-class values
- ▶ Separating pure and impure computations
- ▶ Reasoning about your programs
- ▶ Use strong types
- ▶ ...

You can write “functional code” in almost any language!



# Why Functional Programming?



Universiteit Utrecht

[Faculty of Science  
Information and Computing  
Sciences]



# To create better software

## 1. Short term: fewer bugs

- ▶ **Purity** means fewer surprises when programming
  - ▶ A function can no longer mutate a global state
- ▶ **Purity** makes it easier to **reason** about programs
  - ▶ Reasoning about OO  $\implies$  master/PhD course
  - ▶ Reasoning about FP  $\implies$  this course
- ▶ Higher-order functions **remove** lots of **boilerplate**
  - ▶ Also, less code to test and fewer edge cases
- ▶ **Types** prevent the “stupid” sort
  - ▶ What does `True + "1"` mean?



# To create better software

## 1. Short term: fewer bugs

- ▶ **Purity** means fewer surprises when programming
  - ▶ A function can no longer mutate a global state
- ▶ **Purity** makes it easier to **reason** about programs
  - ▶ Reasoning about OO  $\implies$  master/PhD course
  - ▶ Reasoning about FP  $\implies$  this course
- ▶ Higher-order functions **remove** lots of **boilerplate**
  - ▶ Also, less code to test and fewer edge cases
- ▶ **Types** prevent the “stupid” sort
  - ▶ What does `True + "1"` mean?

## 2. Long term: more maintainable

- ▶ Types are **always updated** documentation
- ▶ Types help a lot in **refactoring**
  - ▶ Change a definition, fix everywhere the compiler tells you there is a problem



```

import Data.Char(toUpper)

mkWelcome stylize year n = concat
  [ stylize "Welcome"
  , " to INFOFP in " ++ show year ++ "!\n\n"
  , "We have " ++ show n ++ " students.\n\n"
  , "So I will have to grade " ++
    show (numExams n) ++ " exams...."
  ]

where numExams m = 2 * m

main = putStrLn welcomeMsg
  where capitalize = map toUpper
        welcomeMsg = mkWelcome capitalize 2021 351

```



WELCOME to INFOFP in 2021!

We have 351 students.

So I will have to grade 702 exams....



# How?



**Universiteit Utrecht**

[Faculty of Science  
Information and Computing  
Sciences]

## Lectures:

- ▶ Tuesday, 9.00 to 10.45
- ▶ Thursday, 13.15 to 15.00

## Instructions !!!!!: Once a week

- ▶ Thursday, 17.15 to 19.00 (at USP)

## Practicals

- ▶ Tuesday, 11.00 to 12.45 (online)



# Who?

Matthijs Vakar and Frank Staals (me) in the lectures

- ▶ Contact us through email
- ▶ We both speak Dutch

10 teaching assistants in the labs

- ▶ Most of them are Dutch speakers

Guest lecture at the end of the course



# Resources

1. **Slides** contain most of the content
  - ▶ In some cases, supplemented by additional material
2. Pen-and-paper **exercises**
  - ▶ There's more than programming in this course
  - ▶ Ask questions during instruction sessions
  - ▶ Remember: there is **no compiler** at the exam
3. Book: **Programming in Haskell** (2nd edition) by Graham Hutton
  - ▶ The course follows it, except for chapters 13 and 17
  - ▶ More resources can be found in the website





# Midterm & Final Exam

- ▶ 'Pen-and-Paper' style exam questions
  - ▶ Closed book
  - ▶ No compiler
- ▶ Remindo-based



# Practical assignments

1. The first one helps you getting started
2. Three small ones with DOMJudge, one per week
3. One bigger project at the end



# DOMJudge assignments

- ▶ Submissions are **individual**
  - ▶ Do not plagiarize!
- ▶ Graded automatically : Pass vs Fail
  - ▶ correct = Pass, not fully correct = Fail
- ▶ You need to pass at least 2 out of 3 DOMJudge Assignments



# Style

- ▶ Hints in DOMJudge for good style
- ▶ Ask TAs for advice during practicals
- ▶ Important part of the final project grade



# Final project

Develop your own **game** in Haskell

- ▶ Work in **pairs** is allowed and recommended
- ▶ Submission in two parts
  1. Preliminary design document
  2. Code of the project



# Optional bonus assignment

Learn and explain a Haskell library or language feature

- ▶ Up to additional 0.5 points for the final grade
- ▶ Work in groups of at most three
- ▶ More details after mid-term exam



# Grading

Linear combination of three grades

- ▶ **Theory**  $T = 0.3 \times \text{midterm} + 0.7 \times \text{final}$
- ▶ **Practical** = Final project
- ▶ **Optional** assignment O

**Final** grade  $F = 0.5 \times T + 0.5 \times P + 0.05 \times O$

To pass the course, you essentially need

- ▶  $F \geq 5.5, T \geq 5, P \geq 5$
- ▶ Pass at least two DOMJudge assignments

See website for details.



# If you did the course last year

- ▶ **Resubmit** your DOMJudge assignments
- ▶ Redo the **final project**
  - ▶ Using the same code as last year is **not** allowed
- ▶ Redo **all** the **exams**





# Communication channels

- ▶ Teams
  - ▶ Lectures and Practicals through Teams
- ▶ E-mail
  - ▶ Check your UU-mail regularly
- ▶ Blackboard
  - ▶ As a means to access your grades.



# Course Website

<http://www.cs.uu.nl/docs/vakken/fp>

- ▶ All important information is found there
- ▶ Schedule, slides, assignments, exercises



# Getting Started:



Universiteit Utrecht

[Faculty of Science  
Information and Computing  
Sciences]

# Functional Programming Features?

Some distinguishing **features** of FP:

1. Recursion instead of iteration
2. Pattern matching on values
3. Expressions instead of statements
4. Functions as first-class citizens



# Try it!

1. Go to <http://repl.it/languages/haskell>

2. Write your definitions on the left pane

```
sumUpTo 0 = 0
```

```
sumUpTo n = n + sumUpTo (n-1)
```

3. Click **Run**

4. Load your code with “ghci main.hs”

5. Execute your functions on the right pane

```
> sumUpTo 3
```

```
6
```

6. Update the example to compute

$n! = n * (n - 1) * (n - 2) * .. * 1$  instead.



# Recursion instead of iteration

**Iteration** = repeating a process a number of times

```
int sumUpTo(int n) {  
    int total = 0;  
    for (int i = n; n > 0; i--)  
        total += i;  
    return total;  
}
```

**Recursion** = defining something in terms of itself

sumUpTo 0 = 0

sumUpTo n = n + sumUpTo (n-1)



# Pattern matching on values

A function is defined by a series of **equations**

- ▶ The value is compared with each left side until one “fits”
- ▶ In `sumUpTo`, if the value is zero we return zero, otherwise we fall to the second one

`sumUpTo 0 = 0`

`sumUpTo n = n + sumUpTo (n-1)`



# Expressions instead of statements

What code **does** versus what code **is**

- ▶ Statements manipulate the **state** of the program
- ▶ Statements have an inherent **order**
- ▶ **Variables** name and store pieces of state

```
int sumUpTo(int n) {  
    int total = 0;  
    for (int i = n; n > 0; i--)  
        total += i;  
    return total;  
}
```





# Expressions instead of statements

What code **does** versus what code **is**

- ▶ Value of a **whole expr.** depends only on its **subexpr.**
- ▶ Easier to compose and **reason** about
  - ▶ We will learn how to reason about programs

```
sumUpTo 3  --> 3 + sumUpTo 2
           --> 3 + 2 + sumUpTo 1
           --> ...
```



# The factorial example:

6. Update the example to compute  $n! = n * (n - 1) * (n - 2) * .. * 1$  instead.



# The factorial example:

6. Update the example to compute  $n! = n * (n - 1) * (n - 2) * .. * 1$  instead.

```
fac    :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```



# The factorial example:

6. Update the example to compute  $n! = n * (n - 1) * (n - 2) * .. * 1$  instead.

```
fac    :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

- ▶ Each equation goes into its own line
- ▶ Equations are checked in order
  - ▶ If `n` is 0, then the function equals 1
  - ▶ If `n` is different from 0, then it goes to the second
- ▶ **Good style:** always write the type of your functions



# Question

What happens if we write?

```
fac :: Int -> Int
fac n = n * fac (n-1)
fac 0 = 1
```



# Functions as first-class citizens

**Function** = mapping of arguments to a result

*-- In the left pane*

```
greet name = "Hello, " ++ name ++ "!"
```

- ▶ Functions can be parameters of another function
- ▶ Functions can be returned from functions

*-- In the right pane*

```
> map greet ["Mary", "Joe"]  
["Hello, Mary!", "Hello, Joe!"]
```

map applies the function greet to each element of the list



# Try it yourself!

Build greet with two arguments

```
> greet "morning" "Paul"  
"Good morning, Paul!"
```

```
-- Here is the version with one argument  
greet name = "Hello, " ++ name ++ "!"
```



# Why Haskell?

Haskell can be defined with four adjectives

- ▶ Functional
- ▶ Statically typed
- ▶ Pure
- ▶ Lazy





# Haskell is statically typed

- ▶ Every expression and function has a **type**
- ▶ The compiler **prevents** wrong combinations

```
> :t greet  -- Give me the type of greet
greet :: [Char] -> [Char]
> greet "Joe"
"Hello, Joe!"
> greet True
Couldn't match expected type '[Char]'
      with actual type 'Bool'
```

**Inference** = if no type is given for an expression, the compiler **guesses** one



# Haskell is pure

- ▶ You **cannot** use statement-based programming
  - ▶ Variables do not change, only give names
  - ▶ Program is easy to compose, understand and parallelize
- ▶ Functions which interact with the “outer world” are marked in their type with IO
  - ▶ This prevents unintended side-effects

```
readFile :: FilePath -> IO ()
```



# Haskell is lazy

We shall get to this one...



# Why Haskell?

## From a pedagogical standpoint

- ▶ Haskell **forces** a functional style
  - ▶ In contrast with imperative and OO languages
  - ▶ We can do **equational reasoning**
- ▶ Haskell teaches the value of static types
  - ▶ Compiler finds bugs long before run time
  - ▶ We can express really detailed invariants



# How do I “run” Haskell?



- ▶ We are going to use GHC in this course
  - ▶ The (Glorious) **G**lasgow **H**askell **C**ompiler
  - ▶ State-of-the-art and open source
- ▶ Installing
  - ▶ Go to <https://www.haskell.org/downloads>
  - ▶ Follow the installation instructions for installing 'stack' on your OS.



# Compiler versus interpreter


- ▶ Compiler (ghc)
  - ▶ Takes one or more files as an input
  - ▶ Generates a library or complete executable
  - ▶ There is **no interaction**
  - ▶ How you do things in  
Imperatief/Mobiel/Gameprogrammeren
- ▶ Interpreter (ghci)
  - ▶ **Interactive**, expressions are evaluated on-the-go
  - ▶ Useful for **testing** and **exploration**
  - ▶ You can also **load** a file
    - ▶ Almost as if you have typed in the entire file
  - ▶ repl.it is web-based ghci




# GHC interpreter, ghci

1. Open a command line, terminal or console





► Right now, just `repl.it`

2. Write `ghci` and press 

```
GHCi, version 8.10.2: http://www.haskell.org/ghc/  
Prelude>
```

3. Type an expression and press  to evaluate

```
Prelude> 2 + 3  
5  
Prelude>
```

4.  +  ( +  in Mac) or `:q`  to quit

```
Prelude> :q  
Leaving GHCi.
```





# First examples

```
> length [1, 2, 3]
3
> sum [1 .. 10]
55
> reverse [1 .. 10]
[10,9,8,7,6,5,4,3,2,1]
> replicate 10 3
[3,3,3,3,3,3,3,3,3,3]
> sum (replicate 10 3)
30
```

- ▶ Integer numbers appear as themselves
- ▶ `[1 .. 10]` creates a list from 1 to 10
- ▶ Functions are called (applied) **without** parentheses
  - ▶ In contrast to `replicate(10, 3)` in other languages



# More about parentheses

- ▶ Parentheses delimit subexpressions
  - ▶ `sum (replicate 10 3)`: `sum` takes 1 parameter
  - ▶ `sum replicate 10 3`: `sum` takes 3 parameters

```
> sum replicate 10 3
```

```
<interactive>: error:
```

- Couldn't match type '[t0]' with 't1 -> t'  
Expected type: `Int -> t0 -> t1 -> t`  
Actual type: `Int -> t0 -> [t0]`

```
> sum (replicate 10 3)
```

```
30
```



# First examples of types

```
> :t reverse
```

```
reverse :: [a] -> [a]
```

```
> :t replicate
```

```
replicate :: Int -> a -> [a]
```

- ▶ `->` separates each argument and the result
- ▶ `Int` is the type of (machine) integers
- ▶ `[Something]` declares a list of `Something`s
  - ▶ For example, `[Int]` is a list of integers
- ▶ `[a]` means list of **anything**
  - ▶ Note that `a` starts with a lowercase letter
  - ▶ `a` is called a **type variable**



# Operators

```
> [1, 2] ++ [3, 4]
[1, 2, 3, 4]
> (++) [1, 2] [3, 4]
> :t (++)
(++) :: [a] -> [a] -> [a]
```

- ▶ Some names are completely made out of symbols
  - ▶ Think of +, \*, &&, ||, ...
  - ▶ They are called **operators**
- ▶ Operators are used **between** the arguments
  - ▶ Anywhere else, you use parentheses



# Question

What happens if we do?

```
> [1, 2] ++ [True, False]
```



# Question

What happens if we do?

```
> [1, 2] ++ [True, False]
```

Type error!



# Define a function in the interpreter

```
> average ns = sum ns `div` length ns
> average [1,2,3]
2
> :t average
average :: Foldable t => t Int -> Int
```

- ▶ Functions are defined by one or more **equations**
- ▶ You turn a function into an operator with backticks
- ▶ Naming requirements
  - ▶ Function names must start with a lowercase
  - ▶ Arguments names too
- ▶ GHC has **inferred** a type for your function



# Define a function in a file

You can write this definition in a file

```
average :: [Int] -> Int
average ns = sum ns `div` length ns
```

and then load it in the interpreter

```
> :load average.hs
[1 of 1] Compiling Main ( average.hs, interpreted )
> average [1,2,3]
2
```

or even work on it and then reload

```
> :r
[1 of 1] Compiling Main ( average.hs, interpreted )
```





# More basic types

- ▶ Bool: True or False (note the uppercase!)
  - ▶ Usual operations like && (and), || (or) and not
  - ▶ Result of comparisons with ==, /=, <, ...
  - ▶ **Warning!** = defines, == compares

```
> 1 == 2 || 3 == 4
```

```
False
```

```
> 1 < 2 && 3 < 4
```

```
True
```

```
> nand x y = not (x && y)
```

```
> nand True False
```

```
True
```



# More basic types

- ▶ Char: one single symbol
  - ▶ Written in **single** quotes: 'a', 'b', ...
- ▶ String: a sequence of characters
  - ▶ Written in **double** quotes: "hello"
  - ▶ They are simply [Char]
    - ▶ All list functions work for String

```
> ['a', 'b', 'c'] ++ ['d', 'e', 'f']  
"abcdef"  
> replicate 5 'a'  
"aaaaa"
```



# First example of higher-order function

```
> map fac [1 .. 5]
[1,2,6,24,120]
> map not [True, False, False]
[False,True,True]
```

```
> :t map
map :: (a -> b) -> [a] -> [b]
```

- ▶ map takes **two** arguments
  - ▶ The first argument is a function  $a \rightarrow b$
  - ▶ The second argument is a list  $[a]$
- ▶ map works for every pair of types  $a$  and  $b$  you choose
  - ▶ We say that map is **polymorphic**



# Homework

1. Install GHC in your machine
2. Try out the examples
3. Define some simple functions
  - ▶ Sum from  $m$  to  $n$
  - ▶ Build `greeter` with two arguments

```
> greeter "morning" ["P", "Z"]  
["Good morning, P!", "Good morning, Z!"]
```
4. Think about the types of those functions
5. Do Practical Assignment 0.



# Three pieces of advice

## 1. Get yourself a good editor

- ▶ At the very least, with syntax highlighting
- ▶ Visual Studio Code and Atom are quite nice
  - ▶ Available at `code.visualstudio.com` and `atom.io`
  - ▶ Install Haskell syntax highlighting afterwards
- ▶ vi or Emacs for the adventurous

## 2. Get comfortable with the command line

- ▶ [https://tutorial.djangogirls.org/en/intro\\_to\\_command\\_line/](https://tutorial.djangogirls.org/en/intro_to_command_line/)



# Three pieces of advice

## 1. Get yourself a good editor

- ▶ At the very least, with syntax highlighting
- ▶ Visual Studio Code and Atom are quite nice
  - ▶ Available at `code.visualstudio.com` and `atom.io`
  - ▶ Install Haskell syntax highlighting afterwards
- ▶ vi or Emacs for the adventurous

## 2. Get comfortable with the command line

- ▶ [https://tutorial.djangogirls.org/en/intro\\_to\\_command\\_line/](https://tutorial.djangogirls.org/en/intro_to_command_line/)

## 3. Go to the Instruction sessions !!!

- ▶ And do the pen-and-paper exercises !!!

