MOD08: Functional Programming

Introduction to Functional Programming

Marco Gerards

April 2019

Organisation

- Part of Module 8: Programming Paradigms
- Homologation course: “Functional programming”
  - Self-study: compiler construction concepts
  - Permission from study advisor is required
  - Additional project (no supervision)
  - **Register your group today before 18:00**

Organisation

- Part of Module 8: Programming Paradigms
- Homologation course: “Functional programming”
  - Self-study: compiler construction concepts
  - Permission from study advisor is required
  - Additional project (no supervision)
  - **Register your group today before 18:00**
- Functional Programming
  - Language: Haskell
  - Compiler: GHC (Glasgow Haskell Compiler)
    - Interpreter: GHCi (command: `ghci`)
    - Compiler: GHC (command: `ghc`)
- Lectures: provide theoretical background
- Practical sessions: essential to grasp the material

Assignments and grading

- Practical sessions
  - Work in pairs; discussions allowed (but: no copying)
  - Sign-off during the session
  - Sign-off as soon as possible, avoid queues

Assignments and grading

- Practical sessions
  - Work in pairs; discussions allowed (but: no copying)
  - Sign-off during the session
  - Sign-off as soon as possible, avoid queues
- Final assignment (functional programming)
  - Same pairs; after finishing practical sessions
  - All students

Assignments and grading

- Practical sessions
  - Work in pairs; discussions allowed (but: no copying)
  - Sign-off during the session
  - Sign-off as soon as possible, avoid queues
- Final assignment (functional programming)
  - Same pairs; after finishing practical sessions
  - All students
- Final project of module 8
  - Define a programming language
  - Write a compiler
    - Target: Sprockel processor (Processor defined in Haskell)
    - Concurrency: multiple Sprockels
    - Use Haskell (Parsec) or Java (Antlr)

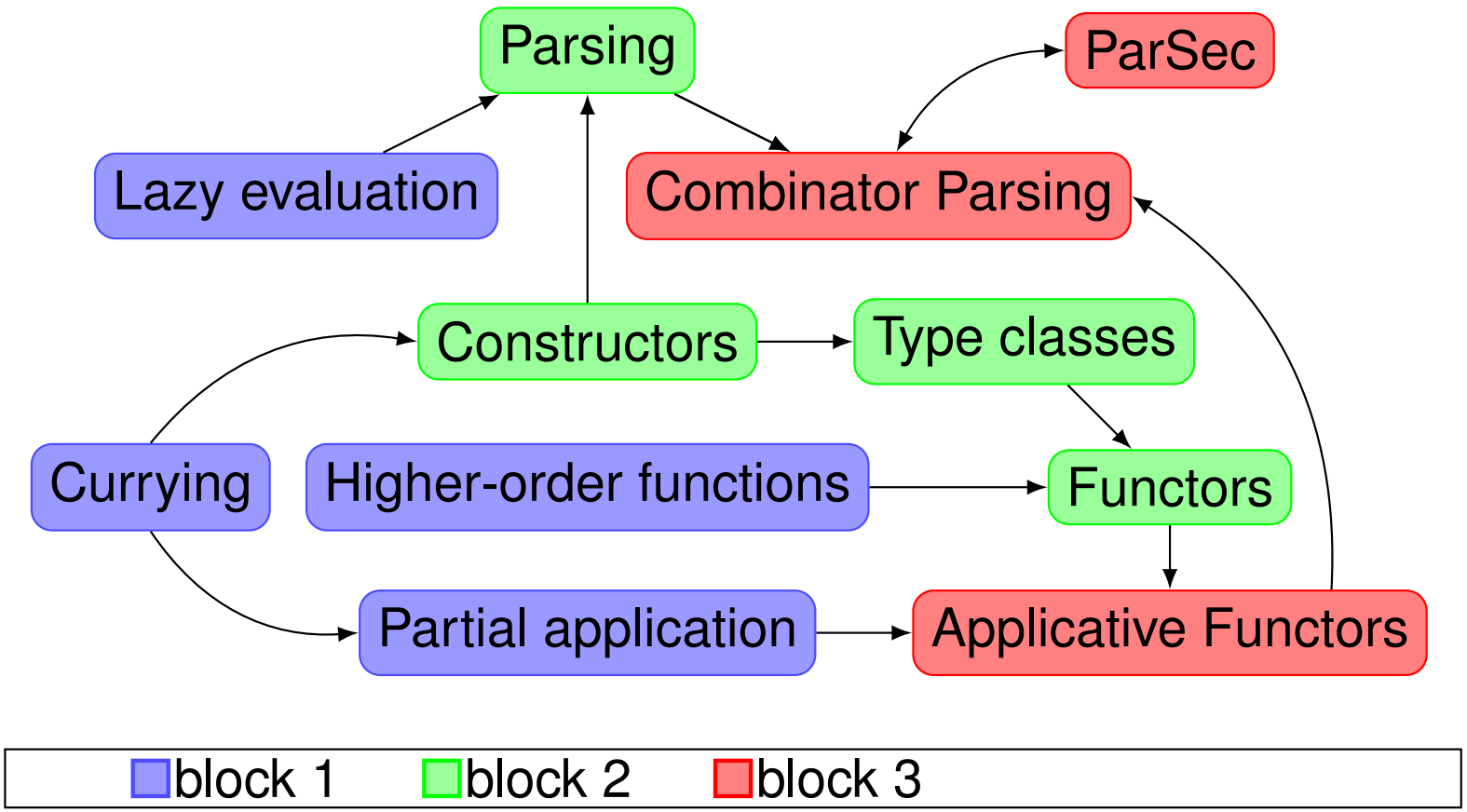
Assignments and grading

- Practical sessions
  - Work in pairs; discussions allowed (but: no copying)
  - Sign-off during the session
  - Sign-off as soon as possible, avoid queues
- Final assignment (functional programming)
  - Same pairs; after finishing practical sessions
  - All students
- Final project of module 8
  - Define a programming language
  - Write a compiler
    - Target: Sprockel processor (Processor defined in Haskell)
    - Concurrency: multiple Sprockels
    - Use Haskell (Parsec) or Java (Antlr)
- Written test

Organisation of the course

Block	Topic
1	- Introduction to Functional Programming
	- Higher order functions
2	- Types and type classes
	- Parsing (application)
3	- Advanced type classes
	- Parser combinators and ParSec (application)
4	- Code generation (application)
	- TBD
5-7	- Project

Connection of some of the topics between blocks



This lecture: Introduction to Functional Programming

- Functional programming (concept)
- Evaluation and execution mechanisms
  - Lazy evaluation
- Basic concepts of functional programming
  - Recursion
  - List comprehension
- (higher order functions)

Materials and resources

- Haskell software: Glasgow Haskell Compiler (GHC)
  - Minimal installer from: `haskell.org/downloads`
  - Version 8.0.2a
- Books (optional); order by lecturers preference
  - Learn You a Haskell for Great Good!, Miran Lipovaca 
  - Programming in Haskell (2nd ed.), Graham Hutton
- Open book test:
  - Any English book about Haskell (also those not listed above)
    - You may print the book yourself
  - *and* print of lecture slides
  - *and* print of module guide
  - No (handwritten) notes
    - Also in the material/book
- Search for Haskell functions: http://haskell.org/hoogle

Functional programming: programming with functions

Functional vs. Imperative

Implementations of $f(x_1, \ldots, x_n) = \sqrt{\sum_i x_i^2}$

Functional vs. Imperative

Implementations of $f(x_1, \ldots, x_n) = \sqrt{\sum_i x_i^2}$

Functional

```
len xs = sqrt (sum (map (^2) xs))
```

Imperative (C)

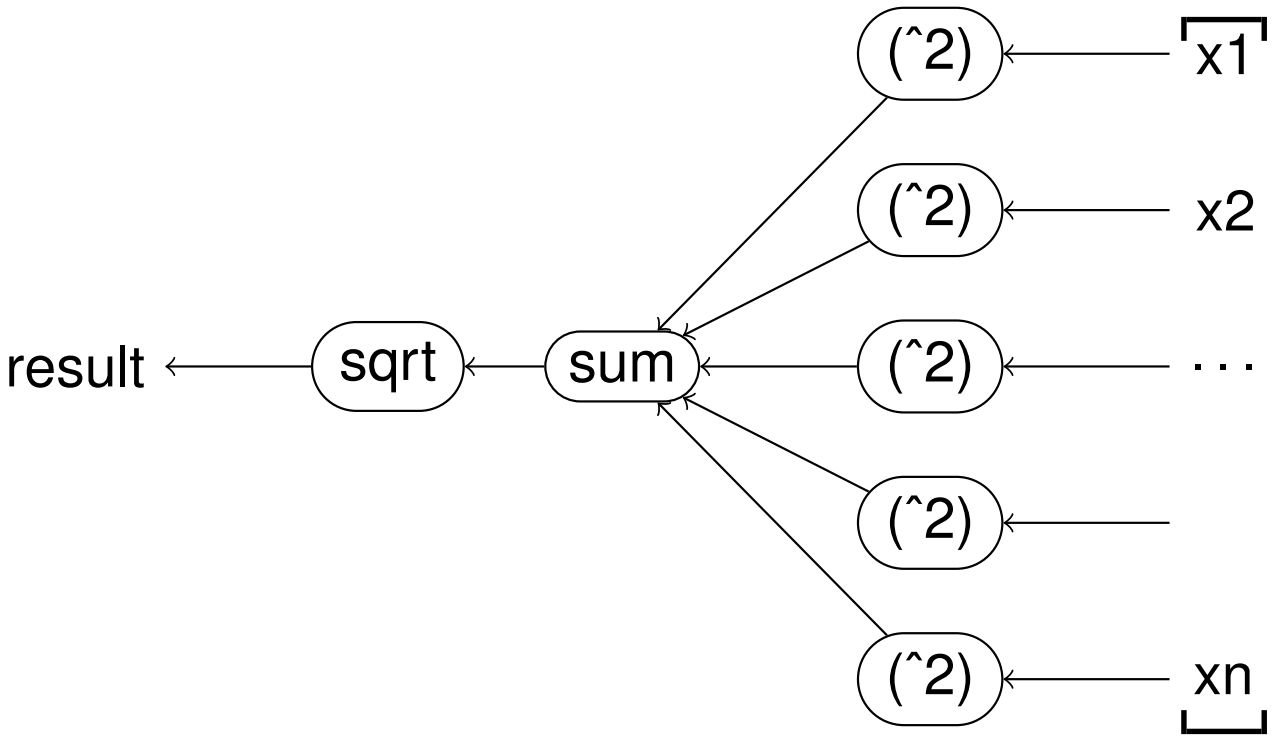
```
double
len (float x[], int l)
{
  int i;
  double r = 0.0;
  for (i = 0; i < l; i++)
    r += x[i]^2;
  return sqrt(r);
}
```

Visualization of a function

```
len  xs = sqrt (sum (map (^2) xs))

len' xs = sqrt $ sum $ (^2) <$> xs
```

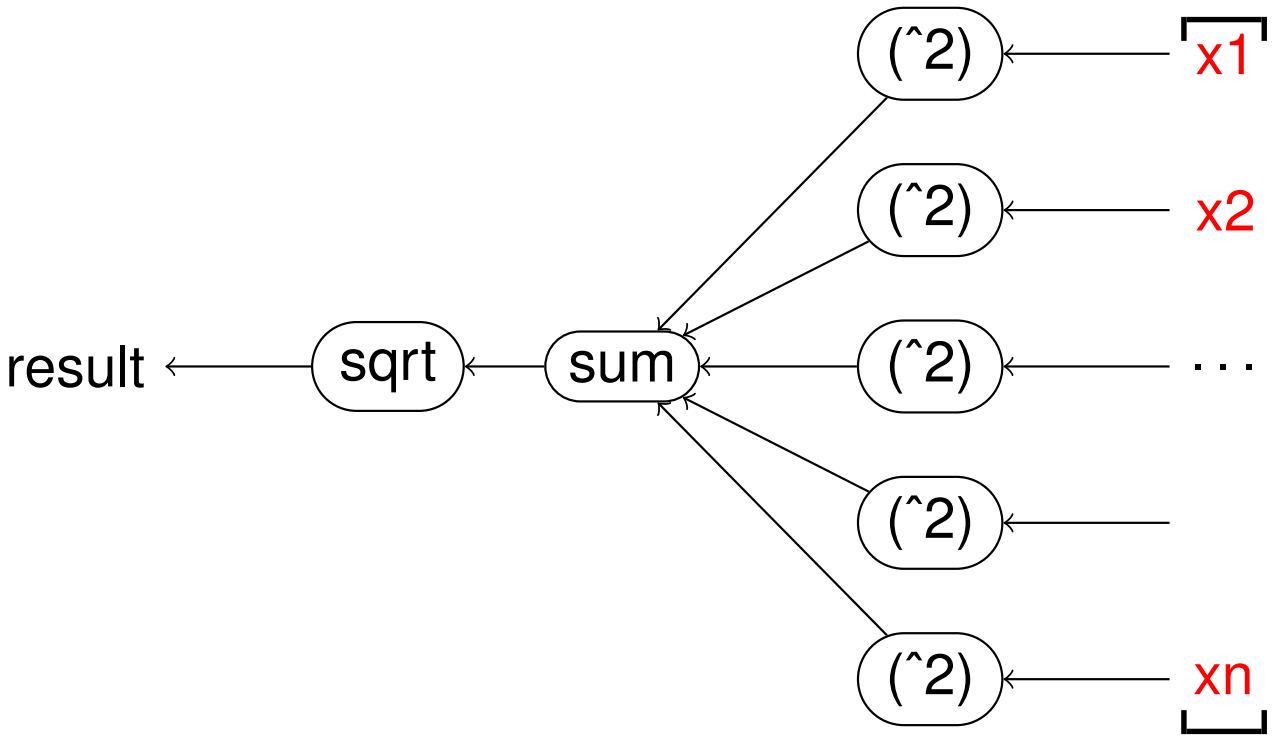
Visualization of a function

```
len  xs = sqrt (sum (map (^2) xs))

len' xs = sqrt $ sum $ (^2) <$> xs
```



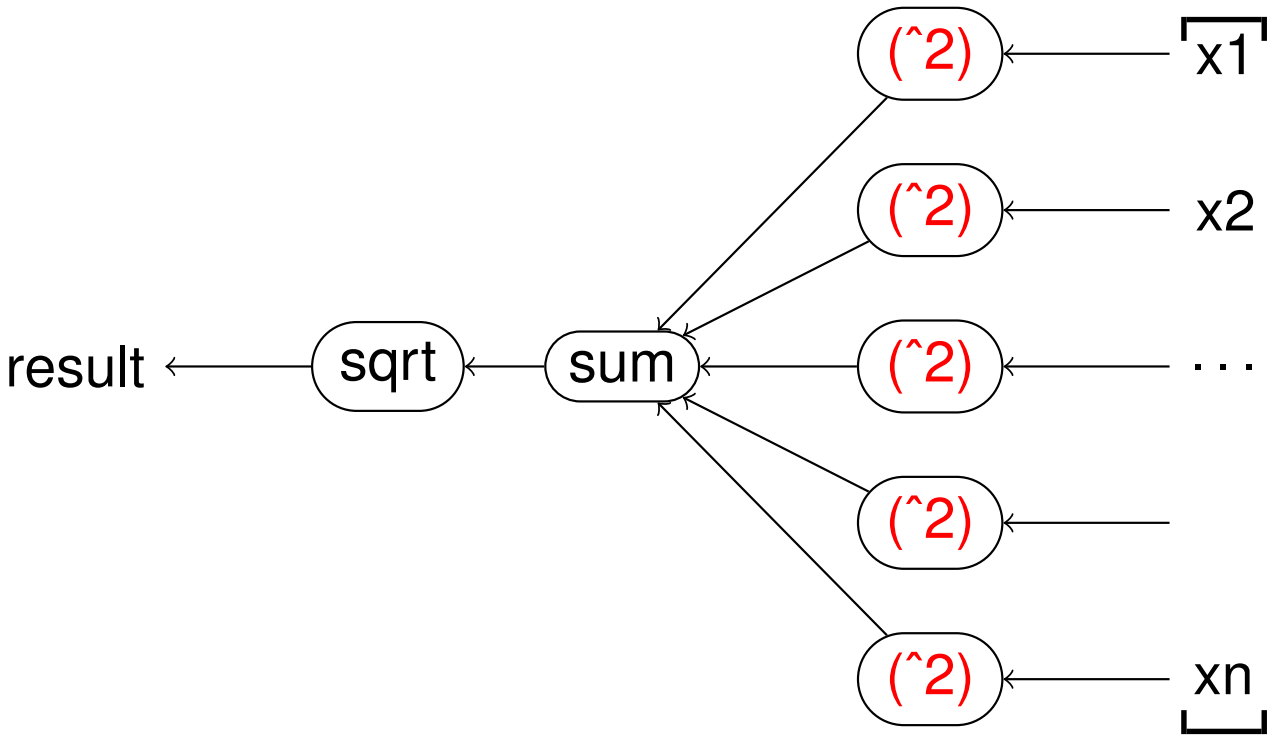
Visualization of a function

```
len  xs = sqrt (sum (map (^2) xs))

len' xs = sqrt $ sum $ (^2) <$> xs
```



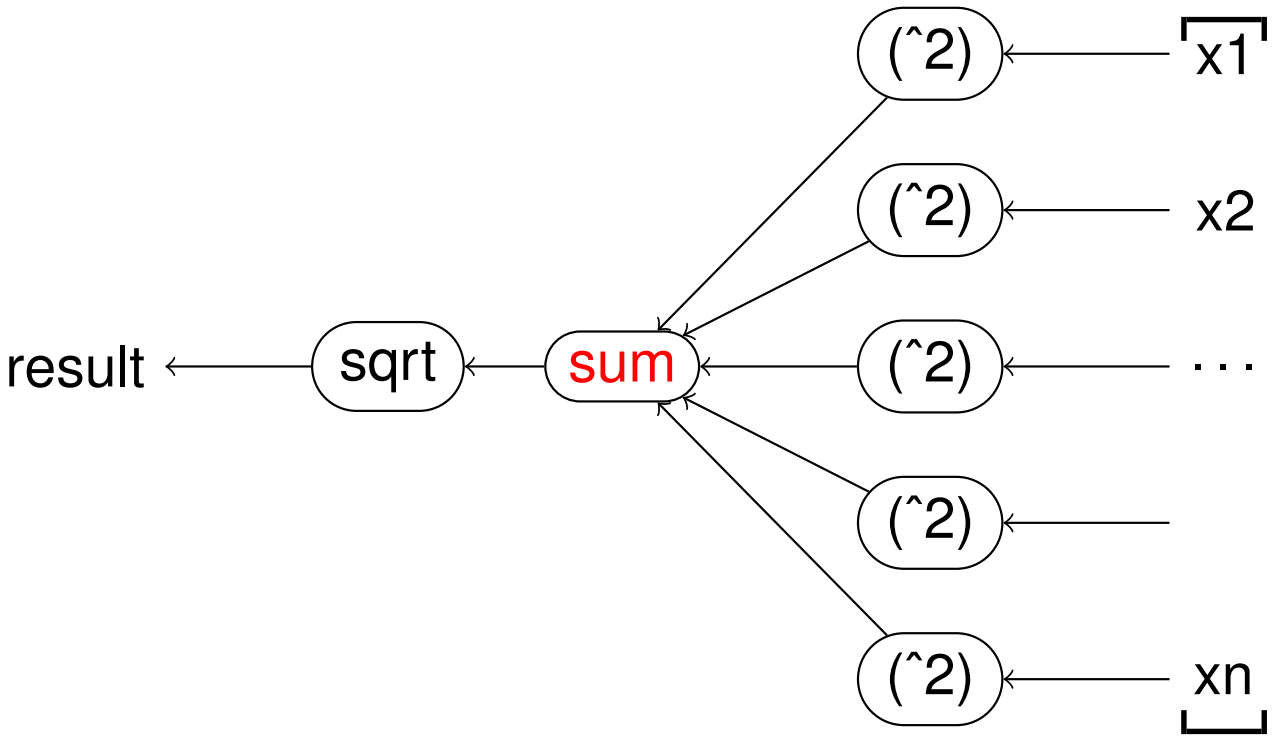
Visualization of a function

```
len  xs = sqrt (sum (map (^2) xs))

len' xs = sqrt $ sum $ (^2) <$> xs
```



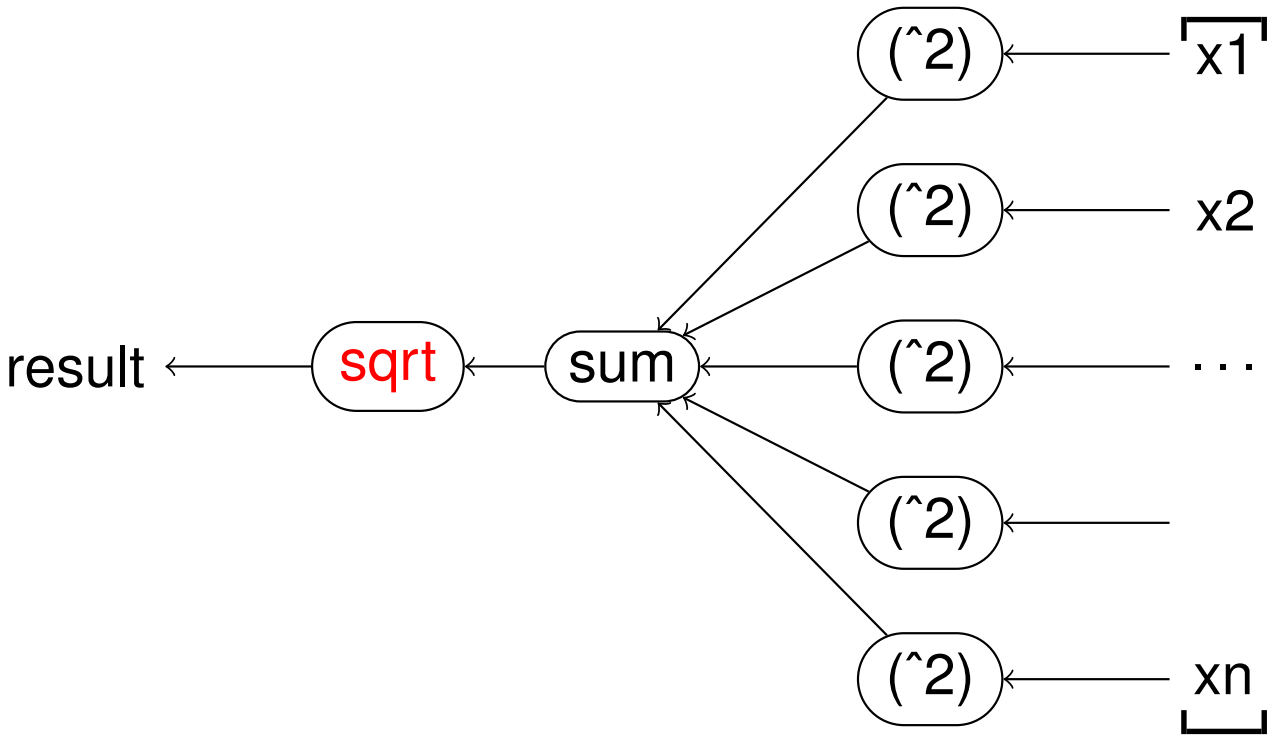
Visualization of a function

```
len  xs = sqrt (sum (map (^2) xs))

len' xs = sqrt $ sum $ (^2) <$> xs
```



Visualization of a function

```
len  xs = sqrt (sum (map (^2) xs))

len' xs = sqrt $ sum $ (^2) <$> xs
```



Why Functional Programming

- Reasoning about what happens
  - Proofs, critical applications (e.g., banking)
  - When the compiler accepts code, it often works

Why Functional Programming

- Reasoning about what happens
  - Proofs, critical applications (e.g., banking)
  - When the compiler accepts code, it often works
- Structural descriptions
  - Examples: testing, parsing, hardware

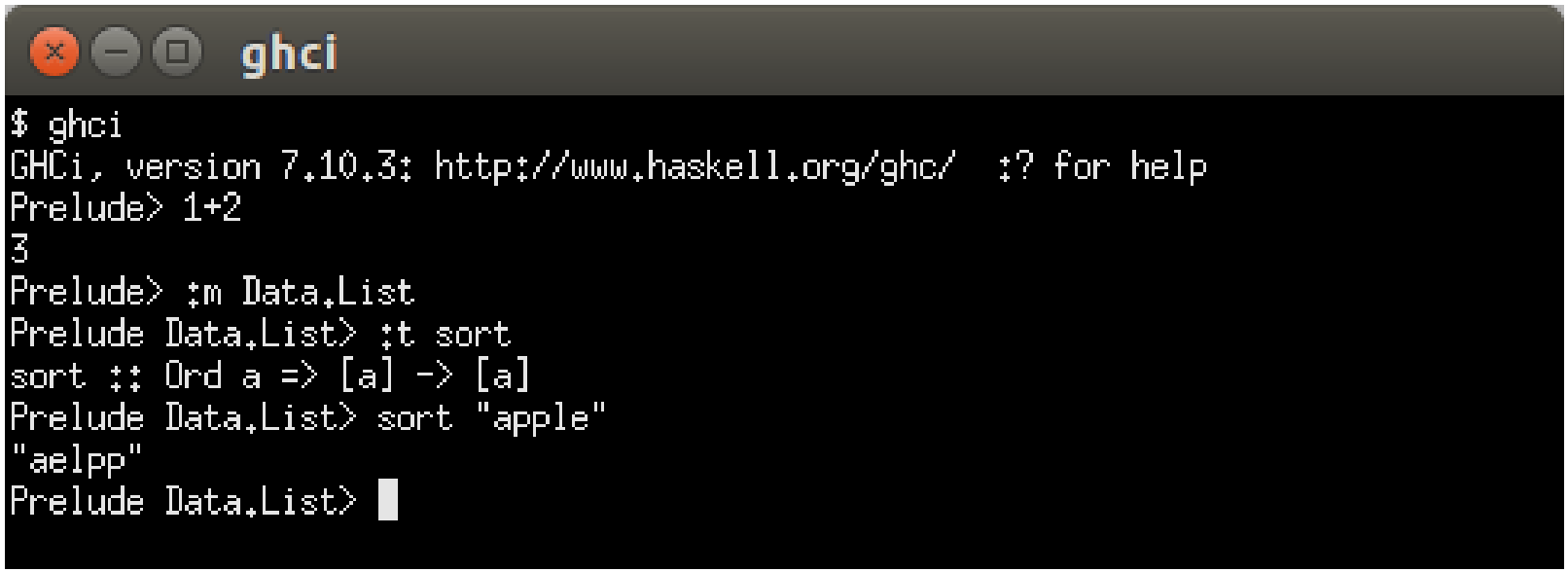
Why Functional Programming

- Reasoning about what happens
  - Proofs, critical applications (e.g., banking)
  - When the compiler accepts code, it often works
- Structural descriptions
  - Examples: testing, parsing, hardware
- Applications
  - Digital hardware design: CλaSH
  - Inherent parallelism: multicore, GPUs, etc.
  - Distributed concurrent applications (e.g., Erlang)
  - Quantum Computing

Why Functional Programming

- Reasoning about what happens
  - Proofs, critical applications (e.g., banking)
  - When the compiler accepts code, it often works
- Structural descriptions
  - Examples: testing, parsing, hardware
- Applications
  - Digital hardware design: CλaSH
  - Inherent parallelism: multicore, GPUs, etc.
  - Distributed concurrent applications (e.g., Erlang)
  - Quantum Computing
- Concepts adapted in imperative languages
  - List comprehension (Python, LINQ in C#, ...)
  - Lambda functions
  - ...

GHCi: GHC interactive environment (interpreter)




```
Prelude> 1+2
Prelude Data.List> :t sort
sort ::  Ord a => [a] -> [a]
Prelude Data.List> sort "apple"
"aelpp"
Prelude Data.List>
```

Expressions in GHCi

Expressions in GHCi

- 3+4

Expressions in GHCi

- 3+4 $\Rightarrow$ 7

Expressions in GHCi

- `3+4` $\Rightarrow$ `7`
- `[1..6]`

Expressions in GHCi

- `3+4` ⇒ `7`
- `[1..6]` ⇒ `[1,2,3,4,5,6]`

Expressions in GHCi

- `3+4` ⇒ `7`
- `[1..6]` ⇒ `[1,2,3,4,5,6]`
- `[1..3] ++ [10..12]`

Expressions in GHCi

- `3+4` ⇒ `7`
- `[1..6]` ⇒ `[1,2,3,4,5,6]`
- `[1..3] ++ [10..12]` ⇒ `[1,2,3,10,11,12]`

Expressions in GHCi

- `3+4` ⇒ `7`
- `[1..6]` ⇒ `[1,2,3,4,5,6]`
- `[1..3] ++ [10..12]` ⇒ `[1,2,3,10,11,12]`
- `['a'..'z']`

Expressions in GHCi

- `3+4 ⇒ 7`
- `[1..6] ⇒ [1,2,3,4,5,6]`
- `[1..3] ++ [10..12] ⇒ [1,2,3,10,11,12]`
- `['a'..'z'] ⇒ "abcdefghijklmnopqrstuvwxyz"`

Expressions in GHCi

- `3+4 ⇒ 7`
- `[1..6] ⇒ [1,2,3,4,5,6]`
- `[1..3] ++ [10..12] ⇒ [1,2,3,10,11,12]`
- `['a'..'z'] ⇒ "abcdefghijklmnopqrstuvwxyz"`
- `take 3 [1..6]`

Expressions in GHCi

- `3+4 ⇒ 7`
- `[1..6] ⇒ [1,2,3,4,5,6]`
- `[1..3] ++ [10..12] ⇒ [1,2,3,10,11,12]`
- `['a'..'z'] ⇒ "abcdefghijklmnopqrstuvwxyz"`
- `take 3 [1..6] ⇒ [1,2,3]`

Expressions in GHCi

- `3+4 ⇒ 7`
- `[1..6] ⇒ [1,2,3,4,5,6]`
- `[1..3] ++ [10..12] ⇒ [1,2,3,10,11,12]`
- `['a'..'z'] ⇒ "abcdefghijklmnopqrstuvwxyz"`
- `take 3 [1..6] ⇒ [1,2,3]`
- `drop 3 [1..6]`

Expressions in GHCi

- `3+4` ⇒ `7`
- `[1..6]` ⇒ `[1,2,3,4,5,6]`
- `[1..3] ++ [10..12]` ⇒ `[1,2,3,10,11,12]`
- `['a'..'z']` ⇒ `"abcdefghijklmnopqrstuvwxyz"`
- `take 3 [1..6]` ⇒ `[1,2,3]`
- `drop 3 [1..6]` ⇒ `[4,5,6]`

Expressions in GHCi

- `3+4` ⇒ `7`
- `[1..6]` ⇒ `[1,2,3,4,5,6]`
- `[1..3] ++ [10..12]` ⇒ `[1,2,3,10,11,12]`
- `['a'..'z']` ⇒ `"abcdefghijklmnopqrstuvwxyz"`
- `take 3 [1..6]` ⇒ `[1,2,3]`
- `drop 3 [1..6]` ⇒ `[4,5,6]`
- `[1..]`

Expressions in GHCi

- `3+4 ⇒ 7`
- `[1..6] ⇒ [1,2,3,4,5,6]`
- `[1..3] ++ [10..12] ⇒ [1,2,3,10,11,12]`
- `['a'..'z'] ⇒ "abcdefghijklmnopqrstuvwxyz"`
- `take 3 [1..6] ⇒ [1,2,3]`
- `drop 3 [1..6] ⇒ [4,5,6]`
- `[1..]`
- `take 3 [1..]`

Expressions in GHCi

- `3+4 ⇒ 7`
- `[1..6] ⇒ [1,2,3,4,5,6]`
- `[1..3] ++ [10..12] ⇒ [1,2,3,10,11,12]`
- `['a'..'z'] ⇒ "abcdefghijklmnopqrstuvwxyz"`
- `take 3 [1..6] ⇒ [1,2,3]`
- `drop 3 [1..6] ⇒ [4,5,6]`
- `[1..]`
- `take 3 [1..] ⇒ [1,2,3]`

Expressions in GHCi

- `3+4 ⇒ 7`
- `[1..6] ⇒ [1,2,3,4,5,6]`
- `[1..3] ++ [10..12] ⇒ [1,2,3,10,11,12]`
- `['a'..'z'] ⇒ "abcdefghijklmnopqrstuvwxyz"`
- `take 3 [1..6] ⇒ [1,2,3]`
- `drop 3 [1..6] ⇒ [4,5,6]`
- `[1..]`
- `take 3 [1..] ⇒ [1,2,3]`
- `reverse [1..6]`

Expressions in GHCi

- `3+4 ⇒ 7`
- `[1..6] ⇒ [1,2,3,4,5,6]`
- `[1..3] ++ [10..12] ⇒ [1,2,3,10,11,12]`
- `['a'..'z'] ⇒ "abcdefghijklmnopqrstuvwxyz"`
- `take 3 [1..6] ⇒ [1,2,3]`
- `drop 3 [1..6] ⇒ [4,5,6]`
- `[1..]`
- `take 3 [1..] ⇒ [1,2,3]`
- `reverse [1..6] ⇒ [6,5,4,3,2,1]`

Expressions in GHCi

- `3+4 ⇒ 7`
- `[1..6] ⇒ [1,2,3,4,5,6]`
- `[1..3] ++ [10..12] ⇒ [1,2,3,10,11,12]`
- `['a'..'z'] ⇒ "abcdefghijklmnopqrstuvwxyz"`
- `take 3 [1..6] ⇒ [1,2,3]`
- `drop 3 [1..6] ⇒ [4,5,6]`
- `[1..]`
- `take 3 [1..] ⇒ [1,2,3]`
- `reverse [1..6] ⇒ [6,5,4,3,2,1]`
- `splitAt 3 [1..6]`

Expressions in GHCi

- `3+4 ⇒ 7`
- `[1..6] ⇒ [1,2,3,4,5,6]`
- `[1..3] ++ [10..12] ⇒ [1,2,3,10,11,12]`
- `['a'..'z'] ⇒ "abcdefghijklmnopqrstuvwxyz"`
- `take 3 [1..6] ⇒ [1,2,3]`
- `drop 3 [1..6] ⇒ [4,5,6]`
- `[1..]`
- `take 3 [1..] ⇒ [1,2,3]`
- `reverse [1..6] ⇒ [6,5,4,3,2,1]`
- `splitAt 3 [1..6] ⇒ ([1,2,3],(4,5,6])`

Expressions in GHCi

- `3+4 ⇒ 7`
- `[1..6] ⇒ [1,2,3,4,5,6]`
- `[1..3] ++ [10..12] ⇒ [1,2,3,10,11,12]`
- `['a'..'z'] ⇒ "abcdefghijklmnopqrstuvwxyz"`
- `take 3 [1..6] ⇒ [1,2,3]`
- `drop 3 [1..6] ⇒ [4,5,6]`
- `[1..]`
- `take 3 [1..] ⇒ [1,2,3]`
- `reverse [1..6] ⇒ [6,5,4,3,2,1]`
- `splitAt 3 [1..6] ⇒ ([1,2,3],(4,5,6))`
- `fst (4, 'a')`

Expressions in GHCi

- `3+4 ⇒ 7`
- `[1..6] ⇒ [1,2,3,4,5,6]`
- `[1..3] ++ [10..12] ⇒ [1,2,3,10,11,12]`
- `['a'..'z'] ⇒ "abcdefghijklmnopqrstuvwxyz"`
- `take 3 [1..6] ⇒ [1,2,3]`
- `drop 3 [1..6] ⇒ [4,5,6]`
- `[1..]`
- `take 3 [1..] ⇒ [1,2,3]`
- `reverse [1..6] ⇒ [6,5,4,3,2,1]`
- `splitAt 3 [1..6] ⇒ ([1,2,3],(4,5,6])`
- `fst (4, 'a') ⇒ 4`

Functions

```
Prelude> let square x = x * x
Prelude>
```

Functions

```
Prelude> let square x = x * x
Prelude> square 4
16
```

Functions

```
Prelude> let square x = x * x
Prelude> square 4
16
Prelude> let greeting name = "Hello "++name++"!"
Prelude>
```

Functions

```
Prelude> let square x = x * x
Prelude> square 4
16
Prelude> let greeting name = "Hello "++name++"!"
Prelude> greeting "John"
"Hello John!"
```

Functions

```
Prelude> let square x = x * x
Prelude> square 4
16
Prelude> let greeting name = "Hello "++name++"!"
Prelude> greeting "John"
"Hello John!"
Prelude> let factorial n = product [1..n]
```

Functions

```
Prelude> let square x = x * x
Prelude> square 4
16
Prelude> let greeting name = "Hello "++name++"!"
Prelude> greeting "John"
"Hello John!"
Prelude> let factorial n = product [1..n]
Prelude> factorial 6
720
```

Functions

```
Prelude> let square x = x * x
Prelude> square 4
16
Prelude> let greeting name = "Hello "++name++"!"
Prelude> greeting "John"
"Hello John!"
Prelude> let factorial n = product [1..n]
Prelude> factorial 6
720
Prelude> let vlen (v1,v2) = sqrt (v1^2 + v2^2)
Prelude>
```

Functions

```
Prelude> let square x = x * x
Prelude> square 4
16
Prelude> let greeting name = "Hello "++name++"!"
Prelude> greeting "John"
"Hello John!"
Prelude> let factorial n = product [1..n]
Prelude> factorial 6
720
Prelude> let vlen (v1,v2) = sqrt (v1^2 + v2^2)
Prelude> vlen (3, 4)
5.0
```

Functions

```
Prelude> let square x = x * x
Prelude> square 4
16
Prelude> let greeting name = "Hello "++name++"!"
Prelude> greeting "John"
"Hello John!"
Prelude> let factorial n = product [1..n]
Prelude> factorial 6
720
Prelude> let vlen (v1,v2) = sqrt (v1^2 + v2^2)
Prelude> vlen (3, 4)
5.0
```

Do not define functions using `let` in scripts

Basic types

`Int`: integer number, 32 or 64 bits

```
1 :: Int
```

`Integer`: unbounded integer

```
99999999999999999 :: Integer
```

`Char`: `'a'`

`Bool`: **`True`** and **`False`**

Basic types

`Int`: integer number, 32 or 64 bits

```
1 :: Int
```

`Integer`: unbounded integer

```
99999999999999999 :: Integer
```

`Char`: `'a'`

`Bool`: **`True`** and **`False`**

`:t` prints the type of an expression in GHCi:

```
Prelude> :t 'a'
'a' ::  Char
Prelude> maxBound ::  Int
9223372036854775807
```

Compound types and `type`

- Lists: combine values of one type
  - Every list is constructed from `[]` and `:`

```
[3,5,7] == 3:5:7:[]
```

Compound types and `type`

- Lists: combine values of one type
  - Every list is constructed from `[]` and `:`

    ```
    [3,5,7] == 3:5:7:[]
    ```

  - Examples:

    ```
    [1,2,3,4,5] :: [Int]
    ['h','e','l','l','o'] :: [Char]
    "hello" :: [Char]
    ```

Compound types and `type`

- Lists: combine values of one type
  - Every list is constructed from `[]` and `:`

    ```
    [3,5,7] == 3:5:7:[]
    ```
  - Examples:

    ```
    [1,2,3,4,5] :: [Int]
    ['h','e','l','l','o'] :: [Char]
    "hello" :: [Char]
    ```
- Tuples: combine values of different types:

  ```
  ("Paul", 42) :: ([Char], Int)
  (3.14, ('a', 'b')) :: (Double, (Char, Char))
  ```

Compound types and `type`

- Lists: combine values of one type
  - Every list is constructed from `[]` and `:`

    ```
    [3,5,7] == 3:5:7:[]
    ```

  - Examples:

    ```
    [1,2,3,4,5] :: [Int]
    ['h','e','l','l','o'] :: [Char]
    "hello" :: [Char]
    ```

- Tuples: combine values of different types:

  ```
  ("Paul", 42) :: ([Char], Int)
  (3.14, ('a', 'b')) :: (Double, (Char, Char))
  ```

- Make compound type more readable with a type *synonym*:

  ```
  type String = [Char]
  type Person = (String, Int)
  ```

Files and function types

- Create file *Example.hs*:

```haskell
module Example where

square :: Double -> Double
square x = x * x -- Wrong: "let square x = ..."

greeting :: String -> String
greeting name = "Hello " ++ name ++ "!"

factorial :: Integer -> Integer
factorial n  = product [1..n]
```

- In GHCi:

```
Prelude> :l(oad) Example.hs
*Example> :r(eload)
```

Type variables: parametric polymorphism

```
Prelude> :t fst
fst ::  (a, b) -> a

Prelude> :t snd
snd ::  (a, b) -> b

Prelude> :t head
head ::  [a] -> a

Prelude> :t tail
tail ::  [a] -> [a]

Prelude> :t zip
zip ::  [a] -> [b] -> [(a,b)]
```

Type classes introduction: relations

- What can be sorted? Type?

Type classes introduction: relations

- What can be sorted? Type?
- Class constraints, example:

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

Type classes introduction: relations

- What can be sorted? Type?
- Class constraints, example:

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

- **Eq**: Equality testing

```
(==), (/=) :: Eq a => a -> a -> Bool
```

Examples: `Int`, `Integer`, `Char`, `(Eq a) => [a]`

Type classes introduction: relations

- What can be sorted? Type?
- Class constraints, example:

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

- **Eq**: Equality testing

```
(==), (/=) :: Eq a => a -> a -> Bool
```

Examples: `Int`, `Integer`, `Char`, `(Eq a) => [a]`

- **Ord**: Ordering

```
(<), (<=), (>), (>=) :: Ord a => a -> a -> Bool
min, max :: Ord a => a -> a -> a
```

Examples: `Int`, `Integer`, `Char`, `(Ord a) => [a]`

Type classes: numbers

- **Num**: numeric types

```
(+), (-), (*) :: Num a => a -> a -> Bool
negate, abs, signum :: Num a => a -> a
```

  Examples: `Int`, `Integer`, `Double`, `Float`

Type classes: numbers

- **Num**: numeric types

```
(+), (-), (*) :: Num a => a -> a -> Bool
negate, abs, signum :: Num a => a -> a
```

  Examples: `Int`, `Integer`, `Double`, `Float`

- **Integral**: whole number types with integer division

```
div, mod :: Integral a => a -> a -> a
toInteger :: Integral a => Integer
```

  Examples: `Int`, `Integer`

```
fromIntegral :: (Integral a, Num b) => a -> b
```

Type classes: numbers

- **Num**: numeric types

```
(+), (-), (*) :: Num a => a -> a -> Bool
negate, abs, signum :: Num a => a -> a
```

  Examples: `Int`, `Integer`, `Double`, `Float`

- **Integral**: whole number types with integer division

```
div, mod :: Integral a => a -> a -> a
toInteger :: Integral a => Integer
```

  Examples: `Int`, `Integer`

```
fromIntegral :: (Integral a, Num b) => a -> b
```

- **Fractional**: types with real division

```
(/) :: Fractional a => a -> a -> a
```

  Examples: `Float`, `Double`

Type classes: numbers

- **Num**: numeric types

```
(+), (-), (*) :: Num a => a -> a -> Bool
negate, abs, signum :: Num a => a -> a
```

  Examples: `Int`, `Integer`, `Double`, `Float`

- **Integral**: whole number types with integer division

```
div, mod :: Integral a => a -> a -> a
toInteger :: Integral a => Integer
```

  Examples: `Int`, `Integer`

```
fromIntegral :: (Integral a, Num b) => a -> b
```

- **Fractional**: types with real division

```
(/) :: Fractional a => a -> a -> a
```

  Examples: `Float`, `Double`

- **Floating**: floating point types

```
sin, exp, sqrt :: Floating a => a -> a
```

  Examples: `Float`, `Double`

Type classes: conversion to/from strings

- **Show**: Types that can be converted *to* `String`

  ```
  show :: Show a => a -> String
  ```

  Examples: `Int`, `Integer`, `Char`, `(Show a) => [a]`

Type classes: conversion to/from strings

- **Show**: Types that can be converted *to* `String`

  ```
  show :: Show a => a -> String
  ```

  Examples: `Int`, `Integer`, `Char`, `(Show a) => [a]`

- **Read**: Types that can be converted *from* `String`

  ```
  read :: Read a => String -> a
  ```

Type classes

- *Type classes* are properties shared between *different* types
- *Type classes*: common *interface*

Type classes

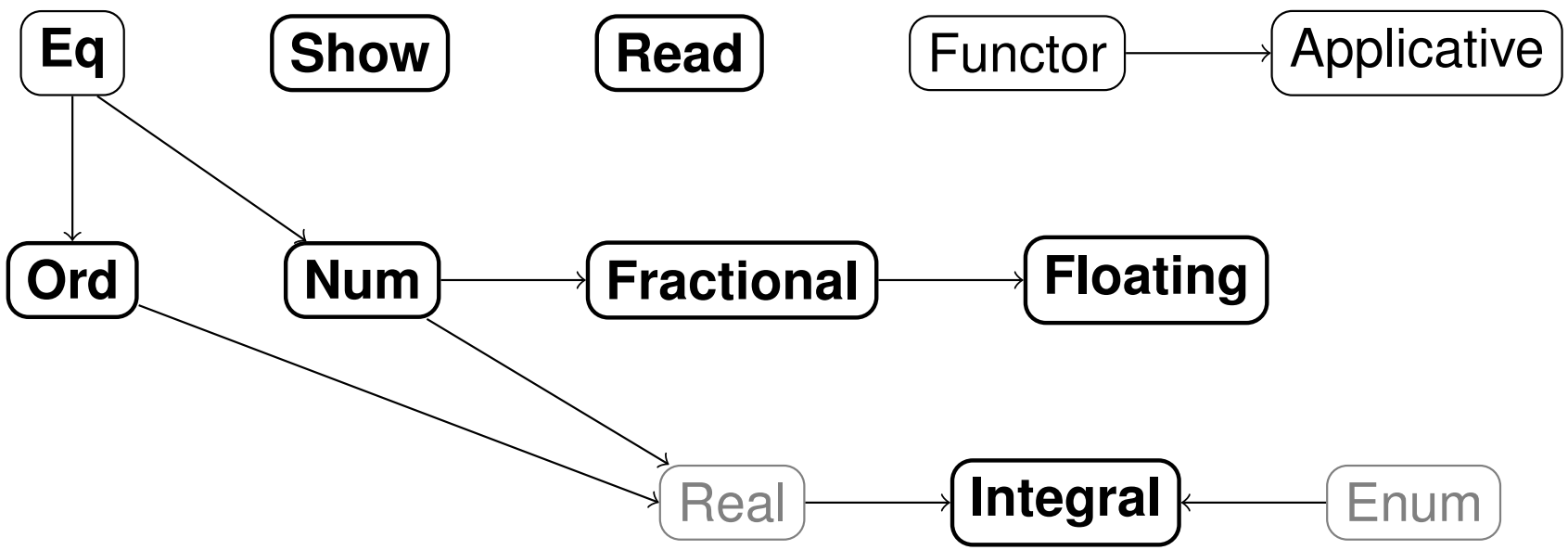
- *Type classes* are properties shared between *different* types
- *Type classes*: common *interface*
- Not the same as Java classes!
  - Closer to Java interfaces, but not the same!

Type classes

- *Type classes* are properties shared between *different* types
- *Type classes*: common *interface*
- Not the same as Java classes!
  - Closer to Java interfaces, but not the same!
- More related functions: `http://haskell.org/hoogle`

Type classes

- *Type classes* are properties shared between *different* types
- *Type classes*: common *interface*
- Not the same as Java classes!
  - Closer to Java interfaces, but not the same!
- More related functions: `http://haskell.org/hoogle`
- Type classes are related



Haskell evaluation: substitution

```
dbl x = x + x
```

Haskell evaluation: substitution

```
dbl x = x + x
```

Evaluation of an expression:

`dbl (1+2)` ⇒ ...

Haskell evaluation: substitution

```
dbl x = x + x
```

Evaluation of an expression:

```
dbl (1+2) ⇒ (1+2) + (1+2)
```

Haskell evaluation: substitution

```
dbl x = x + x
```

Evaluation of an expression:

```
dbl (1+2) ⇒ (1+2) + (1+2)
          ⇒ 3 + (1+2)
```

Haskell evaluation: substitution

```
dbl x = x + x
```

Evaluation of an expression:

```
dbl (1+2) ⇒ (1+2) + (1+2)
          ⇒ 3 + (1+2)
          ⇒ 3 + 3
```

Haskell evaluation: substitution

```
dbl x = x + x
```

Evaluation of an expression:

```
dbl (1+2) ⇒ (1+2) + (1+2)
          ⇒ 3 + (1+2)
          ⇒ 3 + 3
          ⇒ 6
```

Infinite lists

```
ones = 1 : ones
```

Infinite lists

```
ones = 1 : ones
```

Evaluation using substitution:

ones ⇒ ...

Infinite lists

```
ones = 1 : ones
```

Evaluation using substitution:

```
ones ⇒ 1 : ones
```

Infinite lists

```
ones = 1 : ones
```

Evaluation using substitution:

```
ones ⇒ 1 : ones
     ⇒ 1 : 1 : ones
```

Infinite lists

```
ones = 1 : ones
```

Evaluation using substitution:

```
ones ⇒ 1 : ones
     ⇒ 1 : 1 : ones
     ⇒ 1 : 1 : 1 : ones
```

Infinite lists

```
ones = 1 : ones
```

Evaluation using substitution:

```
ones ⇒ 1 : ones
     ⇒ 1 : 1 : ones
     ⇒ 1 : 1 : 1 : ones
     ⇒ 1 : 1 : 1 : 1 : ones
```

Infinite lists

```
ones = 1 : ones
```

Evaluation using substitution:

```
ones ⇒ 1 : ones
     ⇒ 1 : 1 : ones
     ⇒ 1 : 1 : 1 : ones
     ⇒ 1 : 1 : 1 : 1 : ones
               ...
```

Infinite lists

```
ones = 1 : ones
```

Evaluation using substitution:

```
ones ⇒ 1 : ones
     ⇒ 1 : 1 : ones
     ⇒ 1 : 1 : 1 : ones
     ⇒ 1 : 1 : 1 : 1 : ones
               ...
```

Lazy evaluation: calculations are postponed as long as possible:

```
take 3 ones == [1, 1, 1]
```

Name binding, pattern matching and substitution

- Pattern matching:

```
and True True   = True
and x    y      = False
```

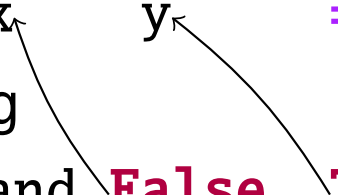
Name binding, pattern matching and substitution

- Pattern matching:

```
and True True   = True
and x    y      = False
```

- Name binding

```
x = and False  True
```

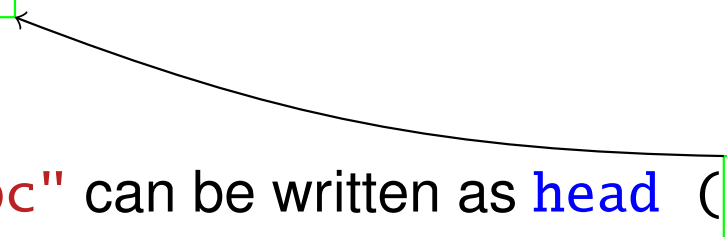


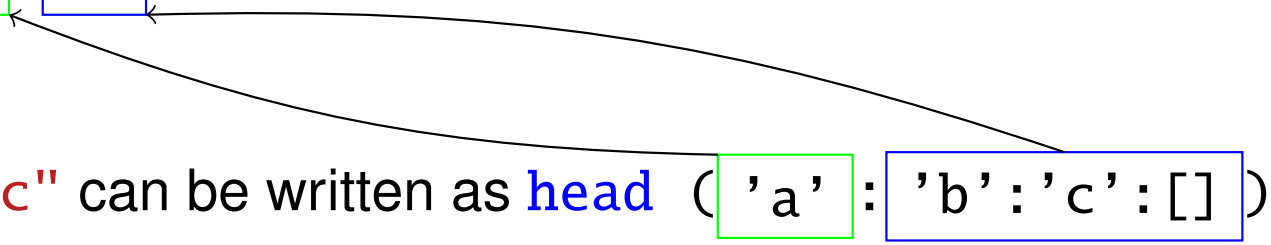
Pattern matching with lists

- ```
  head [] = error "empty list"
  head ( x : xs ) = x
  ```
  ```

Pattern matching with lists

- ```
  head [] = error "empty list"
  head ( x : xs ) = x
  ```

- `head "abc"` can be written as `head ( 'a' : 'b':'c':[] )`
  ```

Pattern matching with lists

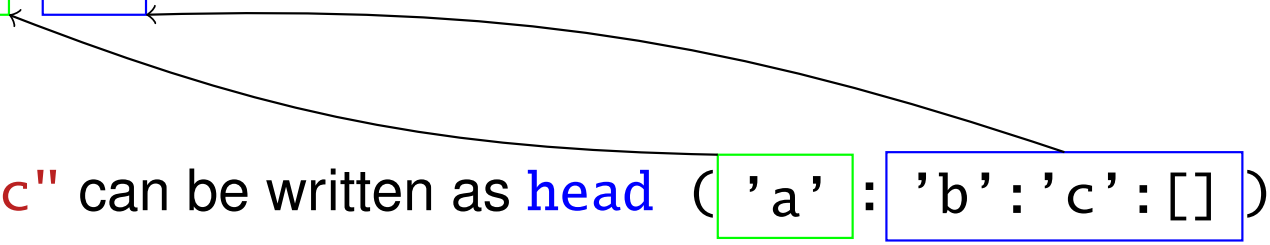
- ```
  head [] = error "empty list"
  head ( x : xs ) = x
  ```

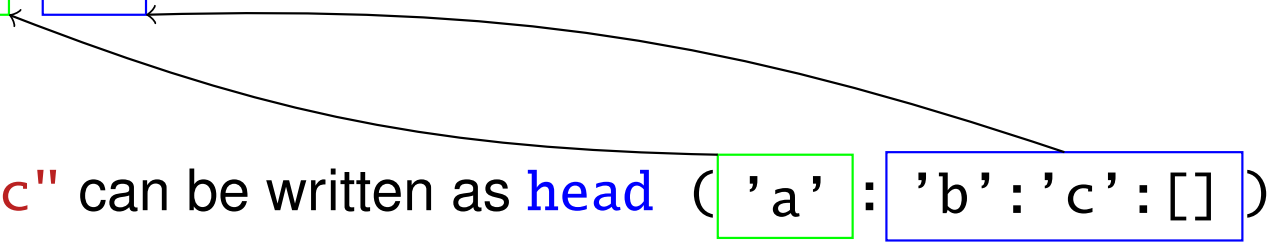


- `head "abc"` can be written as `head ( 'a' : 'b':'c':[] )`
  ```

Pattern matching with lists

- ```
  head [] = error "empty list"
  head ( x : xs ) = x
  ```



- `head "abc"` can be written as `head ( 'a' : 'b':'c':[] )`
  ```

Pattern matching with lists

- ```
  head [] = error "empty list"
  head ( x : xs ) = x
  ```



- `head "abc"` can be written as `head ( 'a' : 'b':'c':[] )`
- `tail (x:xs) = xs`
  ```

Pattern matching with lists

- ```
  head [] = error "empty list"
  head ( x : xs ) = x
  ```



- `head "abc"` can be written as `head ( 'a' : 'b':'c':[] )`
- `tail (x:xs) = xs`
- Alternative, no name binding:

  ```
  head (x:_) = x
  tail (_:xs) = xs
  ```
  ```

Recursion: as substitution

```
total 0 = 0
total n = total (n-1) + n
```

Recursion: as substitution

```
total 0 = 0
total n = total (n-1) + n
```

```
total 5 ⇒ ...
```

Recursion: as substitution

```
total 0 = 0
total n = total (n-1) + n
```

```
total 5 ⇒ total 4 + 5
```

Recursion: as substitution

```
total 0 = 0
total n = total (n-1) + n
```

```
total 5 ⇒ total 4 + 5
        ⇒ total 3 + 4 + 5
```

Recursion: as substitution

```
total 0 = 0
total n = total (n-1) + n
```

```
total 5 ⇒ total 4 + 5
        ⇒ total 3 + 4 + 5
        ⇒ total 2 + 3 + 4 + 5
```

Recursion: as substitution

```
total 0 = 0
total n = total (n-1) + n
```

```
total 5 ⇒ total 4 + 5
        ⇒ total 3 + 4 + 5
        ⇒ total 2 + 3 + 4 + 5
        ⇒ total 1 + 2 + 3 + 4 + 5
```

Recursion: as substitution

```
total 0 = 0
total n = total (n-1) + n
```

```
total 5 ⇒ total 4 + 5
        ⇒ total 3 + 4 + 5
        ⇒ total 2 + 3 + 4 + 5
        ⇒ total 1 + 2 + 3 + 4 + 5
        ⇒ total 0 + 1 + 2 + 3 + 4 + 5
```

Recursion: as substitution

```
total 0 = 0
total n = total (n-1) + n
```

```
total 5 ⇒ total 4 + 5
        ⇒ total 3 + 4 + 5
        ⇒ total 2 + 3 + 4 + 5
        ⇒ total 1 + 2 + 3 + 4 + 5
        ⇒ total 0 + 1 + 2 + 3 + 4 + 5
        ⇒    0    + 1 + 2 + 3 + 4 + 5
```

Recursion: as substitution

```
total 0 = 0
total n = total (n-1) + n
```

```
total 5 ⇒ total 4 + 5
        ⇒ total 3 + 4 + 5
        ⇒ total 2 + 3 + 4 + 5
        ⇒ total 1 + 2 + 3 + 4 + 5
        ⇒ total 0 + 1 + 2 + 3 + 4 + 5
        ⇒    0    + 1 + 2 + 3 + 4 + 5
        ⇒ ...
```

Recursion: as substitution

```
total 0 = 0
total n = total (n-1) + n
```

```
total 5 ⇒ total 4 + 5
        ⇒ total 3 + 4 + 5
        ⇒ total 2 + 3 + 4 + 5
        ⇒ total 1 + 2 + 3 + 4 + 5
        ⇒ total 0 + 1 + 2 + 3 + 4 + 5
        ⇒    0    + 1 + 2 + 3 + 4 + 5
        ⇒ ...
        ⇒ 15
```

Recursion: as substitution

```
total 0 = 0
total n = total (n-1) + n
```

```
total 5
```

Do not think in terms of substitution!

```
⇒     0     + 1 + 2 + 3 + 4 + 5
⇒ ...
⇒ 15
```

Proofs with mathematical induction

Prove:

$$\sum_{i=0}^{n} i \quad = \quad \frac{n\,(n+1)}{2}$$

Proofs with mathematical induction

Prove:

$$\sum_{i=0}^{n} i \quad = \quad \frac{n(n+1)}{2}$$

- Base case $n = 0$: ...
- Induction hypothesis: ...

- Induction case: $n - 1 \Rightarrow n$: ...

Proofs with mathematical induction

Prove:

$$\sum_{i=0}^{n} i \quad = \quad \frac{n\,(n+1)}{2}$$

- Base case $n = 0$: trivial
- Induction hypothesis: ...

- Induction case: $n - 1 \Rightarrow n$: ...

Proofs with mathematical induction

Prove:

$$\sum_{i=0}^{n} i \quad = \quad \frac{n\,(n+1)}{2}$$

- Base case $n = 0$: trivial
- Induction hypothesis:

$$\sum_{i=0}^{n-1} i \quad = \quad \frac{(n-1)*n}{2}$$

- Induction case: $n - 1 \Rightarrow n$: ...

Proofs with mathematical induction

Prove:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

- Base case $n = 0$: trivial
- Induction hypothesis:

$$\sum_{i=0}^{n-1} i = \frac{(n-1)*n}{2}$$

- Induction case: $n - 1 \Rightarrow n$:

$$\begin{aligned}
\sum_{i=0}^{n} i &= \left(\sum_{i=0}^{n-1} i\right) + n \\
&= \frac{(n-1)*n}{2} + n \quad (IH) \\
&= \frac{n(n+1)}{2}
\end{aligned}$$

Recursion: mathematical induction

- Add: $0, 1, 2, \ldots n$

  Thus, define: `total n = 0+1+2+...+n`

Recursion: mathematical induction

- Add: $0, 1, 2, \ldots n$

  Thus, define: `total n = 0+1+2+...+n`

- With recursion ($\approx$ induction):
  - Base case: `total 0 = ...`
  - Assume (!): `total(n-1) = 0+1+2+...+(n-1)` (IH)
  - Recursion case: `total n = ...`

Recursion: mathematical induction

- Add: $0, 1, 2, \ldots n$

  Thus, define: `total n = 0+1+2+...+n`

- With recursion ($\approx$ induction):
  - Base case: `total 0 = 0`
  - Assume (!): `total(n-1) = 0+1+2+...+(n-1)` (IH)
  - Recursion case: `total n = ...`

Recursion: mathematical induction

- Add: $0, 1, 2, \ldots n$

  Thus, define: `total n = 0+1+2+...+n`

- With recursion ($\approx$ induction):
  - Base case: `total 0 = 0`
  - Assume (!): `total(n-1) = 0+1+2+...+(n-1)` (IH)
  - Recursion case: `total n = ... total(n-1) ...`

Recursion: mathematical induction

- Add: $0, 1, 2, \ldots n$

  Thus, define: `total n = 0+1+2+...+n`

- With recursion ($\approx$ induction):
  - Base case: `total 0 = 0`
  - Assume (!): `total(n-1) = 0+1+2+...+(n-1)` (IH)
  - Recursion case: `total n = total(n-1) + n`

Recursion: mathematical induction

- Add: $0, 1, 2, \ldots n$

  Thus, define: `total n = 0+1+2+...+n`

- With recursion ($\approx$ induction):
  - Base case: `total 0 = 0`
  - Assume (!): `total(n-1) = 0+1+2+...+(n-1)` (IH)
  - Recursion case: `total n = total(n-1) + n`

- Thus:

```
total 0 = 0
total n = total (n-1) + n
```

Recursion: mathematical induction

- Add: $0, 1, 2, \ldots n$

  Thus, define: `total n = 0+1+2+...+n`

- With recursion ($\approx$ induction):
  - Base case: `total 0 = 0`
  - Assume (!): `total(n-1) = 0+1+2+...+(n-1)` (IH)
  - Recursion case: `total n = total(n-1) + n`

- Thus:

```
total 0 = 0
total n = total (n-1) + n
```

- Evaluate: `total 10`

Recursion: mathematical induction

- Add: $0, 1, 2, \ldots n$

  Thus, define: `total n = 0+1+2+...+n`

- With recursion ($\approx$ induction):
  - Base case: `total 0 = 0`
  - Assume (!): `total(n-1) = 0+1+2+...+(n-1)` (IH)
  - Recursion case: `total n = total(n-1) + n`

- Thus:

```
total 0 = 0
total n = total (n-1) + n
```

- Evaluate: `total 10`
- Test: `total 10 == 10 * 11 / 2`

Recursion over lists

```
lngth xs = ???
```

Recursion over lists

```
lngth xs = ???

lngth []     =
lngth (x:xs) = ... lngth xs ...
```

Recursion over lists

```
lngth xs = ???

lngth []     = 0
lngth (x:xs) = ... lngth xs ...
```

Recursion over lists

```
lngth xs = ???

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

List comprehension

Resembles set generator notation in mathematics

- Mathematics: $\left\{ x^2 \mid x \in \{1, \ldots, 10\},\ even(x) \right\}$

List comprehension

Resembles set generator notation in mathematics

- Mathematics: $\left\{ x^2 \mid x \in \{1, \ldots, 10\},\ even(x) \right\}$
- `[ x^2 | x <- [1..10] , x `mod` 2 == 0 ]`

List comprehension

Resembles set generator notation in mathematics

- Mathematics: $\left\{ x^2 \mid x \in \{1, \ldots, 10\},\ even(x) \right\}$
- `[ x^2 | x <- [1..10] , x `mod` 2 == 0 ]`
- `[ (x,y)  | x <- [1..6] , y <- [1..6] , y > x ]`

List comprehension

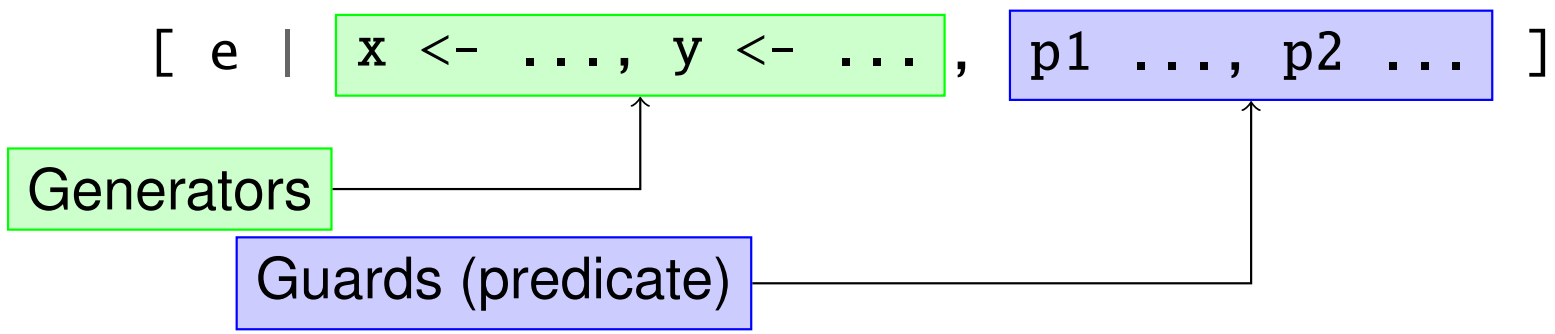
Resembles set generator notation in mathematics

- Mathematics: $\left\{ x^2 \mid x \in \{1, \ldots, 10\},\ even(x) \right\}$
- `[ x^2 | x <- [1..10] , x `mod` 2 == 0 ]`
- `[ (x,y) | x <- [1..6] , y <- [1..6] , y > x ]`
- `[ x+y | (x,y) <- zip [1..6] [1..6] ]`

List comprehension

Resembles set generator notation in mathematics

- Mathematics: $\left\{ x^2 \mid x \in \{1,\ldots,10\},\ even(x) \right\}$
- `[ x^2 | x <- [1..10] , x `mod` 2 == 0 ]`
- `[ (x,y) | x <- [1..6] , y <- [1..6] , y > x ]`
- `[ x+y | (x,y) <- zip [1..6] [1..6] ]`
- General structure



Where-clauses

```
fun x y = a + b
          where
            a = x + y
            b = x * y
```

offside-rule

Alternative syntax for conditional expressions

- **if** expression

```
max a b = if a > b then a else b
```

Alternative syntax for conditional expressions

- **if** expression

```
max a b = if a > b then a else b
```

- **guards**

```
signum x
    | x < 0      = -1
    | x == 0     = 0
    | otherwise = 1
```

Alternative syntax for conditional expressions

- **if** expression

```
max a b = if a > b then a else b
```

- **guards**

```
signum x
    | x < 0     = -1
    | x == 0    = 0
    | otherwise = 1
```

- **case**

```
capital c = case c of
   "the Netherlands"   -> "Amsterdam"
   "Germany"           -> "Berlin"
   "England"           -> "London"
                ...
```

Example: concattenation of lists

Goal: define the function

```
concat :: [[a]] -> [a]
```

Example: `concat [[1,2,3],[],[4,5]] == [1,2,3,4,5]`

Example: concattenation of lists

Goal: define the function

```
concat :: [[a]] -> [a]
```

Example: `concat [[1,2,3],[],[4,5]] == [1,2,3,4,5]`

- Using recursion:

```
concat' []       = []
concat' (xs:xss) = xs ++ concat' xss
```

Example: concattenation of lists

Goal: define the function

```
concat :: [[a]] -> [a]
```

Example: `concat [[1,2,3],[],[4,5]] == [1,2,3,4,5]`

- Using recursion:

```
concat' []       = []
concat' (xs:xss) = xs ++ concat' xss
```

- Using list comprehension:

```
concat'' xss = [ x | xs <- xss, x <- xs ]
```

Testing: QuickCheck (1/3)

- **Proposition**: checks specification for a given input

```
prop_concat'' xs = concat'' xs == concat xs
```

Testing: QuickCheck (1/3)

- **Proposition**: checks specification for a given input

```
prop_concat'' xs = concat'' xs == concat xs
```

```
*Main> prop_concat” [[1],[3,4]]
True
```

Testing: QuickCheck (1/3)

- **Proposition**: checks specification for a given input

```
prop_concat'' xs = concat'' xs == concat xs
```

```
*Main> prop_concat" [[1],[3,4]]
True
```

- Test the proposition for random inputs:

```
import Test.QuickCheck
concat'' xss = [ x | xs <- xss, x <- xs ]
prop_concat'' xss = concat'' xss == concat xss
```

Testing: QuickCheck (1/3)

- **Proposition**: checks specification for a given input

```
prop_concat'' xs = concat'' xs == concat xs
```

```
*Main> prop_concat” [[1],[3,4]]
True
```

- Test the proposition for random inputs:

```
import Test.QuickCheck
concat'' xss = [ x | xs <- xss, x <- xs ]
prop_concat'' xss = concat'' xss == concat xss
```

```
*Main> :load lec1.hs
*Main> quickCheck prop_concat”
+++ OK, passed 100 tests.
```

Testing: QuickCheck (2/3)

- Incorrect specification

```
concat_wrong [[]]       = []
concat_wrong (xs:xss) = xs ++ concat_wrong xss
```

Testing: QuickCheck (2/3)

- Incorrect specification

```
concat_wrong [[]]       = []
concat_wrong (xs:xss) = xs ++ concat_wrong xss
```

- QuickCheck proposition

```
prop_concat_wrong xss = concat_wrong xss == concat xss
```

Testing: QuickCheck (2/3)

- Incorrect specification

```
concat_wrong [[]]       = []
concat_wrong (xs:xss) = xs ++ concat_wrong xss
```

- QuickCheck proposition

```
prop_concat_wrong xss = concat_wrong xss == concat xss
```

```
*Main> quickCheck prop_concat_wrong
*** Failed!  (after 1 test):
Exception:
lec1.hs:(11,1)-(12,46):  Non-exhaustive patterns
in function concat_wrong
[]
```

Testing: QuickCheck (2/3)

- Incorrect specification

```
concat_wrong [[]]       = []
concat_wrong (xs:xss) = xs ++ concat_wrong xss
```

- QuickCheck proposition

```
prop_concat_wrong xss = concat_wrong xss == concat xss
```

```
*Main> quickCheck prop_concat_wrong
*** Failed!  (after 1 test):
Exception:
lec1.hs:(11,1)-(12,46):  Non-exhaustive patterns
in function concat_wrong
[]
```

- Counter example: `[]`

Testing: QuickCheck (3/3)

```haskell
{-# LANGUAGE TemplateHaskell #-}
module Lecture1 where

import Test.QuickCheck
import Test.QuickCheck.All

-- your tests go here, e.g.:
prop_reverse xs  =    xs == reverse (reverse xs)
prop_sort xs = ...

-- QuickCheck collects all prop_* tests
return []
check = $quickCheckAll
```

Trace

- Add: `import Debug.Trace`

Trace

- Add: `import Debug.Trace`
- `trace :: String -> a -> a`

Trace

- Add: `import Debug.Trace`
- `trace :: String -> a -> a`
- Example:

```
sum [x]    = x
sum (x:xs) = trace ("Intermediate result: "
                    ++ show s) (x + s)
  where s = sum xs
```

Trace

- Add: `import Debug.Trace`
- `trace :: String -> a -> a`
- Example:

```
sum [x]    = x
sum (x:xs) = trace ("Intermediate result: "
                    ++ show s) (x + s)
  where s = sum xs
```

- GHCi:

```
*Lecture1 Debug.Trace> sum [1..3]
Intermediate result:  3
Intermediate result:  5
6
```

Function application

Space is (like) an operator: function application:

```
f x + f 1
```

Function application

Space is (like) an operator: function application:

```
f x + f 1
```

Left associative with highest precedence, thus equal to:

```
(f 3) + (f 1)
```

Function application

Space is (like) an operator: function application:

```
f x + f 1
```

Left associative with highest precedence, thus equal to:

```
(f 3) + (f 1)
```

Right associative function application with `$`

```
f $ 3 + f 1 == f (3 + (f 1))
```

Currying

- Function of two arguments

```
max :: (Ord a) => a -> a -> a
```

  Receives two arguments, it produces the maximum

Currying

- Function of two arguments

```
max :: (Ord a) => a -> a -> a
```

  Receives two arguments, it produces the maximum

- Currying:

```
max :: (Ord a) => a -> (a -> a)
```

  Receives *one* argument, it produces a new function (currying)

Currying

- Function of two arguments

```
max :: (Ord a) => a -> a -> a
```

  Receives two arguments, it produces the maximum

- Currying:

```
max :: (Ord a) => a -> (a -> a)
```

  Receives *one* argument, it produces a new function (currying)

- Example:

```
max 10 20 == (max 10) 20
```

Sectioning and infixing

- For any (binary) *operator* $*$
  $(*)$, $(a*)$, $(*a)$ are *functions*
  - `(++"ing") "infix" == "infix" ++ "ing"`
  - `(*) 5 6 = 5 * 6`
  - *Sectioning*: operator between brackets ( ) $\Rightarrow$ Function

Sectioning and infixing

- For any (binary) *operator* $*$
  $(*)$, $(a*)$, $(*a)$ are *functions*
  - `(++"ing") "infix" == "infix" ++ "ing"`
  - `(*) 5 6 = 5 * 6`
  - *Sectioning*: operator between brackets ( ) ⇒ Function
- For any (binary) *function* `f`
  ‘`f`‘ is an (infix, binary) *operator*
  - `elem 'a' "Hello" == 'a' `elem` "Hello"`
  - `mod 10 3 === 10 `mod` 3`
  - *Infixing*: function between backquotes ‘ ⇒ Operator

Higher-order functions

- Functions as an argument

Higher-order functions

- Functions as an argument
- `map` applies a given function to each list element:

```
map f [x1,x2,...,xn] = [f x1, f x2, ..., f xn]
```

Higher-order functions

- Functions as an argument
- `map` applies a given function to each list element:

  ```
  map f [x1,x2,...,xn] = [f x1, f x2, ..., f xn]
  ```

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

Higher-order functions

- Functions as an argument
- `map` applies a given function to each list element:

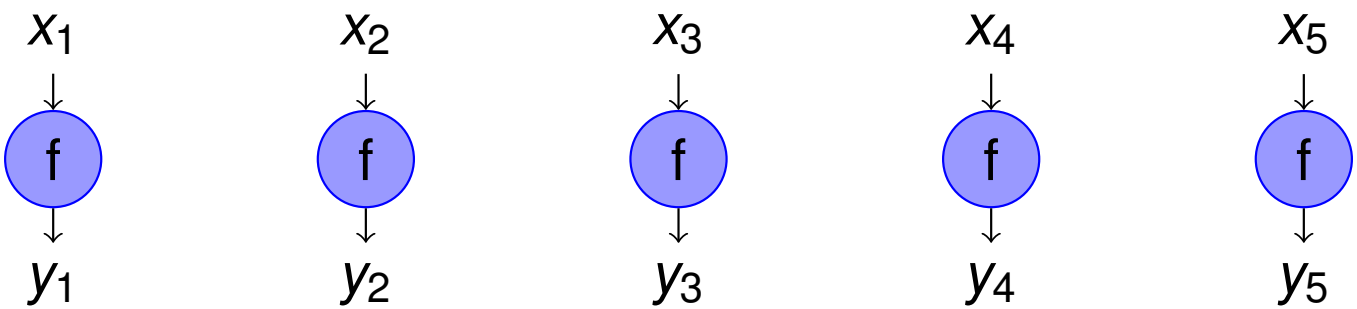
  ```
  map f [x1,x2,...,xn] = [f x1, f x2, ..., f xn]
  ```

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

```
Prelude> map (*2) [1,2,3]
[2,4,6]
```

Higher order functions: map

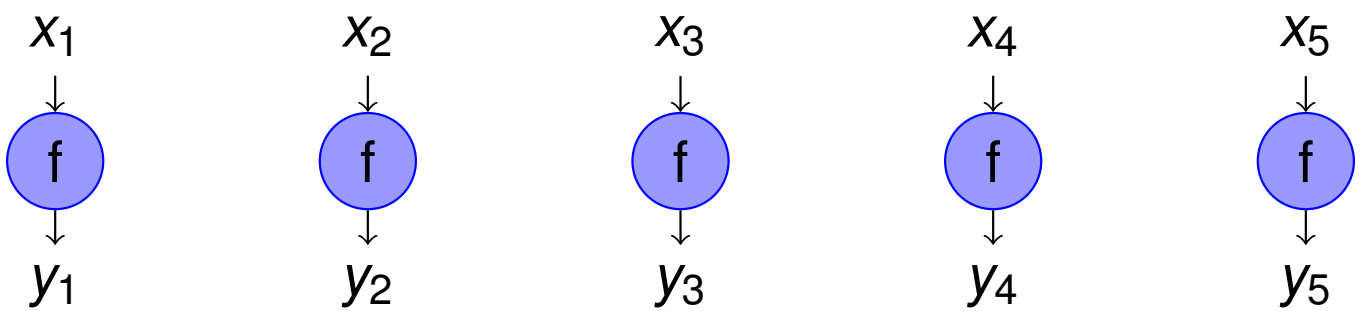
```
map :: (a -> b) -> [a] -> [b]
```




```
ys = map f xs
```

Higher order functions: map

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

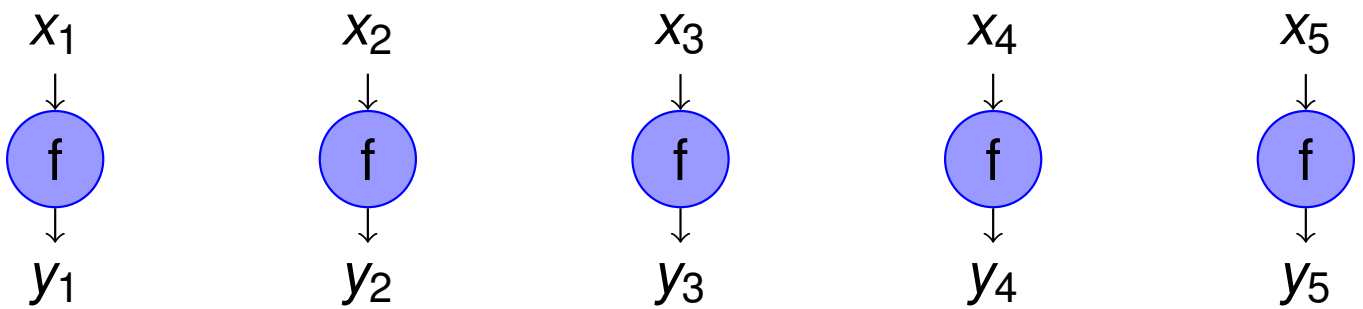



```
ys = map f xs
```

```
map (+10) [1,2,3] == [11,12,13]
```

Higher order functions: map

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



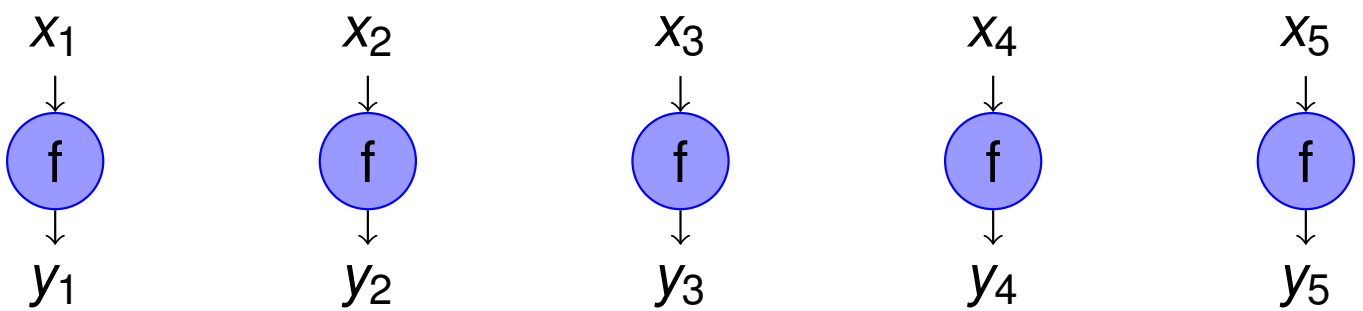

```
ys = map f xs
```

```
map (+10) [1,2,3] == [11,12,13]
```

```
map reverse ["evil", "rats"] == ["live", "star"]
```

Higher order functions: map

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




```
ys = map f xs
```

```
map (+10) [1,2,3] == [11,12,13]

map reverse ["evil", "rats"] == ["live", "star"]

map chr [72,97,115,107,101,108,108] == "Haskell"
```

Example: intersperse

```
Prelude> :m Data.List
Prelude Data.List> :t intersperse
intersperse ::  a -> [a] -> [a] Prelude Data.List>
intersperse ',' "abcde" "a,b,c,d,e"
```

- Base case: intersperse of an empty list?

  `intersperse c [] = ...`

Example: intersperse

```
Prelude> :m Data.List
Prelude Data.List> :t intersperse
intersperse ::  a -> [a] -> [a] Prelude Data.List>
intersperse ',' "abcde" "a,b,c,d,e"
```

- Base case: intersperse of an empty list?
  `intersperse c [] = []`
- Base case: list with one element?
  `intersperse c [x] = ...`

Example: intersperse

```
Prelude> :m Data.List
Prelude Data.List> :t intersperse
intersperse ::  a -> [a] -> [a] Prelude Data.List>
intersperse ',' "abcde" "a,b,c,d,e"
```

- Base case: intersperse of an empty list?
  `intersperse c [] = []`
- Base case: list with one element?
  `intersperse c [x] = [x]`

Example: intersperse

```
Prelude> :m Data.List
Prelude Data.List> :t intersperse
intersperse ::  a -> [a] -> [a] Prelude Data.List>
intersperse ',' "abcde" "a,b,c,d,e"
```

- Base case: intersperse of an empty list?
  `intersperse c [] = []`
- Base case: list with one element?
  `intersperse c [x] = [x]`
- Recursive case: intersperse of
  `intersperse c (x:xs) = ...  intersperse c xs ...`

Example: intersperse

```
Prelude> :m Data.List
Prelude Data.List> :t intersperse
intersperse ::  a -> [a] -> [a] Prelude Data.List>
intersperse ',' "abcde" "a,b,c,d,e"
```

- Base case: intersperse of an empty list?
  `intersperse c [] = []`
- Base case: list with one element?
  `intersperse c [x] = [x]`
- Recursive case: intersperse of
  `intersperse c (x:xs) = x : c : intersperse c xs`

Reference of useful functions (lists)

```
:  :: a -> [a] -> [a]    -- add element in front of list
++ :: [a] -> [a] -> [a] -- list concattenation
\\ :: [a] -> [a] -> [a] -- list subtraction
head, last :: [a] -> a
tail, init :: [a] -> [a]
reverse    :: [a] -> [a]
(!!)       :: [a] -> Int -> a  -- list indexing
length     :: [a] -> Int
drop, take :: Int -> [a] -> [a]
zip        :: [a] -> [b] -> [(a,b)]

filter     :: (a->Bool) -> [a] -> [a]
map        :: (a->b) -> [a] -> [b]
zipWith    :: (a->b->c) -> [a] -> [b] -> [c]
```

Reference of useful functions

```
chr :: Int -> Char
ord :: Char -> Int

fst :: (a,b) -> a
snd :: (a,b) -> b

fromIntegral :: (Num b, Integral a) => a -> b
fromInteger  :: Num a => Integer -> a
toInteger    :: Integral a => a -> Integer

ceiling  :: (RealFrac a, Integral b) => a -> b
floor    :: (RealFrac a, Integral b) => a -> b
truncate :: (RealFrac a, Integral b) => a -> b
round    :: (RealFrac a, Integral b) => a -> b
```

Type errors

```
Prelude> 1+'a'
```

Type errors

```
Prelude> 1+'a'


<interactive>:14:2:
No instance for (Num Char) arising from a use of '+'
In the expression: 1 + 'a'
In an equation for 'it': it = 1 + 'a'
```

Type errors

```
Prelude> 1+'a'


<interactive>:14:2:
No instance for (Num Char) arising from a use of '+'
In the expression: 1 + 'a'
In an equation for 'it': it = 1 + 'a'
```

Important notes for practical session

- Do not use solutions from pearl!
- Use QuickCheck
  - Testing tool, not for trial-and-error development
  - Can only be used to prove *incorrectness*
- Work in pairs, help eachother (but do not copy)
- Sign-off in time to avoid queues
- Third series already available; continued in block 2

Self study

- This lecture: “Introduction to Functional Programming”
  - Read: Appendix A-2 from the module guide (QuickCheck)
  - Learn you a Haskell (Lipovaca): Chapters 1–5
  - Programming in Haskell (Hutton): Chapters 1–6

Self study

- This lecture: “Introduction to Functional Programming”
  - Read: Appendix A-2 from the module guide (QuickCheck)
  - Learn you a Haskell (Lipovaca): Chapters 1–5
  - Programming in Haskell (Hutton): Chapters 1–6
- Next lecture: “Higher-Order Functions”
  - Learn you a Haskell (Lipovaca): Chapter 6
  - Programming in Haskell (Hutton): Chapter 7