The Perfect Functional Programming Course

Peter Achten
Institute for Computing and Information Sciences
Radboud University Nijmegen

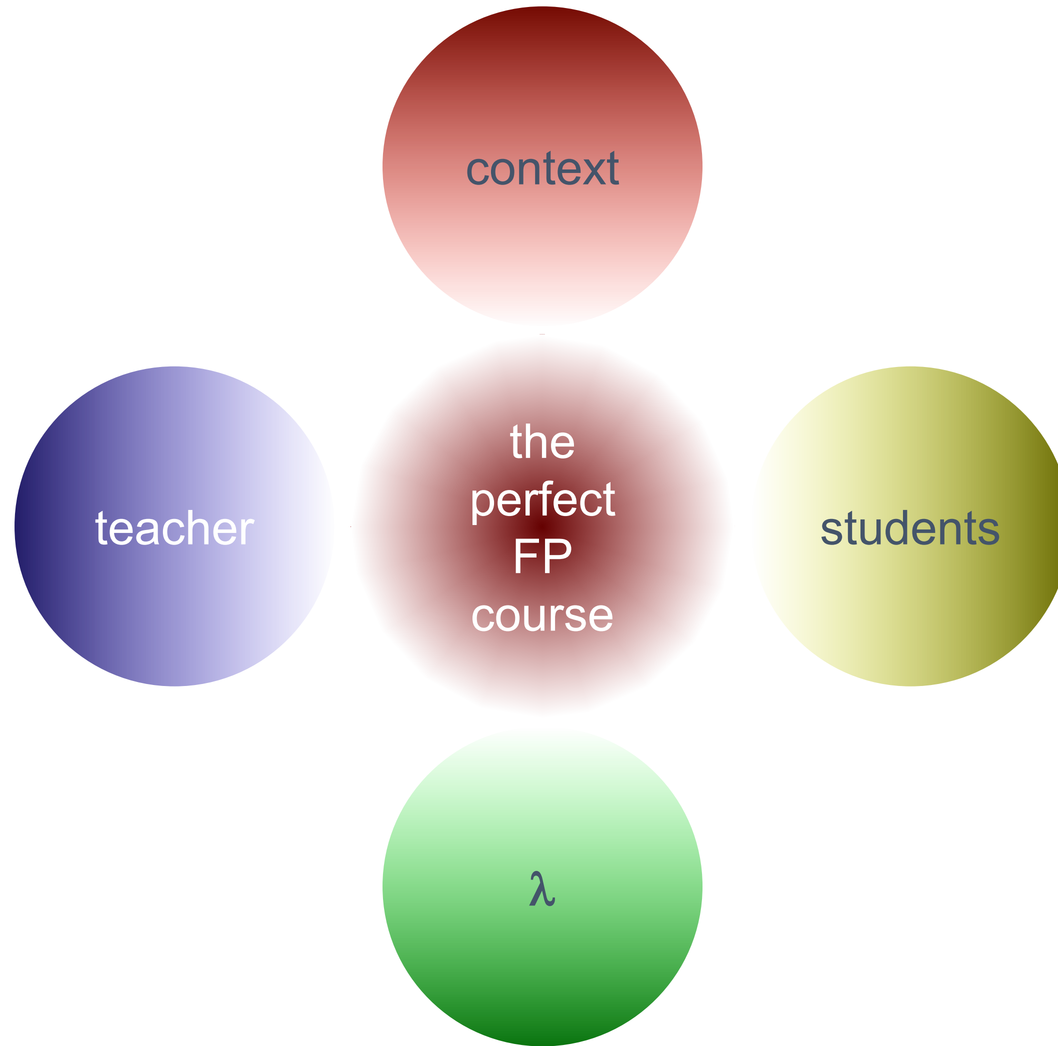
What this talk is about

- Share my experience in teaching functional programming
- A perfect (functional programming) course?
- Effective remedies to deal with hurdles that students face in functional programming
- Discuss engaging student project ideas
- Wrap up

Who am I

- Computing science education at Radboud University (1985 – 1991)
- PhD Thesis ‘Interactive Functional Programs’ (1996)
- Lecturer at Radboud University (2000 –)
- Teacher of programming language related courses:
  - imperative programming (bachelor 1st year introductory course, C++)
  - functional programming (bachelor 1st year, 2nd year versions, also for AI students, Clean)
  - compiler construction (master course, language agnostic)
  - advanced programming (master course, Clean, SaC)

The big picture


context
teacher
the
perfect
FP
course
students
λ

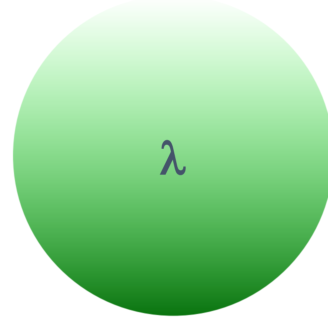
Context

- Placement in curriculum
- Course end terms
- Resources
  - study credits
  - study activities
  - educational forms
  - teaching assistants
  - programming language (support)
- Students
  - background
  - required knowledge
  - problem solving skill set

Students

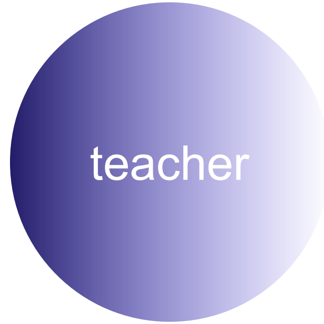
- “What’s in it for me?”
  - extend problem solving skill set
  - revisit known problems and solve them again
  - visit new problems and solve them
  - make reasoning about programs tractable
- Opportunities
  - relate to other courses
  - ‘make the connection’

λ

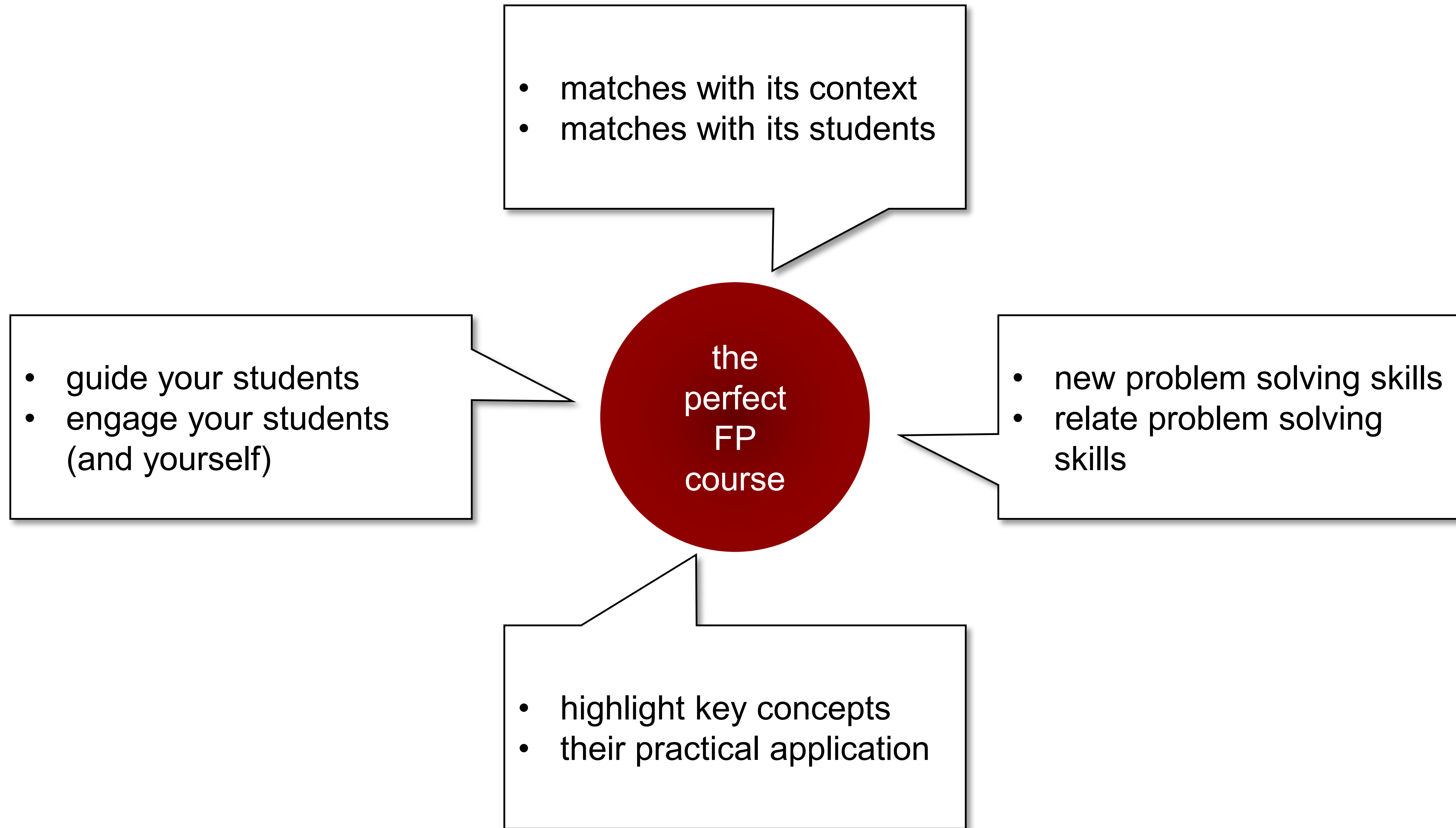


- “What’s in it for me?”
- Liberate programmers from the von Neumann style
- Show the elegance of programming with equations
  - expressive data types and type systems
  - pattern matching / case distinctions
  - higher-order functions
  - evaluation strategies
  - (equational) reasoning instead of debugging

Teacher



- “What’s in it for me?”
- Show students that only a few key concepts lead to a rich problem solving tool set
- Less exam grading time than with imperative / object oriented programming languages
- Engage your students and yourself
  - Why Functional Programming Matters [Hughes 1984]
  - Introduction to Functional Programming [Bird, Wadler 1988]
  - Purely Functional Data Structures [Okasaki 1998]
  - Functional Programs as Executable Specifications [Koopman 1999]
  - Haskell School of Expression [Hudak 2000]
  - How to Design Programs [Felleisen et al 2001]
  - SoccerFun [Achten 2008]
  - Animated Problem Solving [Morazán 2022]



matches with its context
matches with its students
guide your students
engage your students
(and yourself)
the perfect FP course
new problem solving skills
relate problem solving skills
highlight key concepts
their practical application

A perfect functional programming course?

A perfect functional programming course?

❶ Rewriting

Why:

- get into the basics of functional programming
- help students understand the mechanics
- seed for higher-order functions
- seed for evaluation strategies
- seed for equational reasoning

$$
\begin{aligned}
& 8 \div 2(2 + 2) \\
&= (8 \div 2) \bullet (2 + 2) \\
&= (\underline{8 \div 2}) \bullet (2 + 2) \\
&= 4 \bullet (\underline{2 + 2}) \\
&= \underline{4 \bullet 4} \\
&= 16
\end{aligned}
$$

A perfect functional programming course?

❶ Rewriting

❷ Type inference

Why:

- remove the mysticism around types
- help students understand type error messages
- polymorphic functions ‘for free’

A perfect functional programming course?

❶ Rewriting

❷ Type inference

❸ Custom data types

Why:

- intuitive way to model problem domains
- pattern matching on custom types
- recursive problem domains
- overloaded functions

lecture material

Modelling a problem domain: ancestry

- We want to model the following facts regarding a person's ancestry:
  1. a person has a name
  2. a person has a date of birth
  3. a person has a blood group
  4. a person has two biological parents
  5. a date consists of a year, month, day
  6. a biological parent is either a person or is unknown

ancestry is recursive

A perfect functional programming course?

❶ Rewriting

❷ Type inference

❸ Custom data types

❹ Lists and list comprehensions

Why:

- workhorse of functional programming
- pattern matching on lists
- lists and recursion
- list comprehensions
- seed for higher-order functions

A perfect functional programming course?

1. Rewriting
2. Type inference
3. Custom data types
4. Lists and list comprehensions
5. Higher-order functions

Why:

- key concept of functional programming
- key to abstraction and reusable code
- students get used to having functions as first-class citizens

A perfect functional programming course?

❶ Rewriting

❷ Type inference

❸ Custom data types

❹ Lists and list comprehensions

❺ Higher-order functions

❻ Evaluation strategies

Why:

- deepen understanding of rewriting
- discuss and compare evaluation strategies
- referential transparency
- seed for equational reasoning

A perfect functional programming course?

1. Rewriting
2. Type inference
3. Custom data types
4. Lists and list comprehensions
5. Higher-order functions
6. Evaluation strategies
7. (Equational) reasoning

Why:

- software correctness matters
- deepen abstraction
- deepen reusable code
- advocate reasoning over debugging

A perfect functional programming course?

1. Rewriting
2. Type inference
3. Custom data types
4. Lists and list comprehensions
5. Higher-order functions
6. Evaluation strategies
7. (Equational) reasoning
8. Case studies or projects

Why:

- engage students
- not necessarily at the end

A perfect functional programming course?

1. Rewriting
2. Type inference
3. Custom data types
4. Lists and list comprehensions
5. Higher-order functions
6. Evaluation strategies
7. (Equational) reasoning
8. Case studies or projects

Room for deepening:

- context, students, language, teacher
- programming pattern abstractions:
  - monads, functors, monoids, …
  - type constructor classes, gadt’s, …
- programming pragmatics:
  - interactive programs (console, GUI, graphics)
  - file / data processing
  - evaluation strategies and side-effects
- algorithms and data structures:
  - (binary) search trees, AVL trees, …
- compiler / interpreter / DSL:
  - parser combinators, grammars, interpreters
- …

Common hurdles for students in functional programming

Common hurdles for students in functional programming

- Types and type error messages
- List processing functions
- Get a grip on higher-order functions

Types and type error messages

- Students struggle with typing functions and type error messages
- Effective remedy is to explain typing as a logical puzzle

lecture material

Type inference is a logical puzzle

- Rules of the game (Clean book, section 1.5.7):
  - everything has one type
  - in a function the same argument has the same type in all function alternatives
  - all alternatives of a function have the same type
- Strategy to solve the puzzle:
  1. give each argument and result a fresh (lowercase) name-**?** pair and add them in a table
  2. check all pattern matches to extract type structure information
     - replace **?** with the found information and create fresh name-**?** pairs if necessary
     - replace in the function type and the table every occurrence of that name with the found information
  3. check all[1] sub-expressions to extract type information
     - replace **?** with the found information and create fresh name-**?** pairs if necessary
     - replace in the function type and the table every occurrence of that name with the found information
- Any remaining name-**?** pair represents a polymorphic type variable

---

[1] compiler has to check all sub-expressions to detect possible errors

lecture material

Type inference is a logical puzzle

```
module MySecondCleanProgram

import StdEnv


space 0 = ""
space n = " " +++ space (n-1)


pyramid` n ""   = ""
pyramid` n text = space n +++ text +++ "\n" +++
                  pyramid` (n+1) (text%(1,size text-2))
```

❶ ❷

```
(-)    :: Int Int -> Int
(+)    :: Int Int -> Int
(+++)  :: String  String    -> String
(%)    :: String (Int,Int)  -> String
size   :: String -> Int
```

step 1: fresh names for the arguments and result

lecture material

Type inference is a logical puzzle

```
module MySecondCleanProgram

import StdEnv

space :: a -> b
space 0 = ""                                  ❶
space n = " " +++ space (n-1)


pyramid` n ""   = ""                          ❷
pyramid` n text = space n +++ text +++ "\n" +++
                  pyramid` (n+1) (text%(1,size text-2))
```

```
(-)   :: Int Int -> Int
(+)   :: Int Int -> Int
(+++) :: String  String    -> String
(%)   :: String (Int,Int)  -> String
size  :: String -> Int
```

```
a  =  Int

b  =  ?
```

step 2: check all pattern matches

lecture material

Type inference is a logical puzzle

```
module MySecondCleanProgram

import StdEnv

space :: Int -> b
space 0 = ""
space n = " " +++ space (n-1)


pyramid` n ""   = ""
pyramid` n text = space n +++ text +++ "\n" +++
                  pyramid` (n+1) (text%(1,size text-2))
```

❶ ❷

```
(-)   :: Int Int -> Int
(+)   :: Int Int -> Int
(+++) :: String  String     -> String
(%)   :: String (Int,Int) -> String
size  :: String -> Int
```

```
a  =  Int

b  =  String
```

step 3: check all function alternatives

lecture material

Type inference is a logical puzzle

```
module MySecondCleanProgram

import StdEnv

space :: Int -> String
space 0 = ""
space n = " " +++ space (n-1)


pyramid` n ""   = ""
pyramid` n text = space n +++ text +++ "\n" +++
                  pyramid` (n+1) (text%(1,size text-2))
```

❷

```
(-)   :: Int Int -> Int
(+)   :: Int Int -> Int
(+++) :: String  String   -> String
(%)   :: String (Int,Int) -> String
size  :: String -> Int
```

step 1: fresh names for the arguments and result

lecture material

Type inference is a logical puzzle

```
module MySecondCleanProgram

import StdEnv

space :: Int -> String
space 0 = ""
space n = " " +++ space (n-1)

pyramid` :: a b -> c
pyramid` n ""   = ""
pyramid` n text = space n +++ text +++ "\n" +++
                  pyramid` (n+1) (text%(1,size text-2))
```

❷

```
(-)   :: Int Int -> Int
(+)   :: Int Int -> Int
(+++) :: String  String    -> String
(%)   :: String (Int,Int) -> String
size  :: String -> Int
```

```
a  =  ?

b  =  ?

c  =  ?
```

lecture material

Type inference is a logical puzzle

```
module MySecondCleanProgram

import StdEnv

space :: Int -> String
space 0 = ""
space n = " " +++ space (n-1)

pyramid` :: a b -> c
pyramid` n ""   = ""
pyramid` n text = space n +++ text +++ "\n" +++
                  pyramid` (n+1) (text%(1,size text-2))
```

❷

```
(-)   :: Int Int -> Int
(+)   :: Int Int -> Int
(+++) :: String String    -> String
(%)   :: String (Int,Int) -> String
size  :: String -> Int
```

```
a  =  ?

b  =  String

c  =  ?
```

step 2: check all pattern matches

lecture material

Type inference is a logical puzzle

```
module MySecondCleanProgram

import StdEnv

space :: Int -> String
space 0 = ""
space n = " " +++ space (n-1)

pyramid` :: a String -> c
pyramid` n ""    = ""
pyramid` n text = space n +++ text +++ "\n" +++
                  pyramid` (n+1) (text%(1,size text-2))
```

❷

```
(-)   :: Int Int -> Int
(+)   :: Int Int -> Int
(+++) :: String String    -> String
(%)   :: String (Int,Int) -> String
size  :: String -> Int
```

```
a  =  ?

b  =  String

c  =  String
```

step 3: check all function alternatives

lecture material

Type inference is a logical puzzle

```
module MySecondCleanProgram

import StdEnv

space :: Int -> String
space 0 = ""
space n = " " +++ space (n-1)

pyramid` :: a String -> String
pyramid` n ""    = ""
pyramid` n text = space n +++ text +++ "\n" +++
                  pyramid` (n+1) (text%(1,size text-2))
```

❷

```
(-)   :: Int Int -> Int
(+)   :: Int Int -> Int
(+++) :: String String    -> String
(%)   :: String (Int,Int) -> String
size  :: String -> Int
```

```
a  =  Int

b  =  String

c  =  String
```

step 3: check all function alternatives

Radboud University

lecture material

Type inference is a logical puzzle

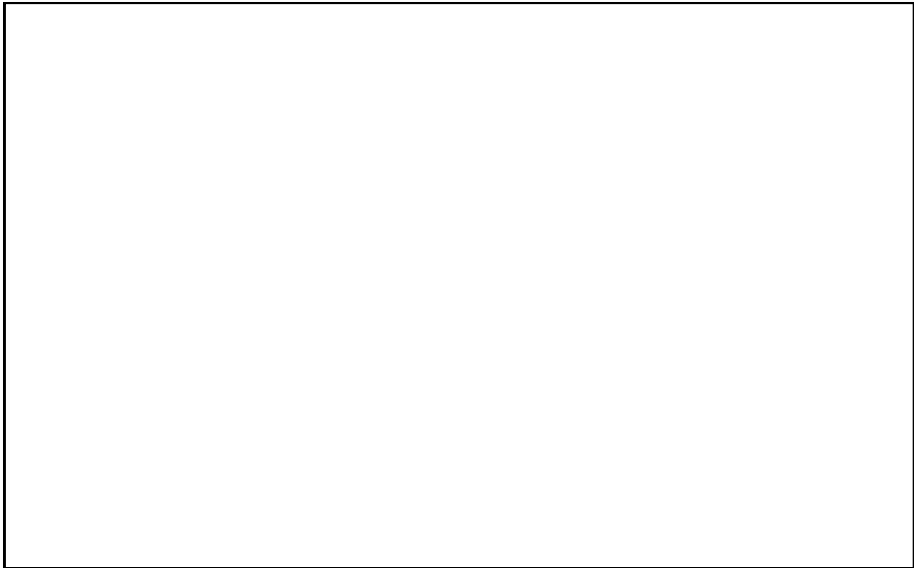
```
module MySecondCleanProgram

import StdEnv

space :: Int -> String
space 0 = ""
space n = " " +++ space (n-1)

pyramid` :: Int String -> String
pyramid` n ""   = ""
pyramid` n text = space n +++ text +++ "\n" +++
                  pyramid` (n+1) (text%(1,size text-2))
```

```
(-)   :: Int Int -> Int
(+)   :: Int Int -> Int
(+++) :: String  String    -> String
(%)   :: String (Int,Int) -> String
size  :: String -> Int
```



lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

❶ first (x,y) = x

❷ swap (x,y) = (y,x)
```

step 1: fresh names for the arguments and result

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: a -> b
❶ first (x,y) = x


❷ swap (x,y) = (y,x)
```

```
a = ?

b = ?
```

step 1: fresh names for the arguments and result

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: a -> b
first (x,y) = x        ❶

swap (x,y) = (y,x)     ❷
```

```
a  =  ?

b  =  ?
```

step 2: check all pattern matches

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: a -> b
first (x,y) = x        ❶


swap (x,y) = (y,x)     ❷
```

```
a  =  (c, d)

b  =  ?

c  =  ?

d  =  ?
```

step 2: check all pattern matches

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> b
first (x,y) = x        ❶


swap (x,y) = (y,x)     ❷
```

```
a  =  (c, d)

b  =  ?

c  =  ?

d  =  ?
```

step 2: check all pattern matches

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> b
first (x,y) = x

swap (x,y) = (y,x)
```

❶ first (x,y) = x

❷ swap (x,y) = (y,x)

```
a = (c, d)
b = ?
c = ?
d = ?
```

step 3: check all function alternatives

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> b
first (x,y) = x

swap (x,y) = (y,x)
```

❶ first (x,y) = x

❷ swap (x,y) = (y,x)

```
a  =  (c, d)
b  =  c
c  =  ?
d  =  ?
```

step 3: check all function alternatives

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> c
first (x,y) = x


swap (x,y) = (y,x)
```

❷

step 1: fresh names for the arguments and result

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> c
first (x,y) = x

swap :: a -> b
swap (x,y) = (y,x)
```

❷

```
a  =  ?

b  =  ?
```

step 1: fresh names for the arguments and result

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> c
first (x,y) = x

swap :: a -> b
swap (x,y) = (y,x)
```

❷

```
a  =  ?

b  =  ?
```

step 2: check all pattern matches

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> c
first (x,y) = x

swap :: a -> b
swap (x,y) = (y,x)
```

❷

```
a  =  (c,d)

b  =  ?

c  =  ?

d  =  ?
```

step 2: check all pattern matches

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> c
first (x,y) = x

swap :: (c,d) -> b
swap (x,y) = (y,x)
```

❷

```
a  =  (c,d)

b  =  ?

c  =  ?

d  =  ?
```

step 2: check all pattern matches

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> c
first (x,y) = x

swap :: (c,d) -> b
swap (x,y) = (y,x)
```

❷

```
a  =  (c,d)

b  =  ?

c  =  ?

d  =  ?
```

step 3: check all function alternatives

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> c
first (x,y) = x

swap :: (c,d) -> b
swap (x,y) = (y,x)
```

❷

```
a  =  (c,d)
b  =  (e,f)
c  =  ?
d  =  ?
e  =  ?
f  =  ?
```

step 3: check all function alternatives

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> c
first (x,y) = x

swap :: (c,d) -> (e,f)
swap (x,y) = (y,x)
```

❷

```
a  =  (c,d)

b  =  (e,f)

c  =  ?

d  =  ?

e  =  ?

f  =  ?
```

step 3: check all function alternatives

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> c
first (x,y) = x

swap :: (c,d) -> (e,f)
swap (x,y) = (y x)
```

❷

```
a  =  (c,d)

b  =  (e,f)

c  =  ?

d  =  ?

e  =  d

f  =  ?
```

step 3: check all function alternatives

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> c
first (x,y) = x

swap :: (c,d) -> (d,f)
swap (x,y) = (y x)
```

❷

```
a  =  (c,d)
b  =  (d,f)
c  =  ?
d  =  ?
e  =  d
f  =  ?
```

step 3: check all function alternatives

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> c
first (x,y) = x

swap :: (c,d) -> (d,f)
swap (x,y) = (y,x)
```

❷

```
a  =  (c,d)

b  =  (d,f)

c  =  ?

d  =  ?

e  =  d

f  =  c
```

step 3: check all function alternatives

lecture material

Type inference with tuples

```
implementation module MyTupleFunctions

first :: (c,d) -> c
first (x,y) = x

swap :: (c,d) -> (d,c)
swap (x,y) = (y,x)
```

❷

```
a  =  (c,d)

b  =  (d,c)

c  =  ?

d  =  ?

e  =  d

f  =  c
```

step 3: check all function alternatives

List processing functions

- Students struggle to orchestrate list pattern-matching and list recursion, in particular when lists-of-lists are involved
- Effective remedy is to provide a design recipe (inspired by HtDP school)

lecture material

How to tackle recursion on lists?

1. Specify the name and write down a few cases:
```
my_function …          [] … = …
my_function …        [x1] … = …
my_function …     [x2,x1] … = …
my_function … [x3,x2,x1] … = …
```
2. Specify the type of the function:
```
my_function :: … [ … ] … -> …
```
3. Make a case distinction based on the structure of your recursive data type.
   a) The list is empty (base case)
   ```
   my_function … [] …       = …
   ```
   b) The list is not empty (recursive case)
   ```
   my_function … [… : …] … = …
   ```

depending on the problem, you may require more base cases

depending on the problem, you may require more recursive cases

lecture material

How to tackle recursion on lists?

1. Specify the name and write down a few cases:

```
my_function …           [] … = …
my_function …         [x1] … = …
my_function …      [x2,x1] … = …
my_function … [x3,x2,x1] … = …
```

2. Specify the type of the function:

```
my_function :: … [ … ] … -> …
```

3. Make a case distinction based on the structure of your recursive data type.
   a) The list is empty (base case)

```
my_function … [] …      = …
```

   b) The list is not empty (recursive case)

```
my_function … [… :xs] … = r
where
     r1 = my_function … xs …
     r  = … r1 …
```

assume you have the result r1 for the smaller list xs

use r1 to solve problem for the argument list

lecture material

Length of a list

1. Specify the name and write down a few cases:

```
• length          [] = 0
• length        [x1] = 1
• length     [x2,x1] = 2
• length  [x3,x2,x1] = 3
```

2. Specify the type of the function:

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

3. Make a case distinction based on the structure of your recursive data type.
   a) the list is empty

```
length [] = 0
```

   b) the list is not empty

```
length [x : xs] = r
where
     r1 = length xs
     r  = 1 + r1
```

a bit verbose, transform

```
length :: [a] -> Int
length []         = 0
length [_ : xs] = 1 + length xs
```

use _ to document that the head is not used in this alternative

lecture material

Initial segment of a list

- We wish to find the initial segment of a list (list without last element):
  - ```
    init          [] = abort "init of []"
    init        [x1] = []
    init     [x2,x1] = [x2]
    init  [x3,x2,x1] = [x3,x2]
    ```
  - ...

this function has 2 base cases

find the symmetry
    ```

lecture material

Initial segment of a list

- We wish to find the initial segment of a list (list without last element):
  - `init            [] = abort "init of []"`
  - `init          [x1] =            []`
  - `init       [x2,x1] =       [x2:[]]`
  - `init    [x3,x2,x1] = [x3:[x2:[]]]`
  - ...

this function has 2 base cases

find the symmetry

apply the steps...

```
init :: [a] -> [a]
init []         = abort "init of []"
init [_]        = []
init [x : xs] = [x : init xs]
```

Higher-order functions

- Students find it hard to get a grip on higher-order functions
- Effective remedy is to show that a function ‘is just another’ parameter

lecture material

“It is also the goal for which functional programmers must strive - smaller and simpler and more general modules, glued together with the new glues we shall describe.”

John Hughes, “Why Functional Programming Matters”, 1989

lecture material

Why smaller and simpler and more general functions matter…

- Smaller modules:
  - a module should concentrate on one piece of functionality
  - less prone to errors
  - easier to understand and reason about
- Simpler modules:
  - less prone to errors
  - easier to combine with other modules
- More general modules:
  - solve a recurring problem once and for all instead of ad-hoc
  - saves development time
  - errors rise from the application, not the implementation

Functional programming:

- functions are compositional modules
- functions can be passed as parameter
- new functions can be computed

Higher-order functions (HOF)

lecture material

Many functions look very similar

```
• increase xs = [ x + 1                   \\   x    <- xs ]
• decrease xs = [ x - 1                   \\   x    <- xs ]
• squares  xs = [ x * x                   \\   x    <- xs ]
• avgs     xs = [ (x + y) / (fromInt 2)   \\ (x,y)  <- xs ]
```

```
inc x      = x + 1
dec x      = x - 1
square x   = x * x
avg (x,y)  = (x + y) / (fromInt 2)
```

lecture material

Many functions look very similar

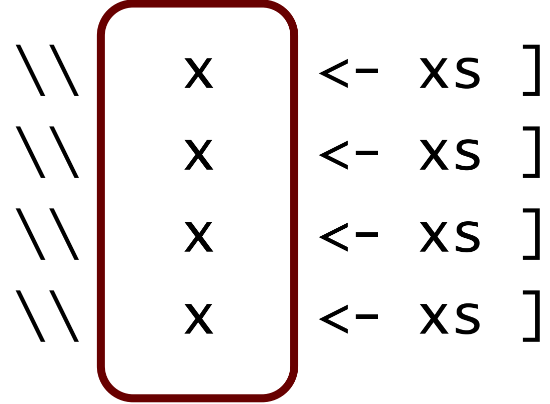
```
• increase xs = [ inc    x     \\  x     <- xs ]
• decrease xs = [ dec    x     \\  x     <- xs ]
• squares  xs = [ square x     \\  x     <- xs ]
• avgs     xs = [ avg    (x,y) \\  (x,y) <- xs ]
```

```
inc x      = x + 1
dec x      = x - 1
square x   = x * x
avg (x,y)  = (x + y) / (fromInt 2)
```

lecture material

Many functions look very similar

- `increase xs = [ inc x`
- `decrease xs = [ dec x`
- `squares xs = [ square x`
- `avgs xs = [ avg x`



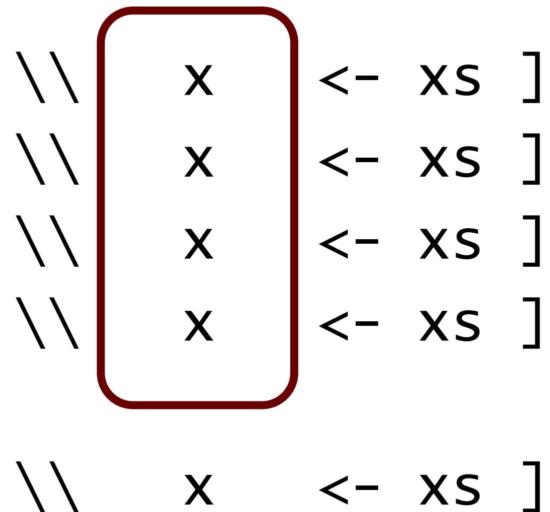

```
inc x        = x + 1
dec x        = x - 1
square x     = x * x
avg (x,y)    = (x + y) / (fromInt 2)
```

lecture material

Many functions look very similar

```
• increase xs = [ inc    x \\ x <- xs ]
• decrease xs = [ dec    x \\ x <- xs ]
• squares  xs = [ square x \\ x <- xs ]
• avgs     xs = [ avg    x \\ x <- xs ]

• map    f xs = [ f      x \\ x <- xs ]
```



```
inc x      = x + 1
dec x      = x - 1
square x   = x * x
avg (x,y)  = (x + y) / (fromInt 2)
```

lecture material

Many functions look very similar

```
• increase xs = map inc    xs
• decrease xs = map dec    xs
• squares  xs = map square xs
• avgs     xs = map avg    xs

• map   f  xs = [ f x \\ x <- xs ]
```

```
inc x      = x + 1
dec x      = x - 1
square x   = x * x
avg (x,y)  = (x + y) / (fromInt 2)
```

- all applications of the same computational pattern: emphasize similarity
- ad-hoc part is identified: emphasize difference

- module with single purpose, and customizable due to HOF

- smaller and simpler functions
- easier to maintain
- can be reused in other context

lecture material

Many recursive functions look very similar

```
sum     :: [a] -> a | zero, + a
sum     []         = zero
sum     [x : xs] = x + sum xs

product :: [a] -> a | one, * a
product []         = one
product [x : xs] = x * product xs

and     :: [Bool] -> Bool
and     []         = True
and     [x : xs] = x && and xs

or      :: [Bool] -> Bool
or      []         = False
or      [x : xs] = x || or xs
```

```
sum     xs = foldr (+)  zero  xs

product xs = foldr (*)  one   xs

and     xs = foldr (&&) True  xs

or      xs = foldr (||) False xs
```

```
foldr :: (a b -> b) b [a] -> b
foldr op e []       = e
foldr op e [x:xs] = op x (foldr op e xs)
```

Higher-order functions

- Students find it hard to get a grip on higher-order functions
- Effective remedy is to show that a function 'is just another' parameter
- Using higher-order functions effectively is an acquired taste
  - in the 'Segments: An alternative rainfall problem'[1] study, 32% of solutions use higher-order functions

[1] ACHTEN, P. (2021). Segments: An alternative rainfall problem. *Journal of Functional Programming, 31*, E23. doi:10.1017/S0956796821000216

Engaging projects

SoccerFun[1]

- Implement a team of football players (or rather, their brain functions)
- Brain function:

```
:: (BrainInput,memory) -> (BrainOutput,memory)
```

- Brain input:
  - referee information
  - state of the ball
  - all other field players
  - player
- Brain output:
  - move, feint
  - kick ball, head ball
  - gain ball, catch ball, tackle

[1] Achten, P. (2008) Teaching functional programming with Soccer-Fun. In *Proceedings of the 1st Workshop on Functional and Declarative Programming in Education, FDPE'08*. Huch, F. & Parkin, A. (eds). Victoria, BC, Canada: ACM Press, pp. 61-72

SoccerFun

- Can be tuned for full-fledged implementation to minimalistic approach
- Minimalistic approach can already be done after custom data types
- Recommended position is after lists / higher-order functions
- Students can concentrate on different aspects:
  - best written / structured code
  - best performing team
- Clever, unexpected solutions!

General guidelines

General guidelines

- Know the context and your students
- Construct a narrative for the course
- Anticipate hurdles and eliminate / lower them before they occur
- Don’t forget yourself

- Thank you for your attention!