



# Introduction

Advanced functional programming

---

Wouter Swierstra

**Welcome to the Utrecht AFP Summer School**

## This morning

- Introduce myself and the other lecturers
- Overview of the course
- Basic Haskell review
- Answer any questions regarding organizational issues

- Wouter Swierstra
- Alejandro Serrano Mena
- Gabriele Keller
- Trevor McDonell

- Lectures held in room 075 (or 201) of the BBG
- $2 \times 45$  minute slots with a 15 minute break

Short breaks between lectures; longer break for lunch.

Coffee will be provided in the coffee room – but you may want to use the coffee machines.

Labs will be held in rooms 175 and 179 of the BBG.

We have lab machines for you to use – but you may prefer to use your own computer.

We have a series of exercises for you to work on, grouped thematically.

Support from lecturers and assistants on exercises, problems installing Haskell, etc.

- Course homepage:

<https://www.afp.school>

We will update the homepage regularly with slides and further information.

- Lambda calculus
- Monads & I/O
- Datatype generic programming
- Type systems
- Applicative, foldable & traversable
- Generalized Algebraic Data Types
- Laziness
- Parallel programming
- Testing and tools
- Type families



- Jeremie Dimino (Jane Street)
- Atze Dijkstra (Standard Chartered)

## Lunch and dinner

- Catered lunches will be provided in the coffee room (on the fifth or first floor);
- Dinner each night will be at different restaurant across Utrecht.

Both lunches and dinners are included in your registration fee.

Two drinks are included with dinner – feel free to order more drinks at your own expense.

We gratefully acknowledge or sponsors:

- Jane Street Capital
- Standard Chartered Bank
- Tweag.io

They are always looking to recruit new Haskell programmers. If you're interested in working for them, feel free to approach Atze or Jeremie on Friday.

I've also invited a few people from local companies using Haskell to the dinner on Thursday (Chordify and Channable).

## Some suggested further reading

- *Parallel and concurrent programming in Haskell* by Simon Marlow
- *Fun of Programming* edited by Jeremy Gibbons and Oege de Moor
- *Purely Functional Data Structures* by Chris Okasaki
- *Types and Programming Languages* by Benjamin Pierce
- *AFP summer school* series of lecture notes on various topics

I'll try to collect links online about the individual lectures – but previous year's website has more info for now.

**Questions?**

- A pure functional language
- Data types and pattern matching
- Higher order functions
- Polymorphism
- Type classes

## Function definitions

Functions are typically defined by pattern matching:

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

```
map f []      = []
```

```
map f (x:xs) = f x : map f xs
```

Here the map function takes a function and input list as argument;

It produces a new list, where the function has been applied to every element of the input list.

## Data types and pattern matching

Besides defining functions on *lists* we can declare our own data types:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
mapTree :: (a -> b) -> Tree a -> Tree b
```

```
mapTree f (Leaf x)  = Leaf (f x)
```

```
mapTree f (Node l r) = Node (mapTree f l)  
                           (mapTree f r)
```

We can define functions on trees by *pattern matching*.



## Higher-order functions

Functions such as `map` and `mapTree` are *higher-order functions* – functions that take functions as arguments.

In Haskell, functions are *first class citizens* – they can be bound to variables or passed to other functions.

This pattern pops up again and again; we'll see lots of higher-order functions in the lectures on monads, testing and elsewhere.

Functions such as `map` and `mapTree` are *polymorphic*:

```
incrementList = map (\x -> x + 1)
```

```
checkList = map (\x -> x > 3)
```

The two calls to `map` pass functions of different **types** as arguments.

**Question:** What are the types of these two functions?

Polymorphic functions are *oblivious* to the values on which they operate.

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

The first argument could be *any* function!

Polymorphic functions are *oblivious* to the values on which they operate.

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

The first argument could be *any* function!

Oftentimes, we want know something about the data which we manipulate.

```
sort :: [a] -> [a]
```

To define a sorting function, we need to compare the elements of type `a`.

# Haskell classes

Haskell classes define an *interface*:

```
class Eq a where  
    (==) :: a -> a -> Bool
```

We can define *instances* by providing operations on a specific type:

```
instance Eq Bool where  
    True == True    = True  
    False == False = True  
    _ == _          = False
```

We can define more complicated instances, such as the `Eq` instance for lists, assuming that we have already defined an `Eq` instance for its elements:

```
instance Eq a => Eq [a] where
    [] == []           = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _ == _            = False
```

We can now compare lists of booleans, lists of lists of booleans, etc.

**Question:** What should the `Eq` instance for pairs be?

## Using classes

When we declare a type signature using a class constraint such as `Eq a`, we can use the equality function to compare elements of type `a`.

For example, the `elem` function checks if a specific element occurs in a list or not:

```
elem :: Eq a => a -> [a] -> Bool
elem e []      = False
elem e (x:xs) = e == x || elem e xs
```

We can use `elem` on lists of any types, provided there is a corresponding `Eq` instance.

This is sometimes referred to as *ad-hoc polymorphism*.

## **Packages and modules**



Once you start to organize larger units of code, you typically want to split this over several different files.

In Haskell, each file contains a separate *module*.

Let's start with a quick recap and reviewing the strengths and weaknesses of Haskell's module system.

## Goals of the Haskell module system

- Units of separate compilation (not supported by all compilers).
- Namespace management

There is (or rather was until recently) language concept of interfaces or signatures in Haskell, except for the class system.

```
module M(D(),f,g) where
import Data.List(unfoldr)
import qualified Data.Map as M
import Control.Monad hiding (mapM)
```

- Hierarchical modules
- Export list
- Import list, hiding list
- Qualified, unqualified
- Renaming of modules

- If the module header is omitted, the module is automatically named `Main`.
- Each full Haskell program has to have a module `Main` that defines a function

```
main :: IO()
```

Module names consist of at least one identifier starting with an uppercase letter, where each identifier is separated from the rest by a period.

- This former extension to Haskell 98, has been formalized in an addendum to the Haskell 98 Report and is now widely used.
- Implementations expect a module `X.Y.Z` to be named `X/Y/Z.hs` or `X/Y/Z.lhs`
- There are no relative module names – every module is always referred to by a unique name.

Most of Haskell 98 standard libraries have been extended and placed in the module hierarchy – moving `List` to `Data.List`.

Good practice: Use the hierarchical modules where possible. In most cases, the top-level module should only refer to other modules in other directories.

## Importing modules

- The `import` declarations can only appear in the module header, i.e., after the `module` declaration but before any other declarations.
- A module can be imported multiple times in different ways.
- If a module is imported qualified, only the qualified names are brought into scope. Otherwise, the qualified and unqualified names are brought into scope.
- A module can be renamed using `as`. Then, the qualified names that are brought into scope are using the new `modid`.
- Name clashes are reported lazily.

- The module `Prelude` is imported implicitly as if

```
import Prelude
```

has been specified.

- An explicit `import` declaration for `Prelude` overrides that behaviour

qualified `Prelude`

causes all names from `Prelude` to be available only in their qualified form.



- Modules are allowed to be mutually recursive.
- This is not supported well by GHC, and therefore somewhat discouraged.

**Question:** Why might it be difficult?

## **Best practices**

- Use qualified names instead of pre- and suffixes to disambiguate.
- Use renaming of modules to shorten qualified names.
- Avoid hiding
- Recall that you can import the same module multiple times.

- Packages are collections of modules that are distributed together.
- Packages are *not* part of the Haskell standard.
- Packages are versioned and can depend on other packages.
- Packages contain modules. Some of those modules may be hidden.

## Never use TABs

- Haskell uses layout to delimit language constructs.
- Haskell interprets TABs to have 8 spaces.
- Editors often display them with a different width.
- TABs lead to layout-related errors that are difficult to debug.
- Even worse: mixing TABs with spaces to indent a line.

**Question:** What might go wrong?

## Never use TABs

- Never use TABs.
- Configure your editor to expand TABs to spaces, and/or highlight TABs in source code.

# Alignment

- Use alignment to highlight structure in the code!
- Do not use long lines.
- Do not indent by more than a few spaces.

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

```
map f []          = []
```

```
map f (x : xs)   = f x : map f xs
```

- Use informative names for functions.
- Use CamelCase for long names.
- Use short names for function arguments.
- Use similar naming schemes for arguments of similar types.



## Spaces and parentheses

- Generally use exactly as many parentheses as are needed.
- Use extra parentheses in selected places to highlight grouping, particularly in expressions with many less known infix operators.
- Function application should always be denoted with a space.
- In most cases, infix operators should be surrounded by spaces.

- Use blank lines to separate top-level functions.
- Also use blank lines for long sequences of `let`-bindings or long `do`-blocks, in order to group logical units.

## Avoid large functions

- Try to keep individual functions small.
- Introduce many functions for small tasks.
- Avoid local functions if they need not be local.

**Question:** Why?

## Type signatures

- Always give type signatures for top-level functions.
- Give type signatures for more complicated local definitions, too.
- Use type synonyms.

```
checkTime :: Int -> Int -> Int -> Bool
```

# Type signatures

- Always give type signatures for top-level functions.
- Give type signatures for more complicated local definitions, too.
- Use type synonyms.

```
checkTime :: Int -> Int -> Int -> Bool
```

```
checkTime :: Hours -> Minutes -> Seconds -> Bool
```

```
type Hours = Int
```

```
type Minutes = Int
```

```
type Seconds = Int
```

Or even better, use new types or data types generously:

```
checkTime :: Hours -> Minutes -> Seconds -> Bool
```

```
newtype Hours    = Hours Int
```

```
newtype Minutes = Minutes Int
```

```
newtype Seconds = SecondsInt
```

**Question:** what are the relative advantages and disadvantages of newtypes vs type synonyms?

- Comment top-level functions.
- Also comment tricky code.
- Write useful comments, avoid redundant comments!
- Use Haddock.

Keep in mind that Booleans are first-class values.

Negative examples:

```
f x | isSpace x == True = ...
```

```
if x then True else False
```



## Use (data)types!

- Whenever possible, define your own datatypes.
- Use `Maybe` or user-defined types to capture failure, rather than `error` or default values.
- Use `Maybe` or user-defined types to capture optional arguments, rather than passing undefined or dummy values.
- Don't use integers for enumeration types.
- By using meaningful names for constructors and types, or by defining type synonyms, you can make code more self-documenting.

## Use common library functions

- Don't reinvent the wheel. If you can use a `Prelude` function or a function from one of the basic libraries, then do not define it yourself.
- If a function is a simple instance of a higher-order function such as `map` or `foldr`, then use those functions.

- When defining functions via pattern matching, make sure you cover all cases.
- Try to use simple cases.
- Do not include unnecessary cases.
- Do not include unreachable cases.

## Avoid partial functions

- Always try to define functions that are total on their domain, otherwise try to refine the domain type.
- Avoid using functions that are partial.

## Negative example

```
if isJust x then 1 + fromJust x else 0
```

Use pattern matching!

## Use `let` instead of repeating complicated code

Write

```
let x = foo bar baz in x + x * x
```

rather than

```
foo bar baz + foo bar baz * foo bar baz
```

### Questions

- Is there a semantic difference between the two pieces of code?
- Could/should the compiler optimize from the second to the first version internally?

## Type-driven development

- Try to make your functions as generic as possible (why?).
- If you have to write a function of type `Foo -> Bar`, consider how you can destruct a `Foo` and how you can construct a `Bar`.
- When you tackle an unknown problem, think about its type first.
- You can insert 'holes' into your program to help develop your program piece by piece:

```
myFunction : (Foo -> Foo) -> Bar -> Baz
myFunction f b =
  let i = computeResult f _ in
  i + _
```

This is particularly helpful if you choose your types carefully!

**Questions?**