Functional Programming

Christian Sternagel Harald Zankl Evgeny Zuenko

Department of Computer Science
University of Innsbruck

WS 2017/2018

Lecture 1

Organization

Lecture

- LV-Number 703024
- VO 2
- http://cl-informatik.uibk.ac.at/teaching/ws17/fp/
- slides are also available online
- office hours: Friday 14:15 – 15:45 in 3M03
- online registration required before 23:59 on November 30

Lecture

- LV-Number 703024
- VO 2
- `http://cl-informatik.uibk.ac.at/teaching/ws17/fp/`
- slides are also available online
- office hours: Friday 14:15 – 15:45 in 3M03
- online registration required before 23:59 on November 30

Lecture

- LV-Number 703024
- VO 2
- http://cl-informatik.uibk.ac.at/teaching/ws17/fp/
- slides are also available online
- office hours: Friday 14:15 – 15:45 in 3M03
- online registration required before 23:59 on November 30
- grading: written exam (closed book)
  - 1st exam on February 2, 2018
  - registration starts 5 weeks before exam
  - registration closes 2 weeks before exam

Exercises

- LV-Number 703025
- PS 1
- group 1 Christian Sternagel Friday 10:15 – 11:00 HS 11
  group 2 Harald Zankl Friday 11:15 – 12:00 HS 11
  group 3 Harald Zankl Friday 12:15 – 13:00 HS 11
  group 4 Evgeny Zuenko Friday 13:15 – 14:00 HS 11
- online registration required before 23:59 on September 21
- grading: 1 test (January 12, 2018) + weekly exercises
- exercises start on October 20

Exercises

- LV-Number 703025
- PS 1
group 1	Christian Sternagel	Friday 10:15 – 11:00	HS 11
group 2	Harald Zankl	Friday 11:15 – 12:00	HS 11
group 3	Harald Zankl	Friday 12:15 – 13:00	HS 11
group 4	Evgeny Zuenko	Friday 13:15 – 14:00	HS 11
- online registration required before 23:59 on September 21
- grading: 1 test (January 12, 2018) + weekly exercises
- exercises start on October 20

Exercises

- LV-Number 703025
- PS 1
- group 1 Christian Sternagel Friday 10:15 – 11:00 HS 11
  group 2 Harald Zankl Friday 11:15 – 12:00 HS 11
  group 3 Harald Zankl Friday 12:15 – 13:00 HS 11
  group 4 Evgeny Zuenko Friday 13:15 – 14:00 HS 11
- online registration required before 23:59 on September 21
- grading: 1 test (January 12, 2018) + weekly exercises
- exercises start on October 20

Exercises

- LV-Number 703025
- PS 1
group 1	Christian Sternagel	Friday 10:15 – 11:00	HS 11
group 2	Harald Zankl	Friday 11:15 – 12:00	HS 11
group 3	Harald Zankl	Friday 12:15 – 13:00	HS 11
group 4	Evgeny Zuenko	Friday 13:15 – 14:00	HS 11
- online registration required before 23:59 on September 21
- grading: 1 test (January 12, 2018) + weekly exercises
- exercises start on October 20

Schedule

week 1	October	6	week 8	December	1
week 2	October	20	week 9	December	15
week 3	October	27	week 10	January	12
week 4	November	3	week 11	January	19
week 5	November	10	week 12	January	26
week 6	November	17	1st exam	February	2
week 7	November	24			

Schedule

week 1	October	6	week 8	December	1
week 2	October	20	week 9	December	15
week 3	October	27	PS test	January	12
week 4	November	3	week 11	January	19
week 5	November	10	week 12	January	26
week 6	November	17	1st exam	February	2
week 7	November	24			

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, implementing a type checker, 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

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, implementing a type checker, 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

History

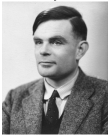
1936 **Alonzo Church:**
λ-calculus

1917 2017

1936 **Alonzo Church:** λ-calculus

1917 2017

1937 **Alan Turing:** turing machines



1936

Alonzo Church:
λ-calculus

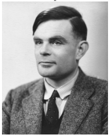
1924

Moses Schönfinkel:
combinatory logic

1917 2017

1937

Alan Turing:
turing machines

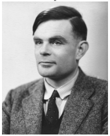


1936 **Alonzo Church:**
λ-calculus

1924 **Moses Schönfinkel:**
combinatory logic

1917 — 2017

1937 **Alan Turing:**
turing machines



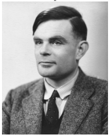
1930 **Haskell Curry:**
combinatory logic

1917 — 2017

1924 **Moses Schönfinkel:** combinatory logic

1930 **Haskell Curry:** combinatory logic

1936 **Alonzo Church:** λ-calculus



1937 **Alan Turing:** turing machines

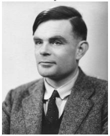
1941 **Z3:** 1st programmable, fully automatic computing machine

1917 — 2017

1924 **Moses Schönfinkel:** combinatory logic

1930 **Haskell Curry:** combinatory logic

1936 **Alonzo Church:** λ-calculus



1937 **Alan Turing:** turing machines

1941 **Z3:** 1st programmable, fully automatic computing machine

1950 **John McCarthy:** LISP

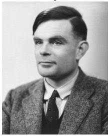
1917 — 2017

1924 **Moses Schönfinkel:** combinatory logic

1930 **Haskell Curry:** combinatory logic

1936 **Alonzo Church:** λ-calculus

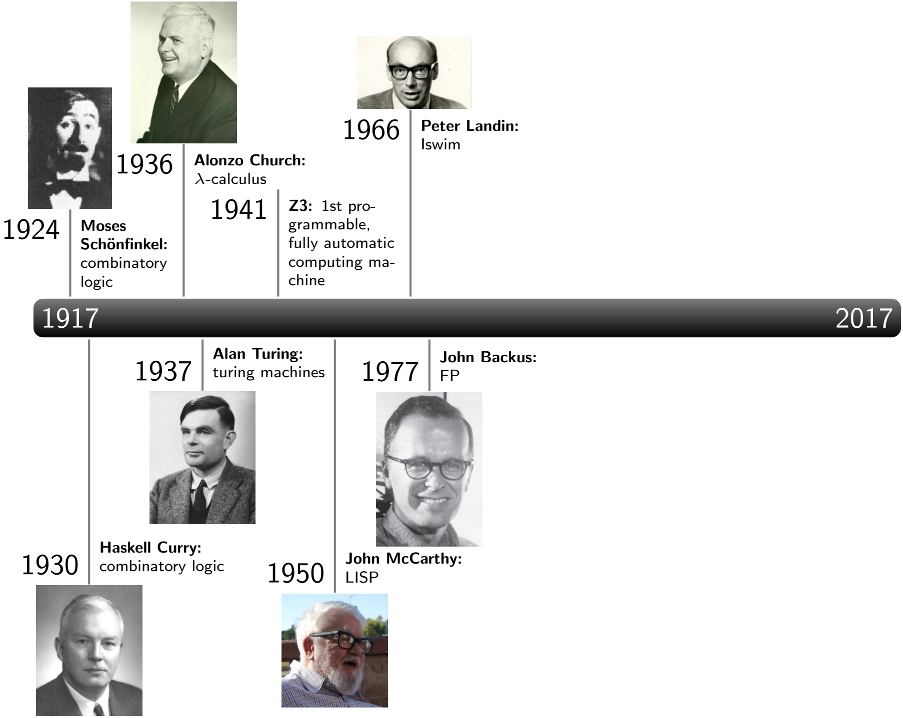
1937 **Alan Turing:** turing machines



1941 **Z3:** 1st programmable, fully automatic computing machine

1950 **John McCarthy:** LISP

1966 **Peter Landin:** Iswim


1917
2017
1924
Moses Schönfinkel: combinatory logic
1936
Alonzo Church: λ-calculus
1941
Z3: 1st programmable, fully automatic computing machine
1966
Peter Landin: Iswim
1930
Haskell Curry: combinatory logic
1937
Alan Turing: turing machines
1950
John McCarthy: LISP
1977
John Backus: FP

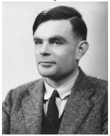
1917

1924 **Moses Schönfinkel:** combinatory logic

1930 **Haskell Curry:** combinatory logic

1936 **Alonzo Church:** λ-calculus

1937 **Alan Turing:** turing machines



1941 **Z3:** 1st programmable, fully automatic computing machine

1950 **John McCarthy:** LISP

1966 **Peter Landin:** Iswim

1977 **John Backus:** FP

1984 **Robin Milner:** LCF, Standard ML

2017

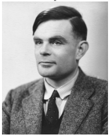
1917 — 2017

1924 **Moses Schönfinkel:** combinatory logic

1930 **Haskell Curry:** combinatory logic

1936 **Alonzo Church:** λ-calculus

1937 **Alan Turing:** turing machines



1941 **Z3:** 1st programmable, fully automatic computing machine

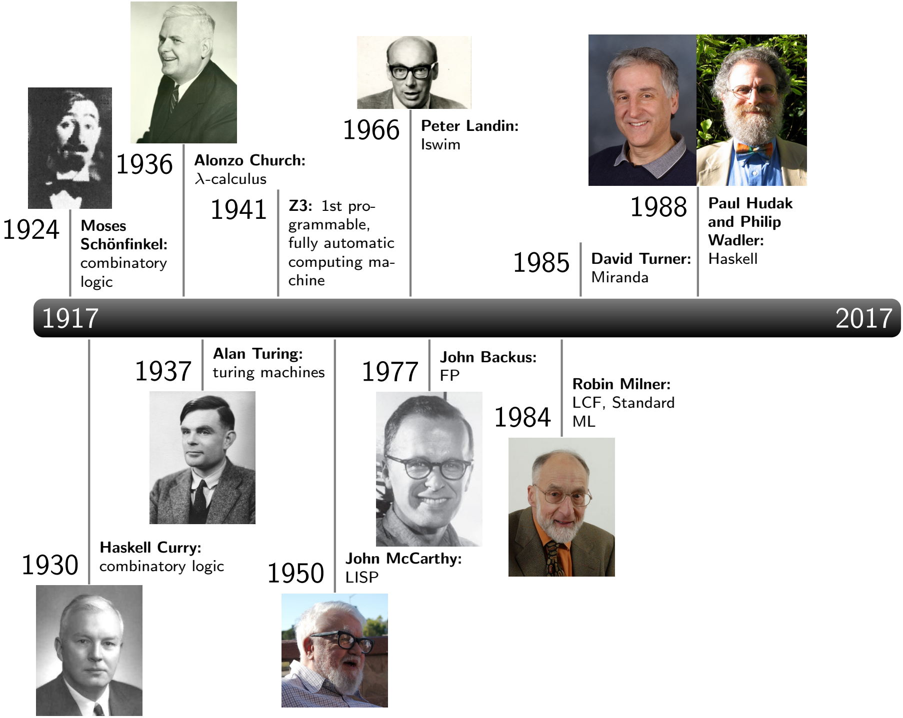
1950 **John McCarthy:** LISP

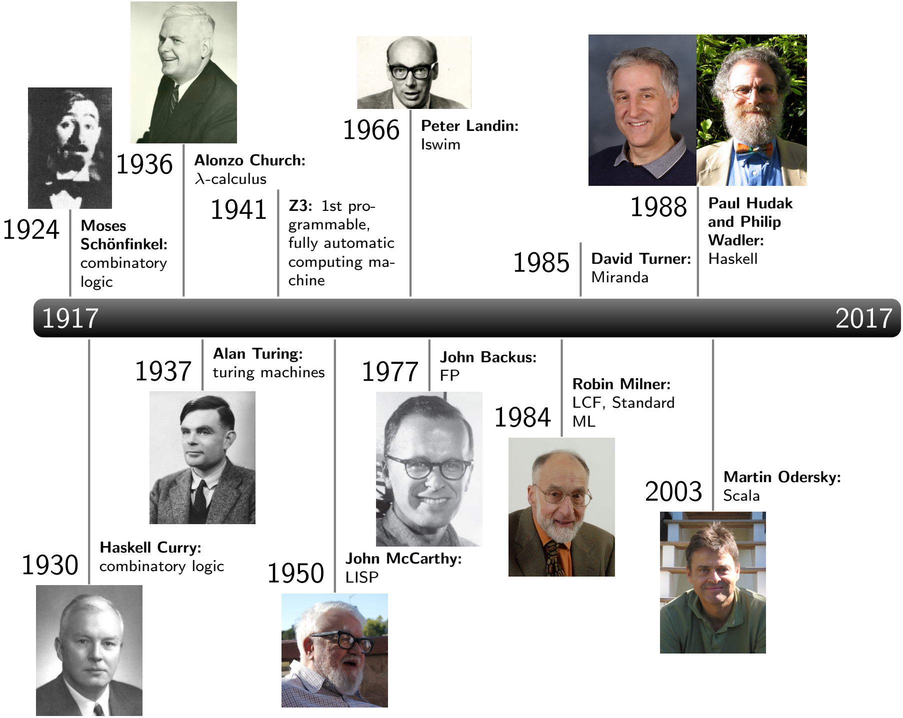
1966 **Peter Landin:** Iswim

1977 **John Backus:** FP

1984 **Robin Milner:** LCF, Standard ML

1985 **David Turner:** Miranda


1917
2017
1924
Moses Schönfinkel: combinatory logic
1930
Haskell Curry: combinatory logic
1936
Alonzo Church: λ-calculus
1937
Alan Turing: turing machines
1941
Z3: 1st programmable, fully automatic computing machine
1950
John McCarthy: LISP
1966
Peter Landin: Iswim
1977
John Backus: FP
1984
Robin Milner: LCF, Standard ML
1985
David Turner: Miranda
1988
Paul Hudak and Philip Wadler: Haskell


1917
2017
1924
Moses Schönfinkel: combinatory logic
1936
Alonzo Church: λ-calculus
1941
Z3: 1st programmable, fully automatic computing machine
1966
Peter Landin: Iswim
1985
David Turner: Miranda
1988
Paul Hudak and Philip Wadler: Haskell
1930
Haskell Curry: combinatory logic
1937
Alan Turing: turing machines
1950
John McCarthy: LISP
1977
John Backus: FP
1984
Robin Milner: LCF, Standard ML
2003
Martin Odersky: Scala

1917
2017
1924 Moses Schönfinkel: combinatory logic
1930 Haskell Curry: combinatory logic
1936 Alonzo Church: λ-calculus
1937 Alan Turing: turing machines
1941 Z3: 1st programmable, fully automatic computing machine
1950 John McCarthy: LISP
1966 Peter Landin: Iswim
1977 John Backus: FP
1984 Robin Milner: LCF, Standard ML
1985 David Turner: Miranda
1988 Paul Hudak and Philip Wadler: Haskell
2003 Martin Odersky: Scala
2010 Haskell2010

Notions

(Program) State

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

(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`, the location x has content 10 (the state changed)

(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`, the location x has content 10 (the state changed)

Side Effects

a function or expression has side effects if it modifies state

(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`, the location x has content 10 (the state 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 – $\sum_{i=0}^{n} i$

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

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
  ```

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

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

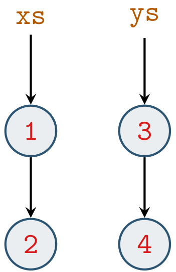
data that does not change after initial creation

Immutable Data

data that does not change after initial creation

Example – Linked Lists

- consider two linked lists `xs = [1,2]` and `ys = [3,4]`

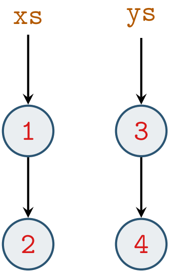


Immutable Data

data that does not change after initial creation

Example – Linked Lists

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



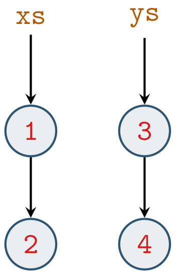
Immutable Data

data that does not change after initial creation

Example – Linked Lists

append elements of `ys` to `xs`

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

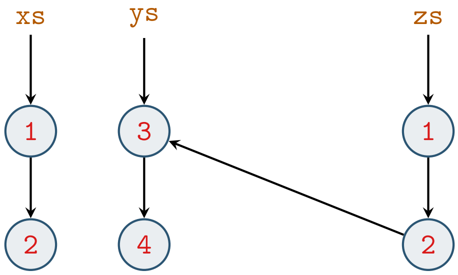


Immutable Data

data that does not change after initial creation

Example – Linked Lists

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

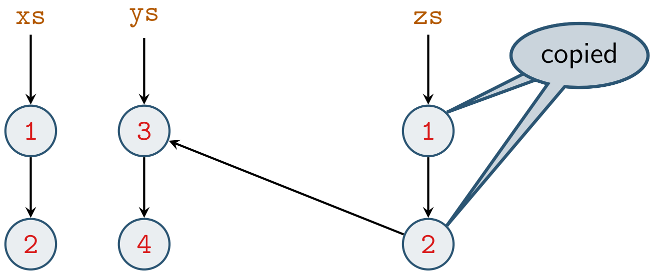


Immutable Data

data that does not change after initial creation

Example – Linked Lists

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



Recursion

a function is recursive if it is used in its own definition

Recursion

a function is recursive if it is used in its own definition

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”

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

```
mySum [] = 0                      (1)
mySum (x:xs) = x + mySum xs       (2)
```

Evaluating Functions by Hand (aka Equational Reasoning)

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

Example – mySum

pattern: empty list

given the two equations

```
mySum [] = 0                      (1)
mySum (x:xs) = x + mySum xs       (2)
```

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

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

```
      mySum [] = 0                    (1)
mySum (x:xs) = x + mySum xs           (2)
```

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

```
mySum [] = 0                     (1)
mySum (x:xs) = x + mySum xs      (2)
```

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

```
mySum [1,2,3]
```

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

```
mySum [] = 0                      (1)
mySum (x:xs) = x + mySum xs       (2)
```

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

`mySum [1,2,3]` $=$ `1 + mySum [2,3]` using (2)

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

```
mySum [] = 0                       (1)
mySum (x:xs) = x + mySum xs        (2)
```

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

```
mySum [1,2,3]  = 1 + mySum [2,3]          using (2)
               = 1 + (2 + mySum [3])      using (2)
```

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

```
mySum [] = 0                   (1)
mySum (x:xs) = x + mySum xs    (2)
```

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

```
mySum [1,2,3] = 1 + mySum [2,3]              using (2)
              = 1 + (2 + mySum [3])          using (2)
              = 1 + (2 + (3 + mySum []))     using (2)
```

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

```
mySum [] = 0                      (1)
mySum (x:xs) = x + mySum xs       (2)
```

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

```
mySum [1,2,3] = 1 + mySum [2,3]              using (2)
              = 1 + (2 + mySum [3])          using (2)
              = 1 + (2 + (3 + mySum []))     using (2)
              = 1 + (2 + (3 + 0))            using (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

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

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

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

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

A Taste of Haskell

Haskell

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

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

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]
```

First Steps

Haskell on the Web

- main entry point `www.haskell.org`
- most widely used Haskell compiler: GHC
- with interpreter GHCi

Haskell on the Web

- main entry point `www.haskell.org`
- most widely used Haskell compiler: GHC
- with interpreter GHCi

Starting the Interpreter (GHCi)

```
$ ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/
:? for help
...
Prelude>
```

The Standard Prelude

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

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`	as the name says: 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`

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`	as the name says: 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 denoted in green and others in blue; variables are denoted in 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 $*$

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
$f(x)$	`f x`
$f(x,y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(x,g(y))$	`f x (g y)`
$f(x)\,g(y)$	`f x * g y`
$f(a,b)+c\,d$	`f a b + c * d`

Haskell Scripts

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

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` ⟨*name*⟩	load script ⟨*name*⟩
`:reload`	reload current script
`:edit` ⟨*name*⟩	edit script ⟨*name*⟩
`:edit`	edit current script
`:type` ⟨*expr*⟩	show type of ⟨*expr*⟩
`:set` ⟨*prop*⟩	change various settings
`:show` ⟨*info*⟩	show various information
`:!` ⟨*cmd*⟩	execute ⟨*cmd*⟩ 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
```

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

$$
\begin{aligned}
\langle \mathit{lower} \rangle &\stackrel{\text{def}}{=} \mathtt{a} \mid \ldots \mid \mathtt{z} \\
\langle \mathit{upper} \rangle &\stackrel{\text{def}}{=} \mathtt{A} \mid \ldots \mid \mathtt{Z} \\
\langle \mathit{digit} \rangle &\stackrel{\text{def}}{=} \mathtt{0} \mid \ldots \mid \mathtt{9} \\
\langle \mathit{name} \rangle &\stackrel{\text{def}}{=} (\langle \mathit{lower} \rangle \mid \mathtt{\_})(\langle \mathit{lower} \rangle \mid \langle \mathit{upper} \rangle \mid \langle \mathit{digit} \rangle \mid \mathtt{'} \mid \mathtt{\_})^*
\end{aligned}
$$

Naming Requirements

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

choice

$$\begin{aligned}
\langle \textit{lower} \rangle &\stackrel{\text{def}}{=} \texttt{a} \mid \ldots \mid \texttt{z} \\
\langle \textit{upper} \rangle &\stackrel{\text{def}}{=} \texttt{A} \mid \ldots \mid \texttt{Z} \\
\langle \textit{digit} \rangle &\stackrel{\text{def}}{=} \texttt{0} \mid \ldots \mid \texttt{9} \\
\langle \textit{name} \rangle &\stackrel{\text{def}}{=} (\langle \textit{lower} \rangle \mid _)(\langle \textit{lower} \rangle \mid \langle \textit{upper} \rangle \mid \langle \textit{digit} \rangle \mid \texttt{'} \mid _)^*
\end{aligned}$$

Naming Requirements

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

choice

zero ore more times

$$
\begin{aligned}
\langle \mathit{lower} \rangle &\stackrel{\text{def}}{=} \texttt{a} \mid \ldots \mid \texttt{z} \\
\langle \mathit{upper} \rangle &\stackrel{\text{def}}{=} \texttt{A} \mid \ldots \mid \texttt{Z} \\
\langle \mathit{digit} \rangle &\stackrel{\text{def}}{=} \texttt{0} \mid \ldots \mid \texttt{9} \\
\langle \mathit{name} \rangle &\stackrel{\text{def}}{=} (\langle \mathit{lower} \rangle \mid \texttt{\_})(\langle \mathit{lower} \rangle \mid \langle \mathit{upper} \rangle \mid \langle \mathit{digit} \rangle \mid \texttt{'} \mid \texttt{\_})^*
\end{aligned}
$$

Naming Requirements

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

$$\langle lower \rangle \stackrel{\text{def}}{=} \mathtt{a} \mid \dots \mid \mathtt{z}$$

$$\langle upper \rangle \stackrel{\text{def}}{=} \mathtt{A} \mid \dots \mid \mathtt{Z}$$

$$\langle digit \rangle \stackrel{\text{def}}{=} \mathtt{0} \mid \dots \mid \mathtt{9}$$

$$\langle name \rangle \stackrel{\text{def}}{=} (\langle lower \rangle \mid \_)(\langle lower \rangle \mid \langle upper \rangle \mid \langle digit \rangle \mid \text{'} \mid \_)^*$$

choice

zero ore 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 _

Naming Requirements

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

choice

zero ore more times

$$
\begin{aligned}
\langle \mathit{lower} \rangle &\stackrel{\text{def}}{=} \mathtt{a} \mid \ldots \mid \mathtt{z} \\
\langle \mathit{upper} \rangle &\stackrel{\text{def}}{=} \mathtt{A} \mid \ldots \mid \mathtt{Z} \\
\langle \mathit{digit} \rangle &\stackrel{\text{def}}{=} \mathtt{0} \mid \ldots \mid \mathtt{9} \\
\langle \mathit{name} \rangle &\stackrel{\text{def}}{=} (\langle \mathit{lower} \rangle \mid \_)(\langle \mathit{lower} \rangle \mid \langle \mathit{upper} \rangle \mid \langle \mathit{digit} \rangle \mid \text{'} \mid \_)^*
\end{aligned}
$$

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, items 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 '`;`'

The Layout Rule

- items that start in same column are grouped together
- by increasing indentation, items 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

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

The Layout Rule

- items that start in same column are grouped together
- by increasing indentation, items 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

```
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 `-}`

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)
-}
```

Exercises (for October 20th)

1. Read
   http://haskell.org/haskellwiki/Functional_programming and
   http://haskell.org/haskellwiki/Haskell_in_5_steps.
2. Work through lessons 1 to 3 on http://tryhaskell.org/.
3. Explain and correct the 3 syntactic errors in the script:

   ```
   N = a 'div' length xs
     where
         a = 10
       xs  = [1,2,3,4,5]
   ```

4. Show how the library function `last` (selecting the last element of a non-empty list) could be defined in terms of the Prelude functions used in this lecture. Can you think of another possible definition?
5. Show two possible definitions of the library function `init` (removing the last element from a list) in terms of the functions introduced so far.
6. Use recursion to define a function `gcd`, computing the greatest common divisor of two given numbers.