Content

Introduction

Lists and tuples

Functions

Ranges and list comprehensions

Anonymous functions, filter, map, foldr, foldl, zip, zipWith

Types

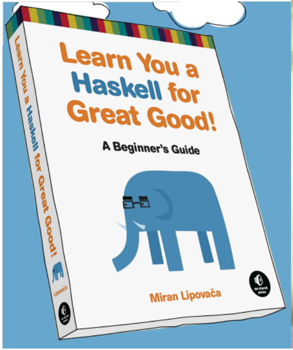
Tips 'n Tricks

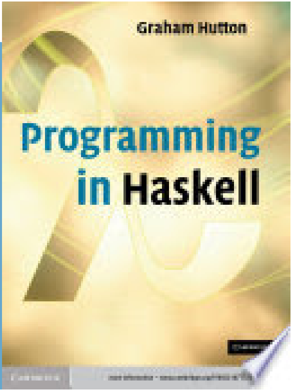
Haskell

- Named after Haskell Curry
- The 90s
- Purely functional
- Lazy
- Strong and statically typed

Learning Haskell – recommended reading

- Learn You a Haskell for Great Good![1] http://learnyouahaskell.com/chapters
- Programming in Haskell http://www.cs.nott.ac.uk/~pszgmh/pih.html





[1] Some images from this presentation come from the book.

Starting Haskell

Open a terminal and start the interactive compiler **ghci**

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Prelude>
```

Simple operations

-
  ```
  Prelude> 5+6
  11
  ```

Simple operations

-
  ```
  Prelude> 5+6
  11
  ```

-
  ```
  Prelude> 3.5*3
  10.5
  ```

Simple operations

-
  ```
  Prelude> 5+6
  11
  ```

-
  ```
  Prelude> 3.5*3
  10.5
  ```

-
  ```
  Prelude> 4/3
  1.3333333333333333
  ```

Simple operations

-
  ```
  Prelude> 5+6
  11
  ```

-
  ```
  Prelude> 3.5*3
  10.5
  ```

-
  ```
  Prelude> 4/3
  1.3333333333333333
  ```

-
  ```
  Prelude> 2**8
  256.0
  ```

Simple operations

-
  ```
  Prelude> 5+6
  11
  ```

-
  ```
  Prelude> 3.5*3
  10.5
  ```

-
  ```
  Prelude> 4/3
  1.3333333333333333
  ```

-
  ```
  Prelude> 2**8
  256.0
  ```

-
  ```
  Prelude> 8/2
  4.0
  ```

Simple operations

-
  ```
  Prelude> 5+6
  11
  ```

-
  ```
  Prelude> 3.5*3
  10.5
  ```

-
  ```
  Prelude> 4/3
  1.3333333333333333
  ```

-
  ```
  Prelude> 2**8
  256.0
  ```

-
  ```
  Prelude> 8/2
  4.0
  ```

-
  ```
  Prelude> 3*3
  9
  ```

Simple operations

-
  ```
  Prelude> 5+6
  11
  ```

-
  ```
  Prelude> 3.5*3
  10.5
  ```

-
  ```
  Prelude> 4/3
  1.3333333333333333
  ```

-
  ```
  Prelude> 2**8
  256.0
  ```

-
  ```
  Prelude> 8/2
  4.0
  ```

-
  ```
  Prelude> 3*3
  9
  ```

-
  ```
  Prelude> 2-4
  -2
  ```

Simple operations

-
  ```
  Prelude> 5+6
  11
  ```

-
  ```
  Prelude> 3.5*3
  10.5
  ```

-
  ```
  Prelude> 4/3
  1.3333333333333333
  ```

-
  ```
  Prelude> 2**8
  256.0
  ```

-
  ```
  Prelude> 8/2
  4.0
  ```

-
  ```
  Prelude> 3*3
  9
  ```

-
  ```
  Prelude> 2-4
  -2
  ```

-
  ```
  Prelude> True
  True
  ```

Simple operations (2)

-
  ```
  Prelude> False
  False
  ```

Simple operations (2)

-
  ```
  Prelude> False
  False
  ```

-
  ```
  Prelude> 3 - 6 == -3
  True
  ```

Simple operations (2)

-
  ```
  Prelude> False
  False
  ```

-
  ```
  Prelude> 3 - 6 == -3
  True
  ```

-
  ```
  Prelude> 4 + 4 /= 8
  False
  ```

Simple operations (2)

-
  ```
  Prelude> False
  False
  ```

-
  ```
  Prelude> 3 - 6 == -3
  True
  ```

-
  ```
  Prelude> 4 + 4 /= 8
  False
  ```

-
  ```
  Prelude> True & True
  <interactive>:16:6: Not in scope: `&'
  ```

Simple operations (2)

-
  ```
  Prelude> False
  False
  ```

-
  ```
  Prelude> 3 - 6 == -3
  True
  ```

-
  ```
  Prelude> 4 + 4 /= 8
  False
  ```

-
  ```
  Prelude> True & True
  <interactive>:16:6: Not in scope: `&'
  ```

-
  ```
  Prelude> True && True
  True
  ```

Simple operations (2)

-
  ```
  Prelude> False
  False
  ```

-
  ```
  Prelude> 3 - 6 == -3
  True
  ```

-
  ```
  Prelude> 4 + 4 /= 8
  False
  ```

-
  ```
  Prelude> True & True
  <interactive>:16:6: Not in scope: `&'
  ```

-
  ```
  Prelude> True && True
  True
  ```

-
  ```
  Prelude> True || False
  True
  ```

Simple operations (2)

-
  ```
  Prelude> False
  False
  ```

-
  ```
  Prelude> 3 - 6 == -3
  True
  ```

-
  ```
  Prelude> 4 + 4 /= 8
  False
  ```

-
  ```
  Prelude> True & True
  <interactive>:16:6: Not in scope: `&'
  ```

-
  ```
  Prelude> True && True
  True
  ```

-
  ```
  Prelude> True || False
  True
  ```

-
  ```
  Prelude> / True
  <interactive>:19:1: parse error on input `/'
  ```

Simple operations (2)

-
  ```
  Prelude> False
  False
  ```

-
  ```
  Prelude> 3 - 6 == -3
  True
  ```

-
  ```
  Prelude> 4 + 4 /= 8
  False
  ```

-
  ```
  Prelude> True & True
  <interactive>:16:6: Not in scope: `&'
  ```

-
  ```
  Prelude> True && True
  True
  ```

-
  ```
  Prelude> True || False
  True
  ```

-
  ```
  Prelude> / True
  <interactive>:19:1: parse error on input `/'
  ```

-
  ```
  Prelude> not True
  False
  ```

Variables, comments, directories and modules

```
Prelude> let foo = 3
Prelude> foo + 3
6
```

Variables, comments, directories and modules

```
Prelude> let foo = 3
Prelude> foo + 3
6
```

FooModule.hs

```haskell
module FooModule where
-- Here variable bar is given value 4.
bar = 4
```

Variables, comments, directories and modules

```
Prelude> let foo = 3
Prelude> foo + 3
6
```

FooModule.hs

```haskell
module FooModule where
-- Here variable bar is given value 4.
bar = 4
```

```
Prelude> :load FooModule.hs
[1 of 1] Compiling FooModule ( FooModule.hs, interpreted )
Ok, modules loaded: FooModule.

*FooModule> foo + bar
7
```

Prefix and infix functions

Prefix functions

`not`

Prefix functions with two arguments can be used as infix functions:

```
*Lecture> add 2 3
5
*Lecture> 2 `add` 3
5
```

Infix functions

`+ - * / ** && || == /=`

Infix functions can be used as prefix functions:

```
*Lecture> (+) 2 3
5
```

Our own addition function

Lecture.hs

```haskell
module Lecture where

add :: Integer -> Integer -> Integer
add a b = a + b
```

or directly in the interpreter:

```
let add a b = a + b
```

What does the following function do?

Lecture.hs

```
foo :: Integer -> Integer
foo 0 = 1
foo n = n * foo (n-1)
```

What does the following function do?

Lecture.hs

```haskell
foo :: Integer -> Integer
foo 0 = 1
foo n = n * foo (n-1)
```

```
*Lecture> foo 0
1
*Lecture> foo 3
6
```

Content

Introduction

Lists and tuples

Functions

Ranges and list comprehensions

Anonymous functions, filter, map, foldr, foldl, zip, zipWith

Types

Tips 'n Tricks

Lists

- Empty list:

```
Prelude> []
[]
```

Lists

- Empty list:

```
Prelude> []
[]
```

- List with 2 elements:

```
Prelude> [1,2]
[1,2]
```

Lists

- Empty list:

```
Prelude> []
[]
```

- List with 2 elements:

```
Prelude> [1,2]
[1,2]
```

- Adding an element to the list:

```
Prelude> 1 : [2,3]
[1,2,3]
```

Lists

- Empty list:

```
Prelude> []
[]
```

- List with 2 elements:

```
Prelude> [1,2]
[1,2]
```

- Adding an element to the list:

```
Prelude> 1 : [2,3]
[1,2,3]
```

- Internal representation:

```
Prelude> 1 : 2 : 3 : []
[1,2,3]
```

Lists (2)

- List concatenation:

```
Prelude> [1,2] ++ [3,4]
[1,2,3,4]
```

Lists (2)

- List concatenation:

```
Prelude> [1,2] ++ [3,4]
[1,2,3,4]
```

- List of characters:

```
Prelude> ['a','b','c']
"abc"
```

Lists (2)

- List concatenation:

```
Prelude> [1,2] ++ [3,4]
[1,2,3,4]
```

- List of characters:

```
Prelude> ['a','b','c']
"abc"
```

- String concatenation:

```
Prelude> "Hello" ++ " " ++ "World"
"Hello World"
```

Lists (2)

- List concatenation:

```
Prelude> [1,2] ++ [3,4]
[1,2,3,4]
```

- List of characters:

```
Prelude> ['a','b','c']
"abc"
```

- String concatenation:

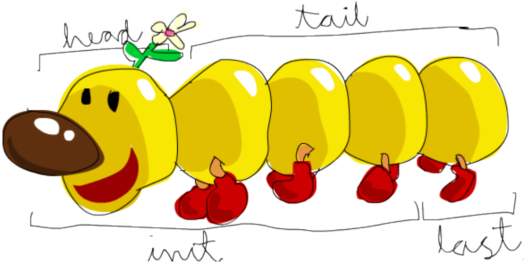
```
Prelude> "Hello" ++ " " ++ "World"
"Hello World"
```

- Length of a list:

```
Prelude> length [1,2,3,4,5,6]
6
```

Lists: head, last, init and tail

- `Prelude> head [1,2,3,4,5]`
  `1`
- `Prelude> last [1,2,3,4,5]`
  `5`
- `Prelude> tail [1,2,3,4,5]`
  `[2,3,4,5]`
- `Prelude> init [1,2,3,4,5]`
  `[1,2,3,4]`



Lists: reverse, !!, null, take

- Reverse a list

  ```
  Prelude> reverse [1,2,3,4,5]
  [5,4,3,2,1]
  ```

Lists: reverse, !!, null, take

- Reverse a list

  ```
  Prelude> reverse [1,2,3,4,5]
  [5,4,3,2,1]
  ```

- Retrieve the *n*-th element of a list

  ```
  Prelude> [1,2,3,4,5] !! 3
  4
  ```

Lists: reverse, !!, null, take

- Reverse a list

  ```
  Prelude> reverse [1,2,3,4,5]
  [5,4,3,2,1]
  ```

- Retrieve the *n*-th element of a list

  ```
  Prelude> [1,2,3,4,5] !! 3
  4
  ```

- Check whether a list is empty

  ```
  Prelude> null []
  True
  Prelude> null [1,2,3]
  False
  ```

Lists: reverse, !!, null, take

- Reverse a list

  ```
  Prelude> reverse [1,2,3,4,5]
  [5,4,3,2,1]
  ```

- Retrieve the *n*-th element of a list

  ```
  Prelude> [1,2,3,4,5] !! 3
  4
  ```

- Check whether a list is empty

  ```
  Prelude> null []
  True
  Prelude> null [1,2,3]
  False
  ```

- Retrieve the first *n* elements of a list

  ```
  Prelude> take 3 [1,2,3,4,5,6]
  [1,2,3]
  ```

Deconstructing lists

```
Prelude> let (x:xs) = [1,2,3,4]
Prelude> x
1
Prelude> xs
[2,3,4]
```

Deconstructing lists

```
Prelude> let (x:xs) = [1,2,3,4]
Prelude> x
1
Prelude> xs
[2,3,4]
```

Lecture.hs

```haskell
my_length :: [a] -> Integer
my_length []     = 0
my_length (x:xs) = 1 + my_length xs
```

Our own reverse function

Could we write our own `reverse` function?

Our own reverse function

Could we write our own `reverse` function? Robin's:

Lecture.hs

```haskell
my_reverse :: [a] -> [a]
my_reverse s = my_reverse' s []

my_reverse' :: [a] -> [a] -> [a]
my_reverse' []     s = s
my_reverse' (x:xs) s = my_reverse' xs (x:s)
```

Tuples

Tuples can hold multiple values of different types, however they have a finite length.

- Tuple with 2 Integers:

```
Prelude> (1,2)
(1,2)
```

- Tuple with an Integer and a Character:

```
Prelude> (1,'a')
(1,'a')
```

Tuples (2)

- List of two tuples:

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

Tuples (2)

- List of two tuples:

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

- List of two different tuples: [(1,2),('a',4)]

Tuples (2)

- List of two tuples:

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

- List of two different tuples: [(1,2),(’a’,4)]

Tuples (2)

- List of two tuples:

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

- List of two different tuples: [(1,2),('a',4)]

  ```
  Prelude> [(1,2),('a',4)]

  <interactive>:7:3:
      No instance for (Num Char) arising from the literal `1'
      Possible fix: add an instance declaration for (Num Char)
      In the expression: 1
      In the expression: (1, 2)
      In the expression: [(1, 2), ('a', 4)]
  ```

Tuples: fst, snd

- First element of a tuple with 2 elementen:

```
Prelude> fst (1,2)
1
```

Tuples: fst, snd

- First element of a tuple with 2 elementen:

```
Prelude> fst (1,2)
1
```

- Second element of a tuple with 2 elementen:

```
Prelude> snd (3,'d')
'd'
```

Tuples: fst, snd

- First element of a tuple with 2 elementen:

  ```
  Prelude> fst (1,2)
  1
  ```

- Second element of a tuple with 2 elementen:

  ```
  Prelude> snd (3,'d')
  'd'
  ```

- First element of a tuple with 3 elementen: fst (1,2,3)

Tuples: fst, snd

- First element of a tuple with 2 elementen:

```
Prelude> fst (1,2)
1
```

- Second element of a tuple with 2 elementen:

```
Prelude> snd (3,'d')
'd'
```

- First element of a tuple with 3 elementen: fst (1,2,3)

Tuples: fst, snd

- First element of a tuple with 2 elementen:

```
Prelude> fst (1,2)
1
```

- Second element of a tuple with 2 elementen:

```
Prelude> snd (3,'d')
'd'
```

- First element of a tuple with 3 elementen: fst (1,2,3)

```
Prelude> fst (1,2,3)

<interactive>:8:5:
    Couldn't match expected type `(a0, b0)'
                with actual type `(t0, t1, t2)'
    In the first argument of `fst', namely `(1, 2, 3)'
    In the expression: fst (1, 2, 3)
    In an equation for `it': it = fst (1, 2, 3)
```

Tuples: thrd function

Can we write a function that returns the third element of a tuple with 3 elements?

Tuples: thrd function

Can we write a function that returns the third element of a tuple with 3 elements?

Lecture.hs

```haskell
thrd :: (a,b,c) -> c
thrd (x,y,z) = z
```

```
*Lecture> thrd (1,2,3)
3
*Lecture> thrd ('a','b','c')
'c'
```

Content

Introduction

Lists and tuples

Functions

Ranges and list comprehensions

Anonymous functions, filter, map, foldr, foldl, zip, zipWith

Types

Tips 'n Tricks

Pattern matching

Lecture.hs

```haskell
lucky :: Integer -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

```
*Lecture> lucky 7
"LUCKY NUMBER SEVEN!”
*Lecture> lucky 4
"Sorry, you're out of luck, pal!”
```

Ignoring arguments

We can ignore arguments by using _ :

Lecture.hs

```haskell
lucky' :: Integer -> String
lucky' 7 = "LUCKY NUMBER SEVEN!"
lucky' _ = "Sorry, you're out of luck, pal!"
```

```
*Lecture> lucky'  7
"LUCKY NUMBER SEVEN!"
*Lecture> lucky' 4
"Sorry, you're out of luck, pal!"
```

If-then-else

Lecture.hs

```haskell
describeLetter :: Char -> String
describeLetter c =
    if c >= 'a' && c <= 'z'
        then "Lower case"
        else if c >= 'A' && c <= 'Z'
            then "Upper case"
            else "No ASCII letter"
```

```
*Lecture> describeLetter '1'
"No ASCII letter"
*Lecture> describeLetter 'a'
"Lower case"
*Lecture> describeLetter 'A'
"Upper case"
```

Guards

We can use “guards” to avoid if-spaghetti:

Lecture.hs

```haskell
describeLetter' :: Char -> String
describeLetter' c | c >= 'a' && c <= 'z' = "Lower case"
                  | c >= 'A' && c <= 'Z' = "Upper case"
                  | otherwise            = "No ASCII letter"
```

```
*Lecture> describeLetter' '1'
"No ASCII letter"
*Lecture> describeLetter' 'a'
"Lower case"
*Lecture> describeLetter' 'A'
"Upper case"
```

Where and let clauses

We can use `let` and `where` to improve code readability.

Lecture.hs

```haskell
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
                              where (f:_) = firstname
                                    (l:_) = lastname


initials' :: String -> String -> String
initials' firstname lastname = let (f:_) = firstname
                                   (l:_) = lastname
                               in  [f] ++ ". " ++ [l] ++ "."
```

```
*Lecture> initials "John" "Doe"
"J. D."
*Lecture> initials' "John" "Doe"
"J. D."
```

Content

Introduction

Lists and tuples

Functions

Ranges and list comprehensions

Anonymous functions, filter, map, foldr, foldl, zip, zipWith

Types

Tips ’n Tricks

Ranges

Problem: we want a list with all the integers between 1 and 15.

```
Prelude> [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Ranges

Problem: we want a list with all the integers between 1 and 15.

```
Prelude> [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Solution: ranges!

- `Prelude> [1..15]`
  `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]`

Ranges

Problem: we want a list with all the integers between 1 and 15.

```
Prelude> [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Solution: ranges!

- `Prelude> [1..15]`
  `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]`
- `Prelude> [2,4..20]`
  `[2,4,6,8,10,12,16,18,20]`

Ranges

Problem: we want a list with all the integers between 1 and 15.

```
Prelude> [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Solution: ranges!

-
  ```
  Prelude> [1..15]
  [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
  ```

-
  ```
  Prelude> [2,4..20]
  [2,4,6,8,10,12,16,18,20]
  ```

-
  ```
  Prelude> [3,6..20]
  [3,6,9,12,15,18]
  ```

Ranges

Problem: we want a list with all the integers between 1 and 15.

```
Prelude> [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Solution: ranges!

- `Prelude> [1..15]`
  `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]`
- `Prelude> [2,4..20]`
  `[2,4,6,8,10,12,16,18,20]`
- `Prelude> [3,6..20]`
  `[3,6,9,12,15,18]`
- `Prelude> [0.1, 0.3 .. 1]`
  `[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]`

Infinite lists

Because Haskell is lazy, we can construct infinite lists. Haskell will not fully evaluate the list unless you ask for it.

-
  ```
  Prelude> [1..]
  [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,..
  ```

Infinite lists

Because Haskell is lazy, we can construct infinite lists. Haskell will not fully evaluate the list unless you ask for it.

-
  ```
  Prelude> [1..]
  [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,..
  ```

-
  ```
  Prelude> take 5 [1..]
  [1,2,3,4,5]
  ```

Infinite lists

Because Haskell is lazy, we can construct infinite lists. Haskell will not fully evaluate the list unless you ask for it.

-
  ```
  Prelude> [1..]
  [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,..
  ```

-
  ```
  Prelude> take 5 [1..]
  [1,2,3,4,5]
  ```

-
  ```
  Prelude> take 10 $ cycle [1,2,3]
  [1,2,3,1,2,3,1,2,3,1]
  ```

Infinite lists

Because Haskell is lazy, we can construct infinite lists. Haskell will not fully evaluate the list unless you ask for it.

-
  ```
  Prelude> [1..]
  [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,..
  ```

-
  ```
  Prelude> take 5 [1..]
  [1,2,3,4,5]
  ```

-
  ```
  Prelude> take 10 $ cycle [1,2,3]
  [1,2,3,1,2,3,1,2,3,1]
  ```

-
  ```
  Prelude> take 10 $ repeat 1
  [1,1,1,1,1,1,1,1,1,1]
  ```

List comprehensions

$$S = \{x \cdot x | x \in \mathbb{N}, x \leq 10, x \mod 2 = 0\}$$

List comprehensions

$S = \{x \cdot x | x \in \mathbb{N}, x \leq 10, x \mod 2 = 0\}$
In Haskell:

```
Prelude> [x * x | x <- [1..10], even x]
[4,16,36,64,100]
```

A Pythagorean triple consists of three positive integers a, b and c, such that $a^2 + b^2 = c^2$:

List comprehensions

$S = \{x \cdot x | x \in \mathbb{N}, x \leq 10, x \mod 2 = 0\}$

In Haskell:

```
Prelude> [x * x | x <- [1..10], even x]
[4,16,36,64,100]
```

A Pythagorean triple consists of three positive integers a, b and c, such that $a^2 + b^2 = c^2$:

```
Prelude> [(a,b,c) | a <- [1..20], b <- [1..20], c <- [1..20]
                   ,a*a + b*b == c*c]
[(3,4,5),(4,3,5),(5,12,13),(6,8,10),(8,6,10),(8,15,17),
(9,12,15),(12,5,13),(12,9,15),(12,16,20),(15,8,17),(16,12,20)]
```

List comprehensions

$S = \{x \cdot x | x \in \mathbb{N}, x \leq 10, x \mod 2 = 0\}$
In Haskell:

```
Prelude> [x * x | x <- [1..10], even x]
[4,16,36,64,100]
```

A Pythagorean triple consists of three positive integers a, b and c, such that $a^2 + b^2 = c^2$:

```
Prelude> [(a,b,c) | a <- [1..20], b <- [1..20], c <- [1..20]
                   ,a*a + b*b == c*c]
[(3,4,5),(4,3,5),(5,12,13),(6,8,10),(8,6,10),(8,15,17),
(9,12,15),(12,5,13),(12,9,15),(12,16,20),(15,8,17),(16,12,20)]
```

Remove duplicate triplets:

```
Prelude> [(a,b,c) | a <- [1..20], b <- [1..20], c <- [1..20]
                     ,a*a + b*b == c*c, a < b]
[(3,4,5),(5,12,13),(6,8,10),(8,15,17),(9,12,15),(12,16,20)]
```

Sieve of Eratosthenes

Wikipedia

The sieve of Eratosthenes, one of a number of prime number sieves, is a simple, ancient algorithm for finding all prime numbers up to any given limit. It does so by iteratively marking as composite (i.e. not prime) the multiples of each prime, starting with the multiples of 2.

Sieve of Eratosthenes

Wikipedia

The sieve of Eratosthenes, one of a number of prime number sieves, is a simple, ancient algorithm for finding all prime numbers up to any given limit. It does so by iteratively marking as composite (i.e. not prime) the multiples of each prime, starting with the multiples of 2.

Lecture.hs

```haskell
primes :: [Integer]
primes = sieve [2..]
 where sieve (p:xs) = p : sieve [x | x<-xs, x `mod` p /= 0]
```

```
*Lecture> take 15 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

Content

Introduction

Lists and tuples

Functions

Ranges and list comprehensions

Anonymous functions, filter, map, foldr, foldl, zip, zipWith

Types

Tips 'n Tricks

Anonymous functions

Anonymous functions

An anonymous function is a function without a name. It is a lambda abstraction and it might look like this: `\x -> x + 1`

Anonymous functions

Anonymous functions

An anonymous function is a function without a name. It is a lambda abstraction and it might look like this: `\x -> x + 1`

```
Prelude> (\x -> x + 1) 1
2
Prelude> (\a b -> a + b) 3 5
8
Prelude> (\_ b -> b) 17 42
42
```

Filter

Remove all elements from a list for which a given function returns False.

Filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _pred []    = []
filter pred (x:xs)
    | pred x         = x : filter pred xs
    | otherwise      = filter pred xs
```

Filter

Remove all elements from a list for which a given function returns False.

Filter

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _pred []     = []
filter pred (x:xs)
    | pred x          = x : filter pred xs
    | otherwise       = filter pred xs
```

```
Prelude> filter odd [1,2,3,4,5]
[1,3,5]
Prelude> filter (\x -> x > 3) [1,2,3,4,5]
[4,5]
```

Map

Apply a given function to each element of a list.

Definition of map

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Map

Apply a given function to each element of a list.

Definition of map

```haskell
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

```
Prelude> map odd [1,2,3,4,5]
[True,False,True,False,True]
Prelude> map (\x -> x + 3) [1,2,3,4,5]
[4,5,6,7,8]
```

(Ab)using lazyness

We can put arguments in a function in advance:

Lecture.hs

```haskell
add :: Integer -> Integer -> Integer
add a b = a + b
```

(Ab)using lazyness

We can put arguments in a function in advance:

Lecture.hs

```haskell
add :: Integer -> Integer -> Integer
add a b = a + b
```

```
*Lecture> map (add 3) [1,2,3,4]
[4,5,6,7]
```

(Ab)using lazyness

We can put arguments in a function in advance:

Lecture.hs

```haskell
add :: Integer -> Integer -> Integer
add a b = a + b
```

```
*Lecture> map (add 3) [1,2,3,4]
[4,5,6,7]
```

Even when we use anonymous functions (do not forget the parentheses!)

```
*Lecture> map ((\a b -> a + b) 3) [1,2,3,4]
[4,5,6,7]
```

Currying

This *lazy* behaviour of functions in Haskell is due to the fact that functions officially only have one parameter in Haskell. Functions with more parameters are **curried**.

Further reading: `http://learnyouahaskell.com/higher-order-functions`

Foldr

'foldr', applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left.

Definition of foldr

```
foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr _ z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)
```

Foldr

'foldr', applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left.

Definition of foldr

```
foldr              :: (a -> b -> b) -> b -> [a] -> b
foldr _ z []       =  z
foldr f z (x:xs)   =  f x (foldr f z xs)
```

```
Prelude> foldr (+) 0 [1,2,3,4,5]
15
Prelude> foldr (\x y -> x + 2 * y) 4 [1,2,3]
49
```

Foldl

'foldl', applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right:

Definition of foldr

```
foldl        :: (b -> a -> b) -> b -> [a] -> b
foldl f z0 xs0 = lgo z0 xs0
             where
                lgo z []     =  z
                lgo z (x:xs) = lgo (f z x) xs
```

Foldl

'foldl', applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right:

Definition of foldr

```
foldl          :: (b -> a -> b) -> b -> [a] -> b
foldl f z0 xs0 = lgo z0 xs0
             where
                lgo z []     =  z
                lgo z (x:xs) = lgo (f z x) xs
```

```
Prelude> foldl (\x y -> 2*x + y) 4 [1,2,3]
43
```

Zip

'zip' takes two lists and returns a list of corresponding pairs (tuples of 2 elements). If one input list is short, excess elements of the longer list are discarded.

Definition of zip

```
zip :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs) = (a,b) : zip as bs
zip _      _      = []
```

```
Prelude> zip [1,2,3] [4,5,6]
[(1,4),(2,5),(3,6)]
Prelude> zip [1,2,3,4,5] ['a','b','c','d','e']
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e')]
```

ZipWith

'zipWith' generalises 'zip' by zipping with the function given as the first argument, instead of a tupling function.

Definition of zipWith

```
zipWith :: (a->b->c) -> [a]->[b]->[c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _ _      _      = []
```

```
Prelude> zipWith (+) [1,2,3] [4,5,6]
[5,7,9]
Prelude> zipWith (*) [1,2,3] [4,5,6]
[4,10,18]
```

Function composition

Function composition is the act of pipelining the result of one function, to the input of another, creating an entirely new function.
Mathematically, this is most often represented by the $\circ$ operator, where $f \circ g$ (often read as f of g) is the composition of f with g.

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Function composition

Function composition is the act of pipelining the result of one function, to the input of another, creating an entirely new function.

Mathematically, this is most often represented by the $\circ$ operator, where $f \circ g$ (often read as f of g) is the composition of f with g.

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

```
Prelude> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
Prelude> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Function composition

Function composition is the act of pipelining the result of one function, to the input of another, creating an entirely new function.
Mathematically, this is most often represented by the $\circ$ operator, where $f \circ g$ (often read as f of g) is the composition of f with g.

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

```
Prelude> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
Prelude> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]

Prelude> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
Prelude> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

Content

Introduction

Lists and tuples

Functions

Ranges and list comprehensions

Anonymous functions, filter, map, foldr, foldl, zip, zipWith

Types

Tips 'n Tricks

Types

As we have already noticed, variables have a type:

```
Prelude> :t True
True :: Bool
Prelude> :t 'a'
'a' :: Char
Prelude> :t "hooi"
"hooi" :: [Char]
```

In Haskell we have:

- Int: integer numbers, bounded
- Integer: integer numbers, unbounded (thus slower!)
- Float: single precision floats
- Double: double precision floats
- Bool: True of False
- Char: a single character and `[Char]` is a string.

Type signatures

When defining a function, we can also give its type-signature:

Lecture.hs

```haskell
fac :: Integer -> Integer
fac 0 = 1
fac n = n * foo (n-1)
```

or using an unknown type:

Lecture.hs

```haskell
third :: (a,b,c) -> c
third (_,_,z)  = z
```

Type aliases

We can define aliases using the **type** keyword:

Lecture.hs

```haskell
type Feedback = (Int, Int)
```

Custom datatypes

We can define our own types using the **data** keyword:

Lecture.hs

```
data Color = Red | Yellow | Blue | Green | Orange | Purple
             deriving (Eq, Show, Bounded, Enum)
```

```
Prelude> [minBound..maxBound] :: [Color]
[Red,Yellow,Blue,Green,Orange,Purple]
Prelude> let foo = Red
Prelude> foo
Red
Prelude> :t foo
foo :: Color
```

Typeclasses

Types can be part of typeclasses (members):

```
Prelude> :t elem
elem :: Eq a => a -> [a] -> Bool
```

- Eq: type supports == en /=
- Ord: type is ordered
- Show: type can be represented as a string
- Read: string can be represented as the type
- Enum: type is sequentially ordered
- Bounded: type has lower- and upper-bounds
- Num: type acts as a number
- Integral: type is an integer number (Int/Integer)
- Floating: type is a floating point number (Float/Double)

Content

Introduction

Lists and tuples

Functions

Ranges and list comprehensions

Anonymous functions, filter, map, foldr, foldl, zip, zipWith

Types

Tips 'n Tricks

Tips when using GHCi

Show the type of a function or variable:

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

Show type for all subsequent commands:

```
Prelude> :set +t
Prelude> 1
1
it :: Integer
Prelude> filter even [1..20]
[2,4,6,8,10,12,14,16,18,20]
it :: [Integer]
```

Disable:

```
Prelude> :unset +t
```

Remember, operators are also functions

Lecture.hs

```haskell
neg :: Integer -> Integer
neg a = - a
```

```
Prelude> neg 3
−3
Prelude> neg −3
<interactive >:9:1:
No instance for (Num a0) arising from a use of `neg'
The type variable `a0' is ambiguous
Possible fix: add a type signature that fixes these type variable(s)
```

Remember, operators are also functions

Lecture.hs

```
neg :: Integer -> Integer
neg a = - a
```

```
Prelude> neg 3
−3
Prelude> neg −3
<interactive>:9:1:
No instance for (Num a0) arising from a use of `neg'
The type variable `a0' is ambiguous
Possible fix: add a type signature that fixes these type variable(s)
```

Solution: parentheses or $:

```
Prelude>
Prelude> neg (−3)
3
Prelude> neg $ −3
3
```