Welcome to Programming Fundamentals 3

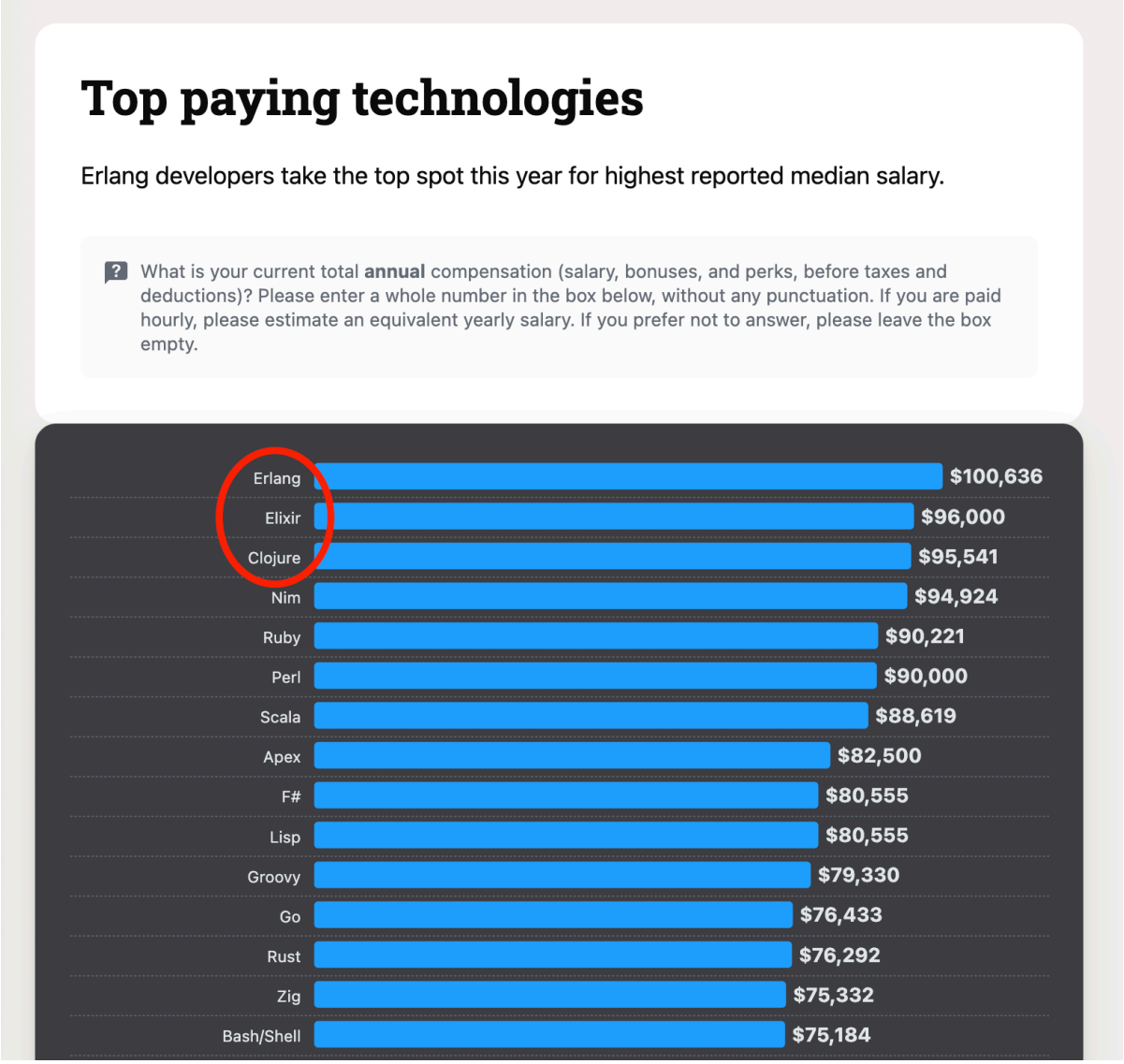
University of Luxembourg

2025/2026

Bachelor in Computer Sciences (BICS)
Semester 3

Lecture - Week #1

Dr. Afonso DELERUE ARRIAGA afonso.delerue@uni.lu

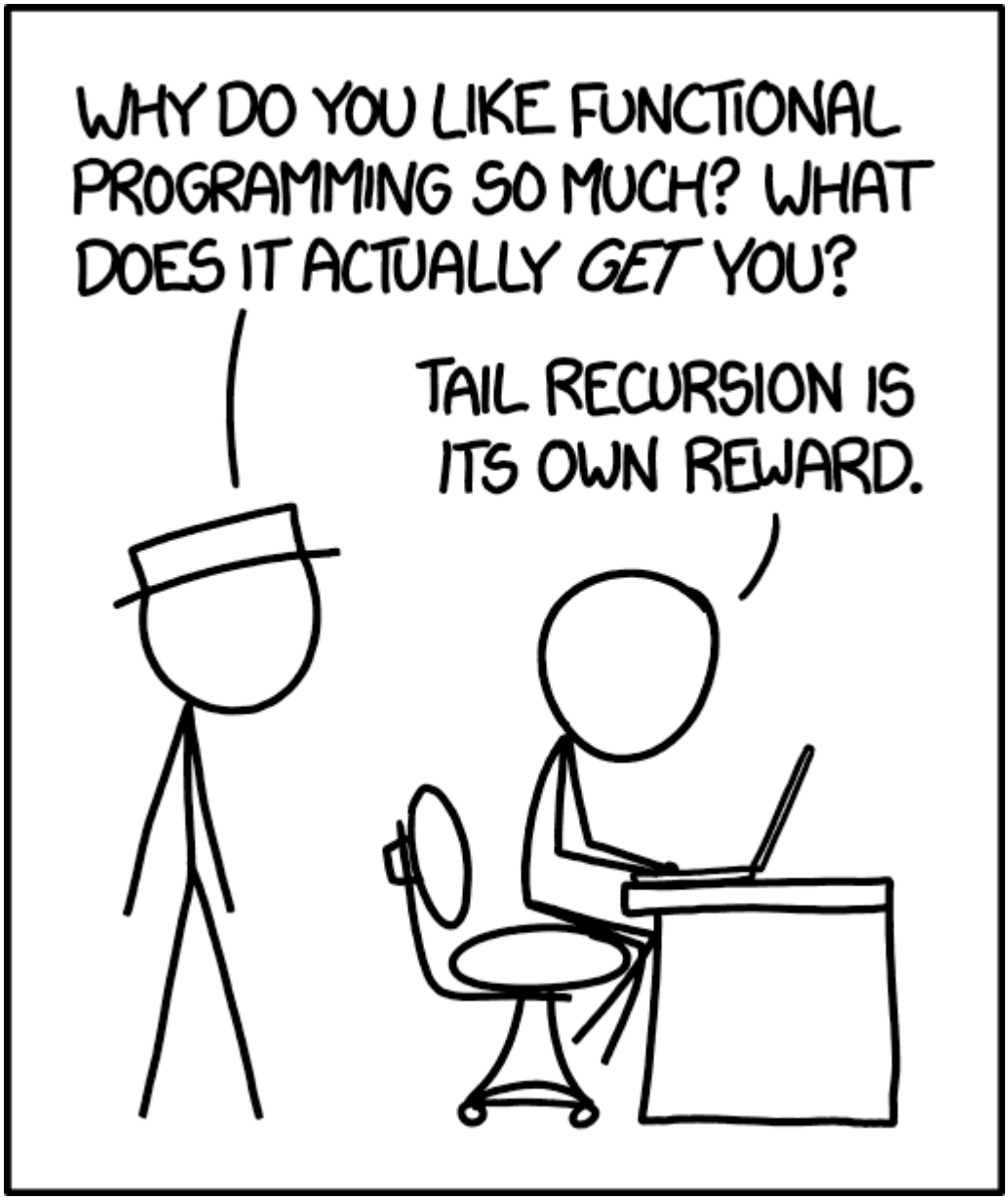


About Programming Fundamentals 3

- Different programming paradigms lead to different ways of solving problems and organizing code. Most programming languages have unique characteristics that

encourage a certain programming style, whether it be procedural, object-oriented, functional or logical.

- PF1: Python
- PF2: Java
- PF3: Haskell



Source: xkcd comic

ACM, IEEE, AAAI recommend Functional Programming

- The joint task force of the ACM, IEEE Computer Society, and AAAI recommend the inclusion of FP in CS curricula:
  - Lambda expressions and evaluation;
  - Function calls have no side effects, facilitating compositional reasoning;
  - Immutable variables;
  - Use of recursion vs loops.
- Learning a functional programming language improves reasoning skills.

Course overview

Lectures:

Wednesdays (or Mondays), 14h00 - 15h30
Dr. Afonso DELERUE ARRIAGA afonso.delerue@uni.lu

Practicals:

Mondays, 08h30 - 10h00
Dr. Miroslav KRATCHVIL miroslav.kratchvil@uni.lu

Student jobs:

(2025) Jiahao LIN, Matthew-Stephen WILEMAN

(2024) Antonia CUBA, Maeva CHEKAM
(2023) Jason BILLARD, Cosmin RADOI

Evaluation 📝

- 1 homework assignment with programming problems (30%)
- 1 mid-term written exam (30%)
- 1 final written exam (40%)

~~**A minimum grade of 8/20 in the final written exam** is required to pass the course. (2024/2025)~~

Students having failed the course may re-sit the final exam at a next exam session, which will then count for 100% of the final grade.

Moodle 📚

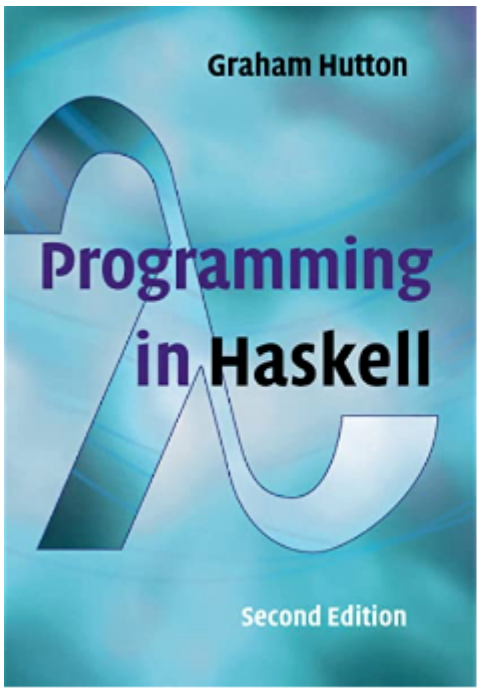
Lecture notes, exercises, complementary material, homework assignments and submission via Moodle.

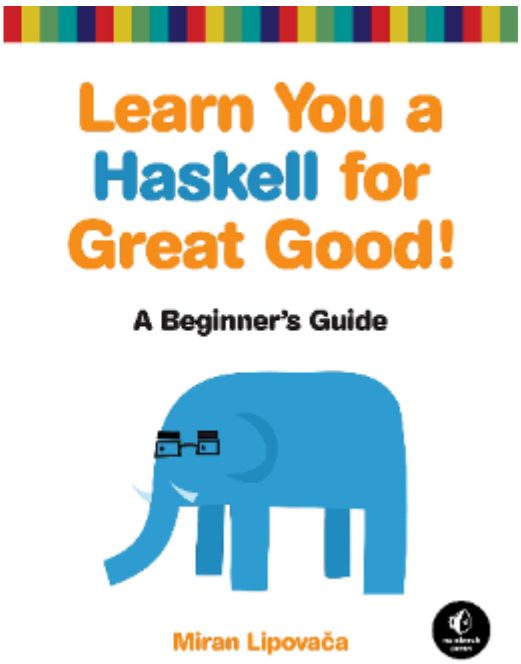
(Homeworks) Moodle does not allow the teacher to download .hs files. Please remember to compress your homeworks in .zip archive format before submitting your assignment. Always double-check your homework by downloading it from Moodle.

Bibliography

- Programming in Haskell, 2nd Edition, by Graham Hutton. September 2016, Cambridge University Press, ISBN-13 978-1316626221. Available via a-z.lu

- Learn You a Haskell for Great Good!, by Miran Lipovaca. April 2011, No Starch Press, ISBN-13 978-1593272838. Free online





Learning outcomes 🧠

- Learn how to program and reason about programs in a purely functional programming style.
- Solve problems in Haskell in a concise, yet expressive, manner.
- Think recursively, make use of higher-order functions, understand the benefits of type systems and learn how some progamming languages like Haskell avoid unnecessary computations via lazy evaluation.
- Know common data structures (like stacks, queues, trees, self-balancing trees, and graphs) and how these can be implemented in a functional language with data immutability.
- Know how to analyse the run-time complexity of operations on these data structures.
- Learn Haskell and mathematical reasoning in practice by implementing concrete arithmetic algorithms.
- Grasp advanced concepts of functional programming such as functors, applicative functors and monads.
- Learn how to approach and solve classical computer science problems in Haskell with essential techniques such as recursion, permutation generation, brute-force and binary search.

Some popular functional programming languages

- **Lisp**: One of the oldest and most influential functional programming languages, Lisp is still used today in fields such as artificial intelligence and data mining.
- **Haskell**: A purely functional programming language that is popular for its strong static type system and expressive type inference. This course! Programming Fundamentals 3.

- **Erlang**: A functional programming language designed for concurrent and distributed systems, with a focus on fault tolerance and scalability.
- **ML**: A family of functional programming languages that are known for their simple, elegant syntax and strong static type systems.
- **Scala**: A hybrid functional programming language that runs on the Java virtual machine and is popular for its interoperability with Java.
- **F#**: A functional programming language developed by Microsoft that is particularly well-suited for writing algorithms and data-driven applications.
- **OCaml**: A functional programming language that is known for its fast runtime and high-level abstractions. Previous editions of PF3.

Why Haskell? λ

- Haskell is a purely functional programming language. Fun fact: Haskell is named after the American logician **Haskell Curry**, who was a pioneer in the field of combinatory logic, which is a branch of mathematical logic that studies the combination of functions.
  - In imperative languages you get things done by giving the computer a sequence of tasks and then it executes them. While executing them, it can change state.
  - In purely functional programming you don't tell the computer what to do as such but rather you tell it what stuff is. There are no variables!
- There is no side-effects.
- Haskell is lazy.
- Haskell is statically typed.
- Haskell is elegant and concise.

Comparing with OCaml

- Similar in many ways, but...
  - OCaml is strict (performance might be more predictable), Haskell is lazy (avoid unnecessary computations, manage infinite lists, etc.)
  - Haskell is *pure*, and the impure world with side-effects is fenced off from pure functions.

Example: Python vs Haskell

Compute the factorial of `n`.

Python

```
def factorial(n: int) -> int:
    result = 1
    for i in range(1, n+1):
        result = result * i
    return result
```

Haskell

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Step by step in Python

factorial(5)

Step	n	i	i in range(1,n+1)?	result
1	5	1	True	1
2	5	2	True	2
3	5	3	True	6
4	5	4	True	24
5	5	5	True	120
6	5	6	False	120

Step by step in Haskell

```
factorial 5 = 5 * factorial 4
            = 5 * 4 * factorial 3
            = 5 * 4 * 3 * factorial 2
            = 5 * 4 * 3 * 2 * factorial 1
            = 5 * 4 * 3 * 2 * 1 * factorial 0
            = 5 * 4 * 3 * 2 * 1 * 1
            = 120
```

We could also write the factorial function as such in Python:

```
from functools import reduce

def factorial(n):
    if n == 0:
        return 1
    else:
        return reduce(lambda x, y: x * y, range(1, n + 1))
```

A functional programming style is not exclusive of functional programming languages, but languages that follow this paradigm *encourage* a functional programming style.

One-liner in Haskell

The same code could also be written in 1 line in Haskell:

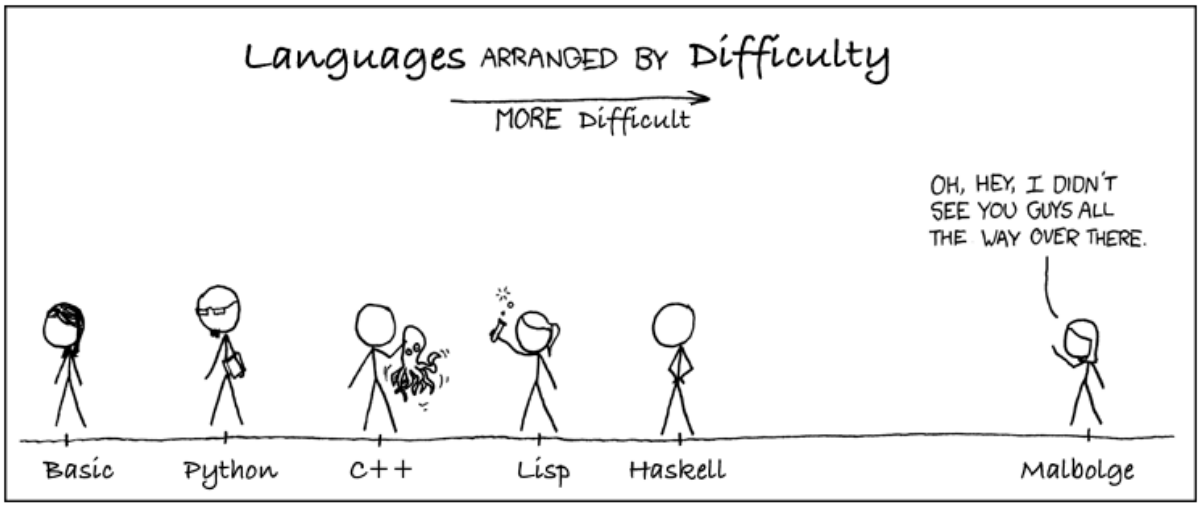
In [1]:
```
factorial n = product [1..n]

factorial 5
```

Pros and cons of functional programming languages

Pros ✅

- Programs are concise, yet expressive.
- Less code, less bugs, less debugging.
- Closer to mathematical specification.
- Helps better reasoning in any programming language.
- Modular thinking: Split problems in smaller sub-problems, and find abstract reusable solutions.
- Easier to do software verification with formal methods.
- Easier to do automatic testing -- see QuickCheck.
- Better for concurrency because there's no state and execution order does not affect results



Source: Parody of xkcd comic

Pros and cons of functional programming languages

Cons ❌

- Not as popular as imperative and object-oriented paradigms.
  - Steeper learning curve?
- Some algorithms are more efficient in imperative languages.
- Functional programming languages are distant from hardware and not so suitable for low-level programming implementations.

Some history of the functional programming paradigm

- **1930s**: Alonzo Church develops λ-calculus, a mathematical formalism for expressing computation using functions. Fun fact: Alonzo Church was the PhD advisor of Alan

Turing, who later developed another famous model of computation known as Turing Machine.

- **1950s**: John McCarthy developed Lisp, one of the first functional programming languages.
- **1970s**: Robin Milner and others developed Standard ML, a functional programming language with polymorphism and type inference that has been influential in the design of other functional languages.
- **1980s**: David Turner developed Miranda, a lazy functional programming language that was influential in the development of Haskell and other functional languages.
- **Late 1980s and early 1990s**: A team led by Simon Peyton Jones at the University of Glasgow developed Haskell, a purely functional, and lazy, programming language.

- **1998**: The Haskell 98 standard was released, which established Haskell as a stable and widely recognized functional programming language. Haskell 98 introduced a number of important features and improvements to the language, including a new syntax for defining datatypes and a module system for organizing code. It has since become the basis for all subsequent versions of Haskell.
- **2010**: The Haskell 2010 standard was published.

Haskell in the industry

- **Meta**: SPAM filter. Fighting spam with Haskell
- **Github**: Tool for parsing, analyzing, and comparing source code. Semantic
- **Cardano**: One of the biggest blockchains with smart contracts. Cardano
- **Galois**: Trustworthiness of critical systems. |galois|
- Other: A List of companies that use Haskell

Open-source projects

- The Glasgow Haskell Compiler (**GHC**), which is the most widely used Haskell compiler and is developed and maintained by a team at the University of Glasgow. Haskell repo
- The **Hasura** GraphQL Engine provides instant GraphQL REST API for SQL databases. GraphQL Engine repo
- **SimpleX** chat is a decentralized messaging and application platform without any personal user identifiers. SimpleX repo
- The **Xmonad** window manager, which is a lightweight, customizable window manager for the X Window System that is written in Haskell. Xmonad repo
- The **ShellCheck** is a tool that gives warnings and suggestions for bash/sh shell scripts. ShellCheck repo
- **Pandoc**, which is a document conversion tool that can convert between a wide variety of formats, including Markdown, LaTeX, and HTML. Pandoc is widely used for converting documents for the web and for creating ebooks. Pandoc repo
- **IHP** is a web framework, built on top of Haskell. IHP repo

Getting started

To get started, you need a Haskell compiler and a source-code editor with Haskell language support.

GHC is a state-of-the-art, open source, compiler and interactive environment for the functional language Haskell. The official way to install GHC for Linux, macOS and Windows is via GHCup, which installs the following tools:

- GHC: the Glasgow Haskell Compiler
- cabal-install: the Cabal installation tool for managing Haskell software
- Stack: a cross-platform program for developing Haskell projects
- haskell-language-server (optional): A language server for developers to integrate with their editor/IDE

We recommended installing Visual Studio Code with an extension for Haskell language support. The Haskell language support extension is powered by the Haskell Language Server, which is installed optionally via GHCup.

GHCup

To install GHCup and the Haskell toolchain, run the following in a terminal (as a non-root user):

```
$ curl --proto '=https' --tlsv1.2 -sSf https://get-
ghcup.haskell.org | sh
```

How to uninstall? GHCup installs only into the following directory, which can be removed anytime: `~/.ghcup`. Alternatively, at any point in time you might run 'ghcup nuke' to uninstall GHCup and Haskell toolchain.

GHCup

During installation, answer a few questions:

1. Do you want ghcup to automatically add the required PATH variable? [P] Yes, prepend [A] Yes, append [N] No [?] Help (default is "P"). **P**
2. Do you want to install haskell-language-server (HLS)? [Y] Yes [N] No [?] Help (default is "N"). **Y**
3. Do you want to enable better integration of stack with GHCup? [Y] Yes [N] No [?] Help (default is "Y"). **Y**

GHCup

Which versions? To install other GHC versions and tools, run: `ghcup tui`

Important: Make sure you install and select as default a version of GHC that is compatible with HLS. These are clearly marked in the TUI. **(2024-09-14)** As of this date, the recommended version of GHC is 9.4.8.

(2022-11-14) GHC 9.4.3 (2022-11-03) and GHC 9.2.5 (2022-11-07) fixes issues on M1/arm64/aarch64 platform (#22282, #21964).

Visual Studio Code

- Download and install Visual Studio Code.
- Open VS Code and install Haskell language support extension:
  - View > Extensions > Search: Haskell > Install

Hello World

Create a file `helloworld.hs`:

```
main :: IO ()
main = putStrLn "Hello world!"
```

Compile it by running `ghc helloworld.hs`

Execute your program by running: `./helloworld`

Installation guide available on Moodle

Download file `how-to-get-started.pdf`.

Let's try out a few things...

- Simple arithmetic (+, -, *, /, ^)
- Boolean algebra (&&, ||, not)
- Mathematical functions (div, mod, sqrt, abs, floor, round, ceiling)

In [19]:
```
3+2
6-1
True || False
div 10 3
mod 10 3
sqrt 4
abs (-10)
floor 7.1
round 7.1
ceiling 7.1
```

```
3+2
(+) 3 2
10 `mod` 3
mod 10 3
```

```
5
5
True
3
1
2.0
10
7
7
8
5
5
1
1
```

Functions

- Function application in C: `foo(x)`.
- Function application in Haskell: `foo x` (notice the **space** between the function `foo` and argument `x`).
- In Haskell, function start with **lower case**, e.g. `min 10 12`.
- Functions starting with letters are **prefix**.
- Operators (which are also functions) are **infix**.
- Functions can also be used like **infix** operators (e.g. `10 `div` 3`) and operators like **prefixed** functions (e.g. `(+) 2 3`).

In [1]:
```
sum [1..10]

10 + 5

10 `div` 3

(+) 10 5
```

```
55
15
3
15
```

Precedence

- You can check precedence with `:info`. The higher the level (from 0 to 10), the stronger the binding.
- Functions have the highest precendece (10).
- You may use parentheses `( )` or `$`.
  - `($): infixr 0`
    - infixr tells us it's an infix operator with right associativity.
    - 0 tells us it has the lowest precedence possible.
    - In contrast, normal function application (via white space) is left associative.

In [3]:
```
:info (*)

(2+3)*4
(*2) $ sum [1,2,3] + 4

:info ($)

reverse "james" ++ "bond"
reverse ("james" ++ "bond")
reverse $ "james" ++ "bond"
```

```
type Num :: * -> Constraint
class Num a where
...
(*) :: a -> a -> a
...
-- Defined in ‘GHC.Num’
infixl 7 *
```

```
20
20
($) :: (a -> b) -> a -> b -- Defined in ‘GHC.Base’
infixr 0 $
```

```
"semajbond"
"dnobsemaj"
"dnobsemaj"
```

Lists

- Lists are the most used data structure in Haskell.
- Try out a few functions over list, from Prelude.
  - `head` , `tail` , `last` , `init`
  - `null` , `reverse` , `take` , `drop`
  - `maximum` , `sum` , `product` , `length` , `elem`
  - concatenate lists `(++)` , list constructor `(:)` appends an element to a list, list index `(!!)`

In [5]:
```
list = ['a','b','c']

head list
tail list
last list
init list

maximum [2,3,6,1]
sum [1,2,3]

length "hello"

'a' `elem` "Software"
elem 'a' ['S','o','f','t','w','a','r','e']

y = 'S' : "oftware"
z = "Software" ++ " Foundations"
[1,2,3] !! 2
```

Use infix

Found:

```
elem 'a' ['S', 'o', 'f', 't', 'w', 'a', 'r', 'e']
```

Why Not:

```
'a' `elem` ['S', 'o', 'f', 't', 'w', 'a', 'r', 'e']
```

```
'a'
"bc"
'c'
"ab"
6
6
5
True
True
3
```

List comprehension

- ranges `[1..n]`
- list comprehension `[x^2 | x <- [1..n]]`

In [6]:
```
{- [1,3..50]
-}


[1..10]

[x^2 | x <- [1..10]]

"Programming" ++ " Fundamentals" ++ " 3"

1 : [2,3,4]
```

```
[1,2,3,4,5,6,7,8,9,10]
[1,4,9,16,25,36,49,64,81,100]
"Programming Fundamentals 3"
[1,2,3,4]
```

Tuples

- Tuples bundle a fixed number of values into a single single value.
- A tuple may bundle values of different types.
- `fst`, `snd`.

In [8]:
```
a = (1,"Alice")

fst a
snd a

-- [1,'a']
-- all elements of a list must be of the same type; the above definition is inv

coordinates = (1.2, 5.0)
coordinates
```

```
doubleXCoordinate (x,y) = (x*2, y)

doubleXCoordinate coordinates
```

```
1
"Alice"
(1.2,5.0)
(2.4,5.0)
```

Pattern matching

- When defining functions, you can define separate function bodies for different patterns.
- This leads to really neat code that's simple and readable.
- You can pattern match on any data type — numbers, characters, lists, tuples, etc.

In [ ]:
```
lucky 7 = "Congrats! You hit the lucky number seven!"
lucky x = "Sorry, you're out of luck!"

--first :: (a, b, c) -> a
first (x, _, _) = x

--sum' :: (Num a) => [a] -> a
sum2 [] = 0
sum2 (x:xs) = x + sum2 xs

sum2 [1,2,3,4]
```

```
10
```

Exercises 💪

(1) String ends with?

- https://www.codewars.com/kata/51f2d1cafc9c0f745c00037d

(2) Get the Middle Character

- https://www.codewars.com/kata/56747fd5cb988479af000028

In [ ]:
```
-- (1) String ends with?
solution :: String -> String -> Bool
solution xs ys = drop (length xs - length ys) xs == ys

solution "abc" "g"

-- (2) Get the Middle Character
-- drop first and last element of the list until we have a list with only 1 or
getMiddle :: String -> String
getMiddle [x] = [x]
getMiddle [x,y] = [x,y]
getMiddle xs = getMiddle (drop 1 (init xs))

-- alternative approach: fast & slow
-- duplicate list, on one list skip one element at the time, on the other skip
-- when we reach the end of the second list, we are in the middle of the first
getMiddle' :: String -> String
```

```haskell
getMiddle' xs = f xs xs
  where
    f (a:_) [_] = [a]
    f (a:b:_) [_,_] = [a,b]
    f (_:as) (_:_:cs) = f as cs
    f _ _ = []

getMiddle' "a"
getMiddle' "ab"
getMiddle' "abc"
getMiddle' "abcd"
getMiddle' "abcde"
```

```
False
"a"
"ab"
"b"
"bc"
"c"
```