



# Functional Programming

## Lecture 1

Cezary Kaliszyk   Jonas Schöpf  
Christian Sternagel   Vincent van Oostrom

Department of Computer Science

## Lecture (VO 2)

- LV-Number: 703024
- lecturer: Christian Sternagel
- course website:  
<http://cl-informatik.uibk.ac.at/teaching/ws18/fp/>  
(slides are available online)
- consultation hours: Friday 14:00–15:30 in 3M03
- online registration required before 23:59 on November 30
- grading: written exam (closed book)
  - 1st exam on February 1, 2019
  - online registration required before 23:59 on January 18, 2019



## Exercises (PS 1)

- LV-Number: 703025
- group 1 Vincent van Oostrom Friday 08:15 – 09:00 HS 11
- group 2 Cezary Kaliszky Friday 09:15 – 10:00 HS 11
- group 3 Christian Sternagel Friday 10:15 – 11:00 HS 11
- group 4 Jonas Schöpf Friday 11:15 – 12:00 HSB 7
- group 5 Jonas Schöpf Friday 10:15 – 11:00 HSB 7
- if you want to change to group 5, please contact CS (indicating valid justification might increase priority)
- grading: weekly exercises
- solved exercises must be marked in OLAT (deadline: 7:30 a.m. before PS on Friday)
- exercises start on October 12



## Schedule

lecture 1	October	5	lecture 7	December	7
lecture 2	October	12	lecture 8	December	14
lecture 3	November	9	lecture 9	January	11
lecture 4	November	16	lecture 10	January	18
lecture 5	November	23	lecture 11	January	25
lecture 6	November	30	1st exam	February	1

## Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, **first steps**, guarded recursion, **Haskell introduction**, higher-order functions, **historical overview**, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

# Overview

- History
- Notions
- A Taste of Haskell
- First Steps



1924

**Moses Schönfinkel:**  
combinatory logic



1936

**Alonzo Church:**  
 $\lambda$ -calculus

1941

**Z3:** 1st programmable, fully automatic computing machine



1966

**Peter Landin:**  
Iswim



1988

**David Turner:**  
Miranda



**Paul Hudak and Philip Wadler:**  
Haskell

1918

2018

1937

**Alan Turing:**  
turing machines



1930

**Haskell Curry:**  
combinatory logic



1958



**John McCarthy:**  
LISP

1977

**John Backus:**  
FP



1984

**Robin Milner:**  
LCF, Standard ML



2003

2005

**Don Syme:** F#

**Martin Odersky:**  
Scala



2010

Haskell2010

## (Program) State

- variables point to storage locations in memory
- **state** is content of variables in scope at given execution point

### Example – Assignment

after  $x := 10$ , location  $x$  has content 10 (state might have changed)

### Side Effects

a function or expression has **side effects** if it modifies state

### Example – $\sum_{i=0}^n i$

```
count := 0
```

```
total := 0
```

```
while count < n
```

```
    count := count + 1
```

```
    total := total + count
```



## Example – $\sum_{i=0}^n i$

the Haskell way of summing up the numbers from 0 to n is

```
sum [0..n]
```

- `[0..4]` generates list `[0,1,2,3,4]`
- `sum` is predefined function, summing up elements of a list

## Example – Defining Functions

- `[m..n]` computes range of numbers from m to n

```
range m n =
```

```
  if m > n then []
```

```
  else m : range (m + 1) n
```

- `sum xs` computes sum of elements in `xs`

```
mySum [] = 0
```

```
mySum (x:xs) = x + mySum xs
```

## Pure Functions

a function is **pure** if it always returns same result on same input

## Counterexample – Random Numbers

the C function `rand` (producing random numbers) is not pure

```
rand() = 0
```

```
rand() = 10
```

```
rand() = 42
```

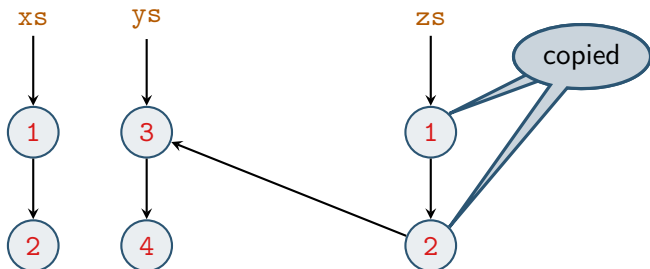
# Immutable Data

data that does not change after initial creation

## Example – Immutable Linked Lists

append elements of  $ys$  to  $xs$

- consider two linked lists  $xs = [1, 2]$  and  $ys = [3, 4]$
- after concatenation  $zs = xs ++ ys$



## Recursion

a function (definition) is **recursive** if it refers to itself

### Example – Factorial Numbers

```
factorial n =  
  if n < 2 then 1  
  else n * factorial (n - 1)
```

## Evaluating Functions by Hand (aka Equational Reasoning)

- functions are defined by equations and pattern matching
- general idea: “replace equals by equals”

### Example – mySum

given the two equations

$$\text{mySum } [] = 0 \tag{1}$$

$$\text{mySum } (x:xs) = x + \text{mySum } xs \tag{2}$$

pattern: empty list

pattern: list with “head”  $x$  and “tail”  $xs$

we evaluate `mySum [1,2,3]` like

$$\begin{aligned} \text{mySum } [1,2,3] &= 1 + \text{mySum } [2,3] && \text{using (2)} \\ &= 1 + (2 + \text{mySum } [3]) && \text{using (2)} \\ &= 1 + (2 + (3 + \text{mySum } [])) && \text{using (2)} \\ &= 1 + (2 + (3 + 0)) && \text{using (1)} \\ &= 6 && \text{by def. of +} \end{aligned}$$

## Haskell

- is a pure language (only allowing “explicit” side effects)
- functions are defined by equations and pattern matching

### Example – Quicksort

- sort list of elements smaller than or equal to  $x$
- sort list of elements larger than  $x$
- insert  $x$  in between

```

qsort []      = []
qsort (x:xs) = qsort le ++ [x] ++ qsort gt
  where
    le = [a | a <- xs, a <= x] -- list comprehension
    gt = [b | b <- xs, b > x]

```

## Haskell on the Web

- main entry point [www.haskell.org](http://www.haskell.org)
- most widely used Haskell compiler: GHC
- with interpreter GHCi

## Starting the Interpreter (GHCi)

```
$ ghci
```

```
GHCi, version 8.2.2: http://www.haskell.org/ghc/
```

```
?:? for help
```

```
...
```

```
Prelude>
```

# The Standard Prelude

on startup GHCi loads the “Prelude,” importing many standard functions

## Examples

- arithmetic: +, -, \*, /, ^, mod, div
- lists
  - `drop n xs` drop first `n` elements from list `xs`
  - `head xs` extract first element from list `xs`
  - `length xs` number of elements in list `xs`
  - `product xs` multiply elements of list `xs`
  - `reverse xs` reverse list `xs`
  - `sum xs` sum up elements of list `xs`
  - `tail xs` obtain list `xs` without its first element
  - `take n xs` take first `n` elements from list `xs`
- note: in code examples Prelude functions are colored green and others blue; variables are colored dark orange



## Function Application

- in mathematics: function application is denoted by enclosing arguments in parentheses, whereas multiplication of two arguments is often implicit (by juxtaposition)
- in Haskell: reflecting its primary status, function application is denoted silently (by juxtaposition), whereas multiplication is denoted explicitly by `*`

## Examples

Mathematics	Haskell
$f(x)$	<code>f x</code>
$f(x, y)$	<code>f x y</code>
$f(g(x))$	<code>f (g x)</code>
$f(x, g(y))$	<code>f x (g y)</code>
$f(x)g(y)$	<code>f x * g y</code>
$f(a, b) + cd$	<code>f a b + c * d</code>

## Haskell Scripts

- define new functions inside **scripts**
- text file containing definitions
- common suffix `.hs`

### My First Script – `test.hs`

- set editor from inside GHCi `:set editor vim`
- start editor `:edit test.hs` and type

```
double x      = x + x
quadruple x = double (double x)
```
- load script

```
Prelude> :load test.hs
```

```
[1 of 1] Compiling Main ( test.hs, interpreted )
```

```
Ok, modules loaded: Main.
```

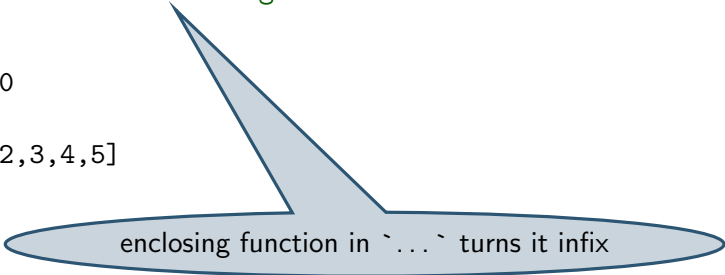
```
*Main>
```

## Interpreter Commands

Command	Meaning
:load <i>&lt;filename&gt;</i>	load script <i>&lt;filename&gt;</i>
:reload	reload current script
:edit <i>&lt;filename&gt;</i>	edit script <i>&lt;filename&gt;</i>
:edit	edit current script
:type <i>&lt;expression&gt;</i>	show type of <i>&lt;expression&gt;</i>
:set <i>&lt;property&gt;</i>	change various settings
:show <i>&lt;info&gt;</i>	show various information
:! <i>&lt;command&gt;</i>	execute <i>&lt;command&gt;</i> in shell
:?	show help text
:quit	bye-bye!

## Example Session

```
> :load test.hs
> quadruple 10
40
> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
> :edit test.hs
factorial n = product [1..n]
average ns = sum ns `div` length ns
> :reload
> factorial 10
3628800
> average [1,2,3,4,5]
3
```



enclosing function in `...` turns it infix

## Naming Requirements

names of functions and their arguments have to conform to following syntax

$\langle \textit{lower} \rangle ::= a \mid \dots \mid z$

$\langle \textit{upper} \rangle ::= A \mid \dots \mid Z$

$\langle \textit{digit} \rangle ::= 0 \mid \dots \mid 9$

$\langle \textit{name} \rangle ::= (\langle \textit{lower} \rangle \mid \_)(\langle \textit{lower} \rangle \mid \langle \textit{upper} \rangle \mid \langle \textit{digit} \rangle \mid ' \mid \_)^*$

choice

zero or more times

## Reserved Names

`case class data default deriving do else foreign if import in  
infix infixl infixr instance let module newtype of then type  
where _`

## Examples

`myFun fun1 arg_2 x'`

## The Layout Rule

- items that start in same column are grouped together
- by increasing indentation, single item may span multiple lines
- groups end at EOF or when indentation decreases
- script content is group, start nested group by **where**, **let**, **do**, or **of**
- **ignore layout**: enclose groups in '{' and '}' and separate items by ';' ;'

## Examples

with layout:

```
main =
  let x = 1
      y = 1
  in
  putStrLn (take
    (x+y) (zs++us))
  where
    zs = []
    us = "abc"
```

without layout:

```
main =
  let { x = 1; y = 1 } in
  putStrLn (take (x+y) (zs++us))
  where { zs = []; us = "abc" }
```

## Comments

there are two kinds of comments

- single-line comments: starting with `--` and extending to EOL
- multi-line comments: enclosed in `{-` and `-}`

## Examples

-- Factorial of a positive number:

```
factorial n = product [1..n]
```

-- Average of a list of numbers:

```
average ns = sum ns `div` length ns
```

{- currently not used

```
double x    = x + x
```

```
quadruple x = double (double x)
```

```
-}
```

## Homework (for October 12th)

1. Read [http://haskell.org/haskellwiki/Functional\\_programming](http://haskell.org/haskellwiki/Functional_programming) and [http://haskell.org/haskellwiki/Haskell\\_in\\_5\\_steps](http://haskell.org/haskellwiki/Haskell_in_5_steps).
2. Work through lessons 1 to 3 on <http://tryhaskell.org/>.
3. Find, explain, and correct the 4 errors in the following code:

```
x = mod length data Y
  where
    { data = [1..10] Y = 5 }
```

4. Implement a function `nth`, where `nth xs i` yields the `i`s element of the list `xs`, in terms of the Prelude functions from this lecture.  
**Example:** `nth ["a","b","c"] 1 = "b"`
5. Implement a function `fromTo`, where `fromTo xs i j` yields the part of `xs` between positions `i` and `j`, in terms of the Prelude functions from this lecture.  
**Example:** `fromTo ["a","b","c","d"] 1 2 = ["b","c"]`
6. Use recursion to implement a function `allTrue` that, given a list of boolean values, checks whether they are all true.  
**Example:** `allTrue [True,False,True] = False`