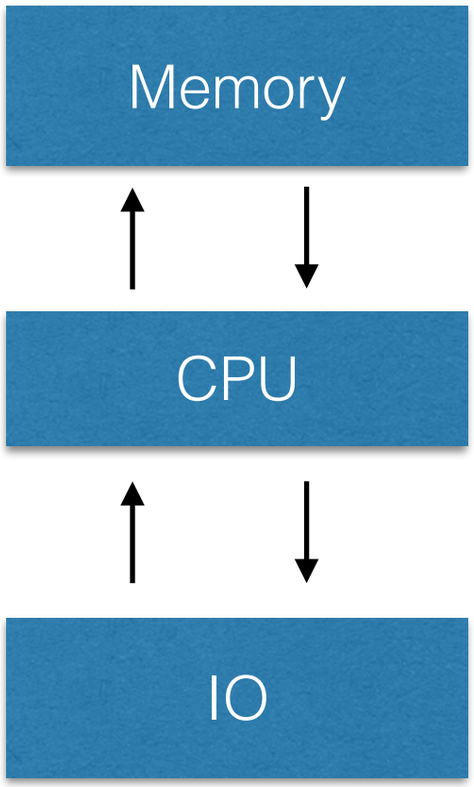
Prelude

Functional Programming 2018-2019
Christophe Scholliers

Slides based on the book learn you a Haskell

Von Neumann



Programming Languages

~ Von Neumann

SWI Prolog

! Von Neumann

Von Neumann

```
int x = 2;
int  executeExample() {
        System.out.println("Result = "+ multiply(4,5));
        System.out.println("Result = "+ square(4));
        return multiply(2,square(x))+multiply(2,x)+3;
}
```

```
Result = 20
Result = 16

executeExample = ?
```

Von Neumann

```
int x = 2;
int  executeExample() {
        System.out.println("Result = "+ multiply(4,5));
        System.out.println("Result = "+ squared(4));
        return multiply(2,squared(x))+multiply(2,x)+3;
}
```

```
Result = 20
Result = 16

executeExample = 1539
```

WHY ?

Von Neumann

```
int x = 2;

int  executeExample() {
        System.out.println("Result = "+ multiply(4,5));
        System.out.println("Result = "+ squared(4));
        return multiply(2,squared(x))+multiply(2,x)+3;
}

int multiply(int p,int q) {
                x = p *q;
                return x;
}

int squared(int p) {
    x = p * p;
    return x;
}
```

Variables in Math

$$3x^2 + 8x + 42$$

*"Curiously enough mathematicians tend to call these things '**variables**' although their most important property is precisely that **they do not vary**."*

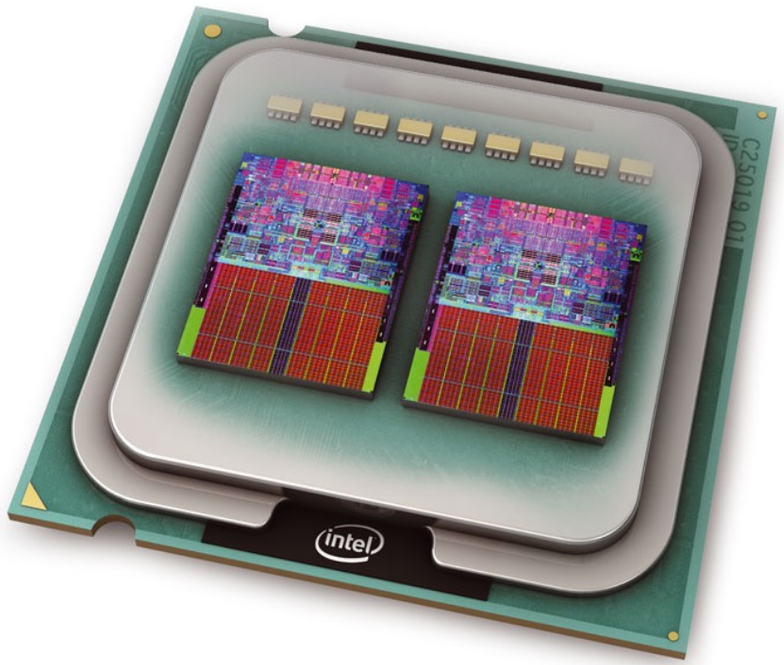
Fundamental Concepts in Programming Languages
CHRISTOPHER STRACHEY
Reader in Computation at Oxford University, Programming Research Group, 45 Banbury Road, Oxford, UK

Referential Transparency

$$f(x) = f(x)$$

Fundamental property of functional programming languages

Assignment (=, :=)



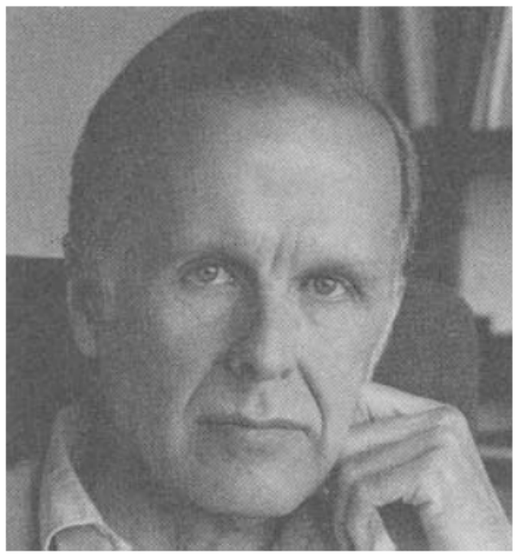
Race conditions

Variables that can vary

Enemy of referential transparency

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

In this course

λ

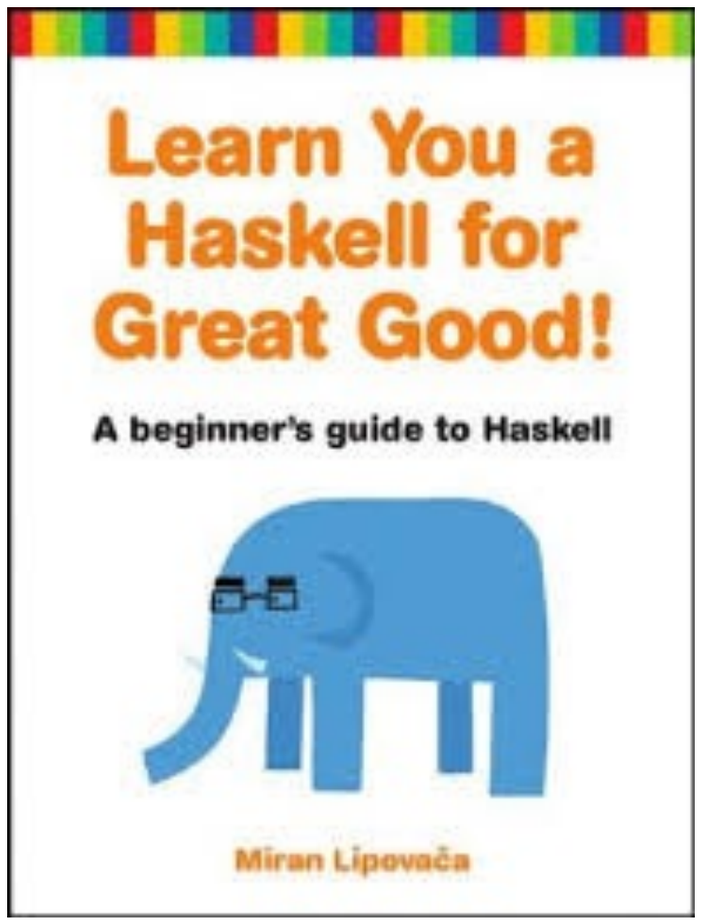
Theory

```haskell
module Main where

-- Let's import monads
import Control.Monad ((>>=))

-- Main program
main :: IO ()
main = getLine >>= doThings >>= putStrLn

doThings :: String -> IO String
doThings = return . reverse
```

Haskell

Project

Book



http://learnyouahaskell.com/

Grading

Examen (50%)

Project (50%)

Haskell

Philip Wadler

Simon Peyton Jones

Erik Meijer

Starting out

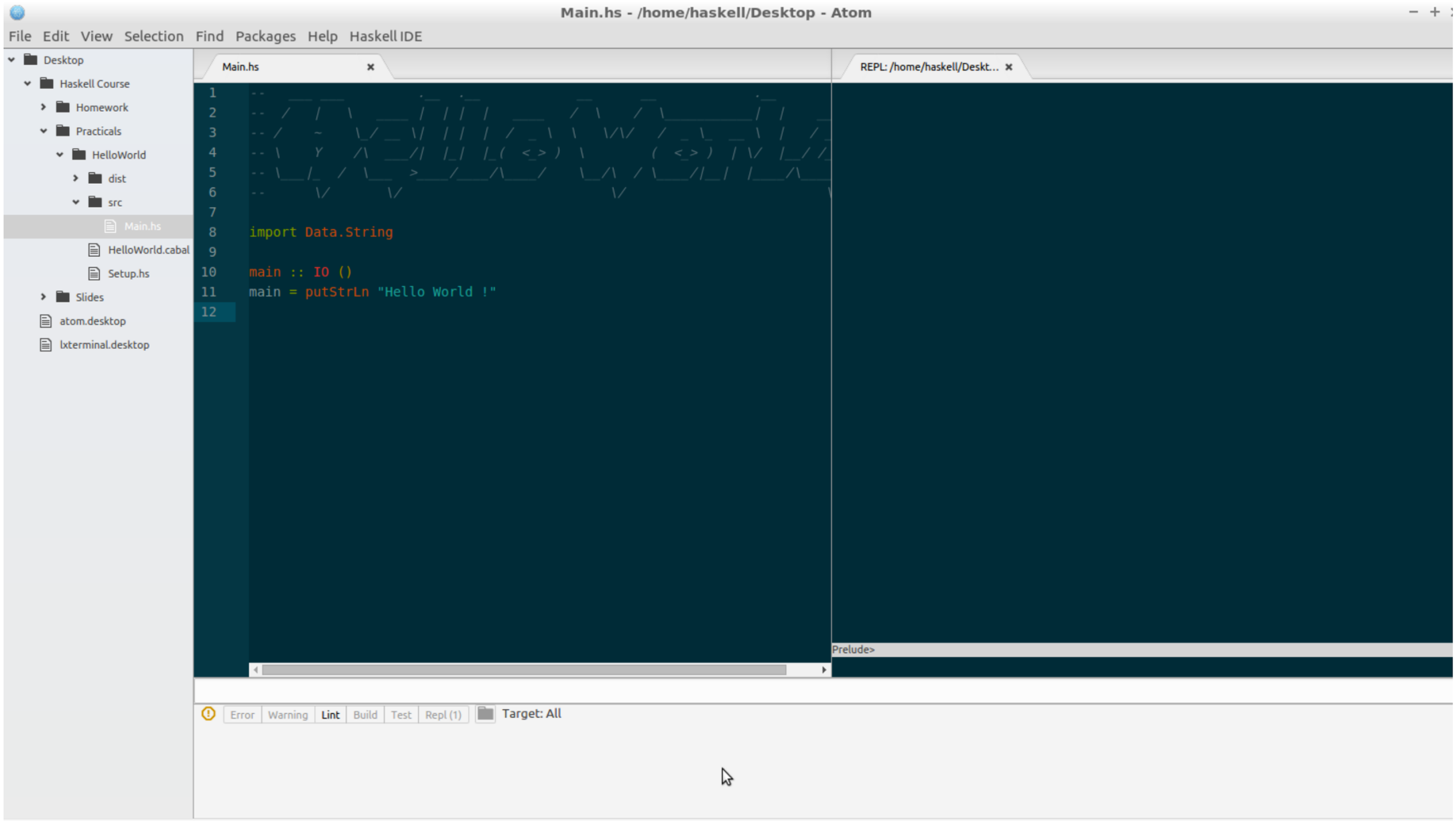
https://www.haskell.org/platform/ 8.4.3

https://atom.io/

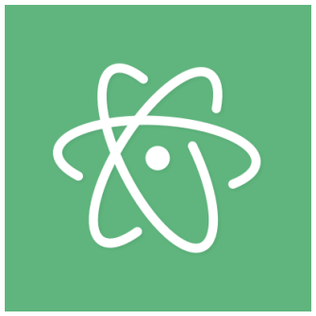
https://atom.io/packages/ide-haskell

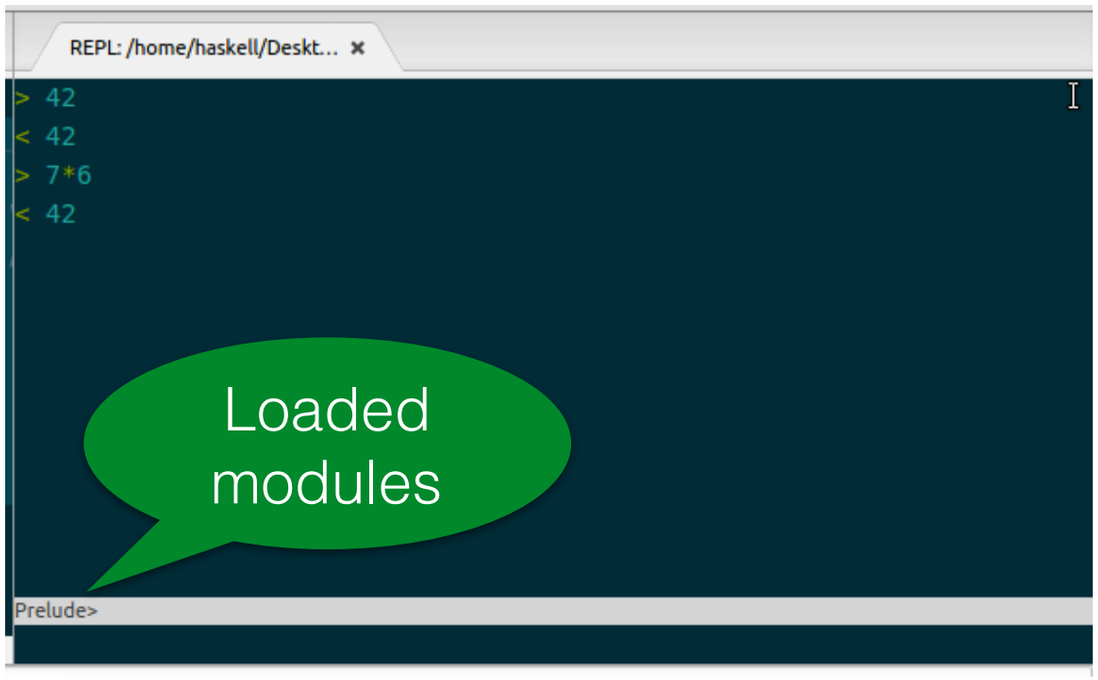
Easy for windows
and linux

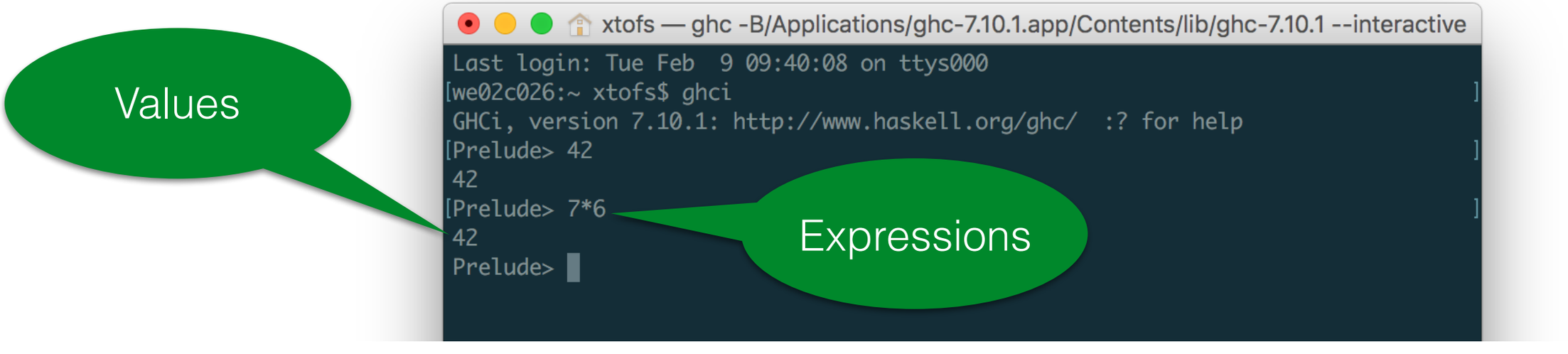
Atom


Main.hs - /home/haskell/Desktop - Atom
File Edit View Selection Find Packages Help Haskell IDE
Desktop
Haskell Course
Homework
Practicals
HelloWorld
dist
src
Main.hs
HelloWorld.cabal
Setup.hs
Slides
atom.desktop
lxterminal.desktop
Main.hs
REPL: /home/haskell/Deskt...
import Data.String
main :: IO ()
main = putStrLn "Hello World !"
Prelude>
Error Warning Lint Build Test Repl (1)
Target: All

Read-Eval-Print-Loop







Arithmetic Expressions

```
Prelude> 2 + 45
47
Prelude> 49*23
1127
Prelude> 1938-1384
554
Prelude> 5/23
0.21739130434782608
```

Precedence

```
Prelude> 12 * 39 - 39
429
Prelude> (12*39) - 39
429
Prelude> 12*(39 - 39)
0
```

Negative numbers

```
Prelude> 5 * -3

<interactive>:2:1:
    Precedence parsing error
        cannot mix ‘*’ [infixl 7] and prefix `-' [infixl 6] in the same infix expression
```

```
Prelude> 5 * (-3)
-15
```

Boolean Values

```
Prelude> True && False
False
Prelude> not False
True
Prelude> not (True && False)
True
Prelude> False || True
True
```

Type error !

```
Prelude> 5 == 23
False
Prelude> 5 == "UGent"
<interactive>:11:1:
    No instance for (Num [Char]) arising from the literal ‘5’
    In the first argument of ‘(==)’, namely ‘5’
    In the expression: 5 == "UGent"
    In an equation for ‘it’: it = 5 == "UGent"
```

Function Calls

```
Prelude> succ 41
42
Prelude> min 9 123
9
Prelude> min 3.45 3.355
3.355
Prelude> succ 41 + max 3 5  + 1
48
```

Precedence

Infix notation

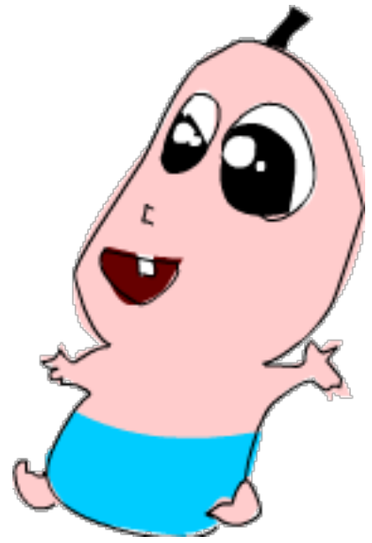
only when the function takes two arguments

```
Prelude> 3 `max` 23
23
```

Currying

```
Prelude> let max10 = max 10
Prelude> max10 4
10
Prelude> max10 94
94
```

Defining functions



```
1 doubleMe x    = x + x
2 doubleUs x y = x*2 + y*2
```

loading files

```
Prelude> :l chapter1.hs
[1 of 1] Compiling Main             ( chapter1.hs, interpreted )
Ok, modules loaded: Main.
*Main> doubleMe 235
470
*Main> doubleUs 20 23
86
```

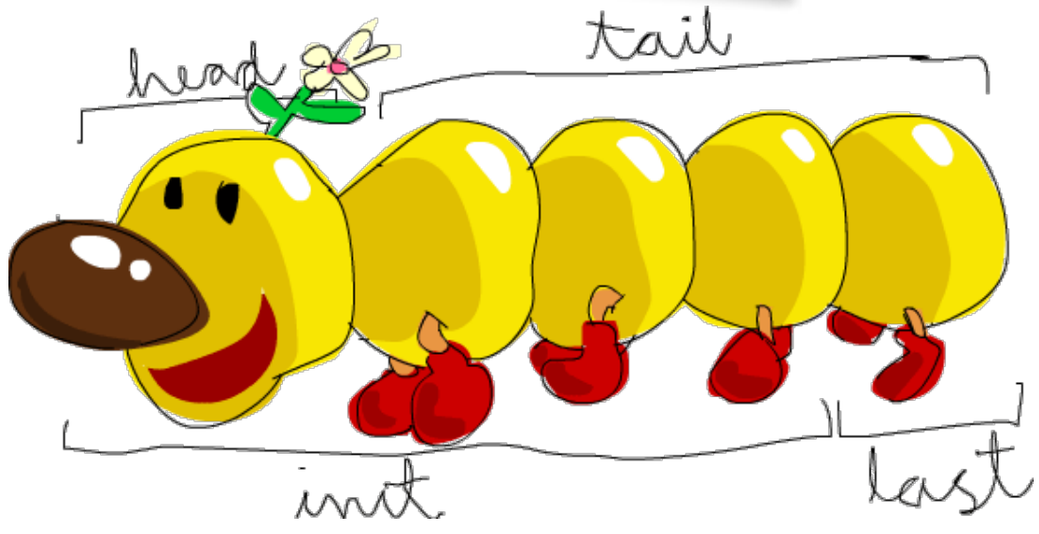
If-Expressie

```
1 doubleSmallNumber x = if x > 100
2                         then x
3                         else x*2
```

Lists

Lists

```
Prelude> let b = [1,2,3,4,2,4,2,12,4,2,3,32]
Prelude> head b
1
Prelude> tail b
[2,3,4,2,4,2,12,4,2,3,32]
Prelude> init b
[1,2,3,4,2,4,2,12,4,2,3]
Prelude> last b
32
```



More Lists

```
Prelude> [13,4,1,4,21] ++ [1,31,4,1,1,44,1,2]
[13,4,1,4,21,1,31,4,1,1,44,1,2]
Prelude> 'A' : " Student"
"A Student”
Prelude> "UGent" !! 2
‘e'
Prelude> [2,3,2] > [2,1,0]
True
Prelude> [3,2,1] > [2,1,0]
True
Prelude> [3,2,1] > [2,1]
True
Prelude> [3,2,1] > [2]
True
Prelude> [3,2,1] == [2]
False
Prelude> [3,2,1] == [3,2,1]
True
Prelude> head []
*** Exception: Prelude.head: empty list
```

concatenation

indexing

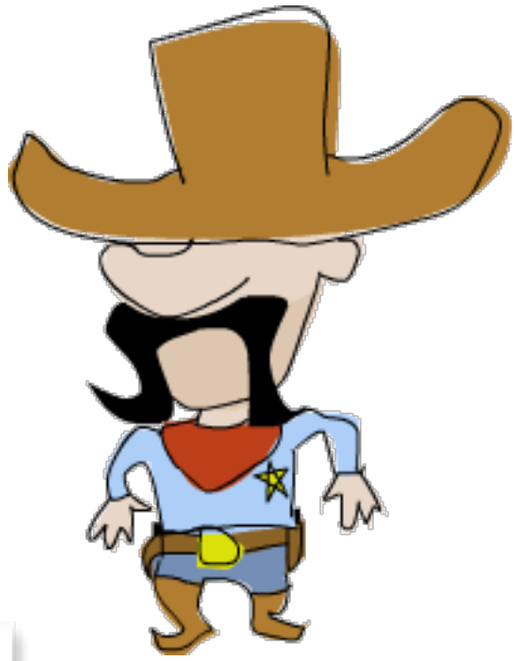
comparison

More Lists

```
Prelude> take 3 [5,4,3,2,1]
[5,4,3]
Prelude> take 1 [3,9,3]
[3]
Prelude> take 5 [1,2]
[1,2]
Prelude> take 0 [6,6,6]
[]
Prelude> drop 3 [8,4,2,1,5,6]
[1,5,6]
Prelude> drop 0 [1,2,3,4]
[1,2,3,4]
Prelude> drop 100 [1,2,3,4]
[]
```

```
Prelude> minimum [8,4,2,1,5,6]
1
Prelude> maximum [1,9,2,3,4]
9
Prelude> sum [5,2,1,6,3,2,5,7]
31
Prelude> product [6,2,1,2]
24
Prelude> product [1,2,5,6,7,9,2,0]
0
Prelude> 4 `elem` [3,4,5,6]
True
Prelude> 10 `elem` [3,4,5,6]
False
```

Ranges



```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
Prelude> ['a' .. 'z']
“abcdefghijklmnopqrstuvwxyz"
Prelude> take 20 $ cycle "1234"
“12341234123412341234"
[1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,1
8,19,20,21,22,23,24,25,26,27,28,29,30,31,32,
33,34,35,36,37,38,39,40,41,42,43,44,45,46,47
,48,49,50,51,52,53,54,55,56,57,58,59,60,61,6
2,63,64,65,66,67,68,69,70,71,72,73,74,75,76,
77,78,79,80,81,82,83,84,85,86,87,88,89,90,91
,92,93,94,95,96,97,98,99,100,101,102,103,104
,105,106,107,108,109,110,111,112,113,114,115
,116,117,118,119,120,121,122,123,124,125,126
```

List Comprehensions

```
Prelude> [x*3 | x <- [1..10] ]
[3,6,9,12,15,18,21,24,27,30]
```

Filters

```
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]

ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
```

Example

```
Prelude> let nouns = ["hobo","frog","pope"]
Prelude> let adjectives = ["lazy","grouchy","scheming"]
Prelude> [adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns]
["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy frog","grouchy
pope","scheming hobo","scheming frog","scheming pope"]
Prelude>
```

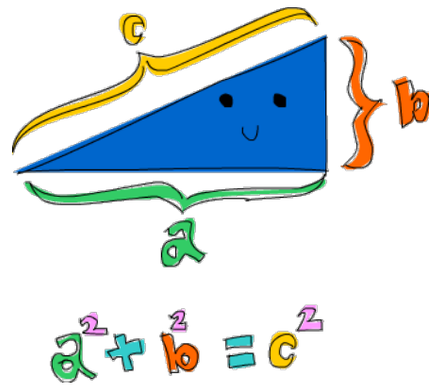
Sorting a List

```haskell
quicksort :: [Int] -> [Int]
quicksort [] = []

quicksort (pivot:rest) = quicksort [ x | x<-rest , x<=pivot  ] ++
                         [pivot] ++
                         quicksort [ x | x<-rest, x>pivot]
```

Tuples

```
Prelude> ("Foo","Bar")
("Foo","Bar")
Prelude> fst ("Foo","Bar")
"Foo"
Prelude> snd ("Foo","Bar")
"Bar"
```

Triangles




```
triangles = [ (a,b,c) |   c <- [1..10],
                          b <- [1..10],
                          a <- [1..10] ]


rightTriangles = [ (a,b,c) | c <- [1..10],
                             b <- [1..c],
                             a <- [1..b],
                             a^2 + b^2 == c^2]

rightTriangles' = [ (a,b,c) | c <- [1..10],
                              b <- [1..c],
                              a <- [1..b],
                              a^2 + b^2 == c^2,
                              a+b+c == 24]
```

Simple Types

```
*Main> :t 'a'
'a' :: Char
*Main> :t True
True :: Bool
*Main> :t 3 == 34
3 == 34 :: Bool
```

:: is read as "has type of"

Composite types 1

```
Prelude> :t ('a',23)
('a',23) :: Num t => (Char, t)
Prelude> :t [1,2,3]
[1,2,3] :: Num t => [t]
Prelude> :t "hallo"
"hallo" :: [Char]
```

Composite type 2

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]

addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Optional but it is good coding style to give all top level functions a type

Polymorphic Functions

"generics"

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


THE End