Functional Programming

WS 2010/11

Christian Sternagel (VO)
Friedrich Neurauter (PS) Ulrich Kastlunger (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

October 6, 2010

Organization

Lecture

- LV-Nr. 703017
- VO 2
- `http://cl-informatik.uibk.ac.at/teaching/ws10/fp/`
- slides are also available online
- office hours: Tuesday 12:00 – 14:00 in 3N01
- online registration required before 23:59 on October 30
- grading: written exam (closed book)

Lecture

- LV-Nr. 703017
- VO 2
- `http://cl-informatik.uibk.ac.at/teaching/ws10/fp/`
- slides are also available online
- office hours: Tuesday 12:00 – 14:00 in 3N01
- online registration required before 23:59 on October 30
- grading: written exam (closed book)

Exercises

- LV-Nr. 703018
- PS 1
- three groups:

three groups:	group 1	Friday 8:15 – 9:00	HS 11
	group 2	Friday 9:15 – 10:00	HS 11
	group 3	Friday 9:15 – 10:00	SR 12

- office hours: Monday 12:00 – 13:30 in 3M03
  or by arrangement
- online registration required before 23:59 on October 8
- grading: 2 tests + weekly exercises
- exercises start on October 15

Exercises

- LV-Nr. 703018
- PS 1
- three groups:

group 1	Friday 8:15 – 9:00	HS 11
group 2	Friday 9:15 – 10:00	HS 11
group 3	Friday 9:15 – 10:00	SR 12

- office hours: Monday 12:00 – 13:30 in 3M03
or by arrangement
- online registration required before 23:59 on October 8
- grading: 2 tests + weekly exercises
- exercises start on October 15

Exercises

- LV-Nr. 703018
- PS 1
- three groups:

	group 1	Friday 8:15 – 9:00	HS 11
	group 2	Friday 9:15 – 10:00	HS 11
	group 3	Friday 9:15 – 10:00	SR 12

- office hours: Monday 12:00 – 13:30 in 3M03
or by arrangement
- online registration required before 23:59 on October 8
- grading: 2 tests + weekly exercises
- exercises start on October 15

Exercises

- LV-Nr. 703018
- PS 1
- three groups:

three groups:	group 1	Friday 8:15 – 9:00	HS 11
	group 2	Friday 9:15 – 10:00	HS 11
	group 3	Friday 9:15 – 10:00	SR 12

- office hours: Monday 12:00 – 13:30 in 3M03
  or by arrangement
- online registration required before 23:59 on October 8
- grading: 2 tests + weekly exercises
- exercises start on October 15

Schedule

week 1	October	6	week 8	November	24
week 2	October	13	week 9	December	1
week 3	October	20	week 10	December	15
week 4	October	27	week 11	January	12
week 5	November	3	week 12	January	19
week 6	November	10	week 13	January	26
week 7	November	17	week 14	February	2

Schedule

week 1	October	6	November 26: 1st test		
week 2	October	13	week 9	December	1
week 3	October	20	week 10	December	15
week 4	October	27	week 11	January	12
week 5	November	3	January 21: 2nd test		
week 6	November	10	week 13	January	26
week 7	November	17	February 2: 1st exam		

Practical Topics

- lists
- strings
- trees
- sets
- combinator parsing
- lazy lists
- monads
- ...

Theoretical Topics

- λ-calculus
- evaluation strategies
- induction
- reasoning about programs
- efficiency
- type checking/inference
- ...

Today's Topics

- Historical Overview
- Notions
- A Taste of Haskell
- First Steps

History

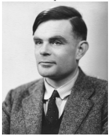
1936 **Alonzo Church:**
λ-calculus

1924 2010

1936 **Alonzo Church:**
λ-calculus

1924 2010

1937 **Alan Turing:**
turing machines

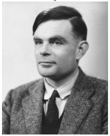


1936 **Alonzo Church:** λ-calculus

1924 **Moses Schönfinkel:** combinatory logic

1924 2010

1937 **Alan Turing:** turing machines

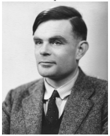


1936 **Alonzo Church:** λ-calculus

1924 **Moses Schönfinkel:** combinatory logic

1924 — 2010

1937 **Alan Turing:** turing machines



1930 **Haskell Curry:** combinatory logic

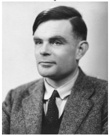
1936 **Alonzo Church:** λ-calculus

1924 **Moses Schönfinkel:** combinatory logic

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

1924 — 2010

1937 **Alan Turing:** turing machines



1930 **Haskell Curry:** combinatory logic

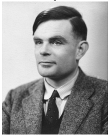
1936 **Alonzo Church:** λ-calculus

1924 **Moses Schönfinkel:** combinatory logic

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

1924 — 2010

1937 **Alan Turing:** turing machines



1930 **Haskell Curry:** combinatory logic

1950 **John McCarthy:** LISP

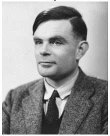
1924 — 2010

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

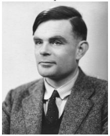
1924 — 2010

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

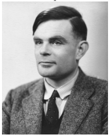
1924 — 2010

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

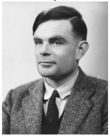
1924 — 2010

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

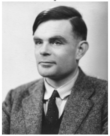
1924 — 2010

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

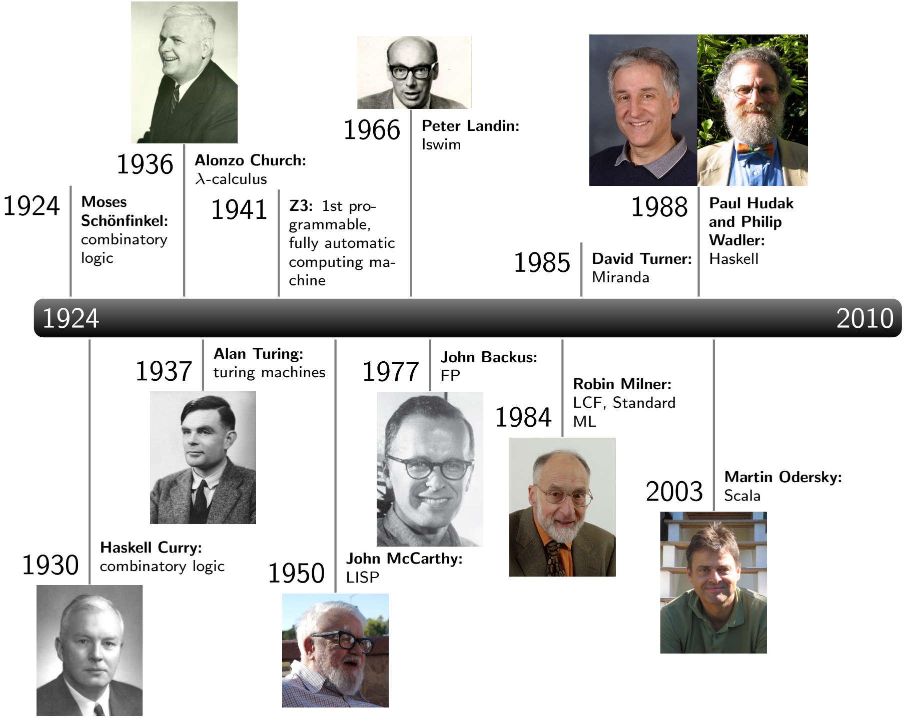
1966 **Peter Landin:** Iswim

1977 **John Backus:** FP

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

1985 **David Turner:** Miranda

1988 **Paul Hudak and Philip Wadler:** Haskell


1924
2010
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

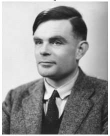
1924 — 2010

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

State

a state is the “content” of a certain “region in memory”

State

a state is the "content" of a certain "region in memory"

Example - Assignment

after `x := 10`, the region called "x" has state 10

State

a state is the "content" of a certain "region in memory"

Example - Assignment

after `x := 10`, the region called "x" has state 10

Side Effects

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

State

a state is the “content” of a certain “region in memory”

Example - Assignment

after `x := 10`, the region called “x” has state 10

Side Effects

a function or expression has side effects if it modifies some 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 [1..n]
```

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

Example - $\sum_{i=0}^{n} i$

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

```
sum [1..n]
```

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

Self-Made Definitions

Example - $\sum_{i=0}^{n} i$

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

```
sum [1..n]
```

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

Self-Made Definitions

- `[m..n]` computes the 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 [1..n]
```

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

Self-Made Definitions

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

```
range m n = if m > n then []
                     else m : range (m+1) n
```

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

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

Pure Functions

a function is pure if it

- evaluates the same result, given the same arguments; and
- does not have side effects

Pure Functions

a function is pure if it

- evaluates the same result, given the same arguments; and
- does not have side effects

Example - Random Numbers

The function rand (producing random numbers) is not pure

$\text{rand}() = 0$
$\text{rand}() = 10$
$\text{rand}() = 42$

Pure Functions

a function is pure if it

- evaluates the same result, given the same arguments; and
- does not have side effects

Example - Random Numbers

The function rand (providing random numbers) is not pure

rand() $= 0$
rand() $= 10$
rand() $= 42$

sometimes called referential transparent

Immutable Data

data whose state does not change after the initial creation

Immutable Data

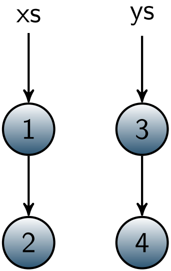
data whose state does not change after the initial creation

Example - Linked Lists

consider 2 linked lists

xs = [1,2]

ys = [3,4]



Immutable Data

data whose state does not change after the initial creation

Example - Linked Lists

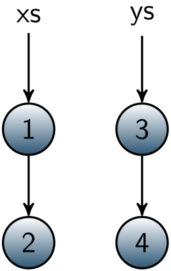
consider 2 linked lists

xs = [1,2]

ys = [3,4]

after concatenation

zs = xs ++ ys



Immutable Data

data whose state does not change after the initial creation

Example - Linked Lists

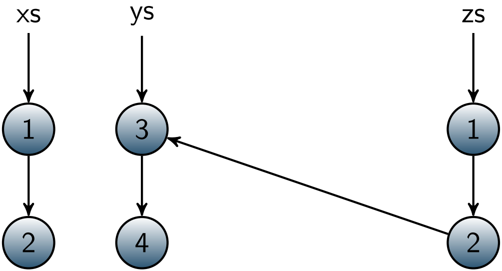
consider 2 linked lists

$xs = [1,2]$

$ys = [3,4]$

after concatenation

$zs = xs \mathbin{++} ys$



Immutable Data

data whose state does not change after the initial creation

Example - Linked Lists

append elements of *ys* to *xs*

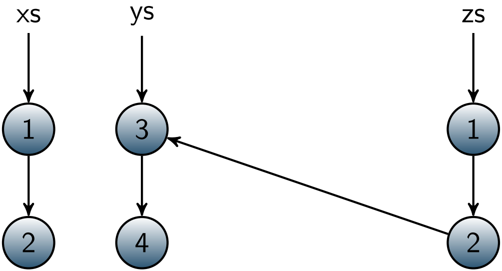
consider 2 linked lists

xs = `[1,2]`

ys = `[3,4]`

after concatenation

zs = xs `++` ys



Immutable Data

data whose state does not change after the initial creation

Example - Linked Lists

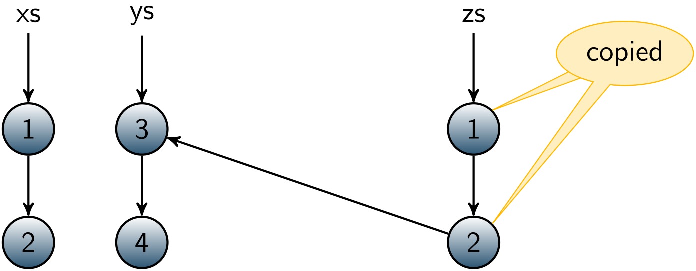
consider 2 linked lists

xs = [1,2]

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 following two equations for `mySum`

$$\begin{aligned} \texttt{mySum}\ [\,] &= 0 && (1) \\ \texttt{mySum}\ (x{:}xs) &= x + \texttt{mySum}\ xs && (2) \end{aligned}$$

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

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

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 following two equations for `mySum`

empty list

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

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

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

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

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 following two equations for `mySum`

list with "head" x and "tail" xs

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

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

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

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

A Taste of Haskell

Haskell

- is a pure (hence, also no side effects) language
- functions are defined by equations and pattern matching

Haskell

- is a pure (hence, also no side effects) language
- functions are defined by equations and pattern matching

Example - qsort

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

```haskell
qsort []     = []
qsort (x:xs) = qsort leq ++ [x] ++ qsort gt
               where
                 leq = [a | a <- xs, a <= x]
                 gt  = [b | b <- xs, b > x]
```

First Steps

The Haskell Platform

- available from http://hackage.haskell.org/platform/
- ships with the most widely used Haskell compiler: GHC
- and its interpreter GHCi

The Haskell Platform

- available from http://hackage.haskell.org/platform/
- ships with the most widely used Haskell compiler: GHC
- and its interpreter GHCi

Starting the Interpreter (GHCi)

```
$ ghci
GHCi, version 6.12.1: http://www.haskell.org/ghc/
:? for help
...
Prelude>
```

The Standard Prelude

- on startup GHCi loads the file `Prelude.hs`, importing many standard functions

The Standard Prelude

- on startup GHCi loads the file `Prelude.hs`, importing many standard functions

Examples

- arithmetic: `+`, `-`, `*`, `/`, `^`, `mod`, `div`
- lists

`drop`	drop the first n elements from a list
`head`	extract the first element from a list
`length`	number of elements in a list
`reverse`	reverse the order of elements in a list
`sum`	sum up the elements of a list
`tail`	obtain the list without its first element
`take`	take the first n elements from a list

The Standard Prelude

- on startup GHCi loads the file `Prelude.hs`, importing many standard functions

Examples

- arithmetic: `+`, `-`, `*`, `/`, `^`, `mod`, `div`
- lists

`drop`	drop the first *n* elements from a list
`head`	extract the first element from a list
`length`	number of elements in a list
`reverse`	reverse the order of elements in a list
`sum`	sum up the elements of a list
`tail`	obtain the list without its first element
`take`	take the first *n* elements from a list

- note: in code examples Prelude functions are denoted like this and others like this

Function Application

- in mathematics: function application is denoted by enclosing the arguments in parenthesis, 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 the arguments in parenthesis, 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 a function in `...` turns it infix

Naming Requirements

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

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

Naming Requirements

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

choice

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

Naming Requirements

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

choice

zero ore more times

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

Naming Requirements

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

choice

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

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 the following syntax

choice

zero ore more times

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

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 the same column are grouped together
- by increasing indentation, items may span multiple lines
- groups end at EOF or when indentation decreases
- ignore layout: enclosing groups in braces (`{`,`}`) and separating items by semicolons (`;`)
- the content of a script is a group, nested groups are started by one of **`where`**, **`let`**, **`do`**, and **`of`**

The Layout Rule

- items that start in the same column are grouped together
- by increasing indentation, items may span multiple lines
- groups end at EOF or when indentation decreases
- ignore layout: enclosing groups in braces (`{`,`}`) and separating items by semicolons (`;`)
- the content of a script is a group, nested groups are started by one of **`where`**, **`let`**, **`do`**, and **`of`**

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 the same column are grouped together
- by increasing indentation, items may span multiple lines
- groups end at EOF or when indentation decreases
- ignore layout: enclosing groups in braces (`{`,`}`) and separating items by semicolons (`;`)
- the content of a script is a group, nested groups are started by one of **`where`**, **`let`**, **`do`**, and **`of`**

Examples

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

without using 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 integer:
factorial n = product [1..n]

-- Average of a list of integers:
average ns  = sum ns `div` length ns

{- currently not used
double x = x + x
quadruple x = double (double x)
-}
```

Exercises (for October 15th)

1. read
   http://haskell.org/haskellwiki/Functional_programming
   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 integers.