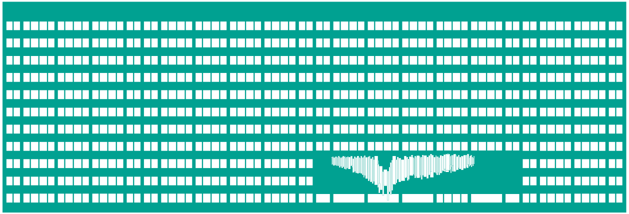
VŠB TECHNICKÁ UNIVERZITA OSTRAVA

VSB TECHNICAL UNIVERSITY OF OSTRAVA



www.vsb.cz

Basics of Functional Programming

behalek.cs.vsb.cz/wiki/Functional_Programming

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

September 26, 2023

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Functional Style of Programming

- Along with logical programming represents declarative style of programming.
- *Omitting some details,* Declarative style of programming is opposite to imperative style of programming.
- Imperative programming
  - Program is a sequence of statement.
  - Exact steps defining what to do.
  - Statements have side effects (`a=5;`).
  - Different meaning to *things* from math.
- Functional programming
  - Program compose from a set of functions that defines what *is*.
  - Program's evaluation is then an evaluation of a main expression.
  - No implicit state, functions have no side effects $\rightarrow$ referential transparency *(good stuff...:-)*
  - Closer to math - for example the term variable have its original meaning.

Functional Programming - Good / Bad / Ugly I

- Good
  - Excellent abstraction mechanisms (high order functions, function composition).
    - Elegant and concise program $\rightarrow$ shorter then imperative counterparts, more error prone $\rightarrow$ easier to maintain
  - New possibilities like lazy evaluation $\rightarrow$ allows us to work with infinite structures.
    - Very nice mechanism how to handle *(big)* data.
  - Referential transparency allows compiler (or other tools) to *reason* about program's behaviour and even *prove* its properties.
  - No side effects $\rightarrow$ efficient and easy (automatic) program's parallelization.
- Bad
  - Debugging - harder (harder to make nontrivial mistakes, if there are no type errors).
  - Performance
    - Restricted by current hardware - more suitable for imperative languages.
    - To get *good* performance we offten need to make sacrifices (OCaml - imperative features, no lazy evaluation) and/or perform complex optimizations.

Functional Programming - Good / Bad / Ugly II

- Ugly - How often are ~~pure~~ functional languages used in real applications?
  - Some languages were successful in some areas like Elrang - Elixier for run-time systems.
  - We will be using **Haskell** (20th (or so) on list for open-source projects, Facebook anti-spam engine).
  - Popular languages implement multiple programming paradigms, in some functional programming is dominant (Python, Javascript, C#), some technologies are even closer to the essence of pure functional programming (LINQ in C#).
  - For some components, you do not even know in which language they were built (Scala runs on JVM, Elm compiles to Javascript).

- Still, functional **style** of programming is often used even in traditional imperative languages.

Haskell

- So, why we learn Haskell?
- Pure functional language → basic principles are exposed
  - It is *simpler* then different dialects of ML, Lisp, ...
- Mature language + plenty of tools
- Compared to other (imperative) languages **very** simple.
- Difficulty of learning new programming language is in:
  - learning a syntax and a semantic of constructs from the language → *simple* for Haskell
  - learning how to solve the problem in *functional way* → that will be **the main scope** of this course
  - learning how to use API → takes usually most time, we will be using only small part of API, moreover, most of these functions we will *write* ourselves

Functions Wherever You Look I

- Function
  - Based on input parameters returns output value (values).
  - Definitions:

```
name :: Type
name = expression
```

  - Examples:

```
doubleMe x = x * x
plus x y  = x + y

max x y | x > y = x
        | x <= y = y

factorial 0 = 1
factorial n = n * factorial (n-1)
```

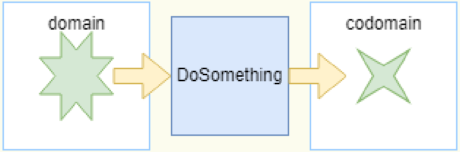
Functions Wherever You Look II

- A process of giving particular inputs is called **functional application.**
  - Example:

    ```
    plus 4 5
    ```
- Functional programming
  - A program is a set of function definitions.
  - These functions captures our particular problem.
  - Desired computation is then an evaluation of *main* expression.
    - How to evaluate an expression: *(7-3)*2* ?
    - There are various options and strategies *(lazy evaluation...)*.

What is a function? More questions?

- *A box* that takes an *item* from functions domain and transforms it into an *item* from functions codomain.



- What is an item in the input domain or output codomain?
- How can we define a function? Which tools we need?
- What can we do with funcitons? Can we compose new functions from existing functions? Can function use other functions?
- How can we build a program from functions?

Input and output of functions

- Can we use more than one item in the input or output?
  - *From definition*: function takes $x \in X$ and *produces* exactly one $y \in Y$, it is usually denoted as: $f : X \to Y$.

$$f : \mathbb{N} \to \mathbb{N}$$
$$f(x) = x + 5$$

- But there are binary (n-ary) functions?
  - It is defined as: $f : X \times Y \to Z$
  - Element in Cartesian product is: $(x, y) \in X \times Y$

$$g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$g(x, y) = x + y$$

- We can use another clever trick to introduce more arguments: *a function can return a function* (more details later).

$$h : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

How can we define a function?

- In computer science, a function is defined by:
  - is **type** signature;
  - and by *a rule*, that assigns an output value for an input value.
- Generally, in computer science, this rule can be a set of instructions (statements).
- In Haskell it is restricted to **expressions** (in fact same as mathematical expressions).
  - It is a *syntactic* entity that may be evaluated to determine its value.
  - It is a combination of one or more constants, variables, functions, and operators.
  - Expressions are evaluated according to its particular rules of precedence and of association (in Haskell, you can safely assume, that it is the same as in math).
- To create expressions we need:
  - basic elements - numbers, characters... $\rightarrow$ basic data types;
    - There is a type `Int` roughly representing natural numbers (set $\mathbb{N}$).
  - basic functions and operators;
    - There are binary operators like $+, -, *$ that works with the type `Int`.
    - Their type signature is: `Int -> Int -> Int`
  - some *mechanisms* to define more complex expressions.

How to build more complex functions?

- We need some syntactic constructs for *branching* $\rightarrow$ Nothing new, we know it from math!

$$abs(x) = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

- A function can **use** another functions! $\rightarrow$ We can define hierarchy and reuse repeating patterns.

$$f(x, y) = abs(x) + abs(y) + abs(1) + 1$$

- Functions are first-class citizens in Haskell (as in math) $\rightarrow$ a function can be used as parameter or as a return value $\rightarrow$ it is a normal value.
  - From mathematics, we know function composition (denoted as $\circ$).
  - It is an *operator* (like $+$), that takes two functions f and g, and produces a function: $h = g \circ f$ such that: $h(x) = g(f(x))$.

$$test1(x) = (sin \circ abs)(x)$$
$$test2(f, x) = (f \circ abs)(x)$$

How can we build a program from functions?

- We decompose the problem into smaller parts.
- These parts are implemented as elementary functions.
- We have a mechanisms, how to combine these functions together.
- So, a whole program will be a set of functions.
- Finally, we need some starting point for our program → usually a function named **main**.
- By evaluation of **main** function (its right-side expression), the program is performed.
- Still, we are missing some key ideas:
  - Can we compose any two functions? → Are there some rules for working with functions?
  - Long running computation → Something like a cycle in C++ → recursive functions.
  - How to store real data? → User defined data types

Tools

- Haskell Platform
  - Glasgow Haskell Compiler (Interpreter)
  - Stack - package manager
  - Hoogle - API documentation
- Visual Studio Code
  - *Haskell*
  - Haskero
  - Haskell GHCi Debug Adapter Phoityne

GHCi

- Basic usage

```
>ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude>2*(3+5)
16
```

- File containing user's definitions

  `>ghci example.hs`

- GHCi commands:

```
:edit|:e [file.hs]
:load [file.hs]
:reload
:quit
:?
```

GHC

- Basic usage

```
main = do putStr "Your name:"
          name <- getStr
          putStr "Hello " ++ name
```

- Traditional compiler:

```
>ghc example.hs
```

- Result will be an executable file.
- Where are the functions? What is this `do`?
  - Monads - sequence of actions enveloped by *pure functions*.
  - Closer to *real world* program in Haskell.

References

- Lipovaca M.
  Learn You a Haskell for Great Good!: A Beginner's Guide (1st ed.).
  No Starch Press, San Francisco, CA, USA, 2011 - for free at:
  http://learnyouahaskell.com/
- O'Sullivan B., Goerzen J., Stewart D.
  Real world Haskell.
  O'Reilly Media, Inc. 2008. ISBN:0596514980 - for free at:
  http://book.realworldhaskell.org/read/
- Thompson S.
  The Haskell: The Craft of Functional Programming (3rd ed.).
  Addison-Wesley Professional, October 2, 2011, ISBN-10: 0201882957.

Practical demonstration

- Preparing work environment.
- Usage of GHC Interpreter.

Input file format

- Files with extension **.hs** (that is what we will use)

```
module Example where
-- Function computing sum of two numbers
sum x y = x + y
```

- Files with extension **.lhs**

```
> module Example where

Function computing factorial
> f n = if n == 0 then 1 else n * f (n-1)
```

Module Prelude

- Like in other languages, source codes are divided into separated *parts*, named modules in Haskell (packages in Java, namespaces in C#,...), will be explained in details later.
- Modules are composed from functions and user defined data types.

```haskell
module Ant where ...
...
```

- Modules can be imported

```haskell
module Example where
import Ant
...
```

- Special package imported by default: `Prelude`.
  - Some function names are *taken*.
  - Hiding functions:

```haskell
import Prelude hiding (max, min)
```

Names

- **Identifiers** - begin with letter, followed by a sequence of letters, digits, underscores and single quotes.

  `abc, h_e_l_l_o, hello', hello123, HeLLo, Hello`
- Names used in definition for *values* begin with **small** letters.
- Types and constructors begins with **capital** letters.

```
data Tree a  = Leaf a
             | Node (Tree a) (Tree a)

f::Int -> Int -> Int
f x y = x + y
```

Layout I

- Indentations are important! They define the structure of your source files.
- Internally, constructs with { } ; are used (similarly to C++).
- Strange errors with ';'

```
funny x = x +
        1
        ERROR ... : Syntax error in expression (unexpected ';').
```

- Basic rule → **all constructs with the same indentation belongs to the same scope**.
- Inner scope requires a bigger indentation.

- Be careful, spaces are not the same as tabulators.
- Examples

Layout II

- ```
  maxsq x y
            | sqx > sqy = sqx
            | otherwise = sqy
            where
                    sqx = sq x
                    sqy = sq y
                    sq :: Int -> Int
                    sq z = z*z
  ```
- ```
  maxsq x y
            | sq x > sq y = sq x
            | otherwise = sq y
                    where
                        sq x = x * x
  ```

Are there some rules for working with functions?

- Consider function:

$$f : \mathbb{R} \rightarrow \mathbb{R}$$
$$f(x) = \sqrt{log(x)}$$

  Is the definition OK?
- What options we have, if we want to work with this function?
  - Better specification of the function's domain $\rightarrow$ In terms of computer science, introduce a new type like: $X = \{x \in \mathbb{R}, x \geq 1\}$, then $f : X \rightarrow \mathbb{R}$
  - Be *really careful* when evaluation this function $\rightarrow$ while evaluating, we will handle possible exceptional situations.
  - Do not worry at all (programmer is always right, what will be will be...:-)

Type system

- *Very important part of any programming language.*
- **Loosely definition**
  - Type system associates one (or more) type(s) with each program value.
  - By examining the flow of these values, a type system attempts to prove that no "type error" can occur.
- Assigning data types (typing) gives meaning to collections of bits.
- Types usually have associations either with values or with objects such as variable.
- Types allow programmers to think about programs at a higher level than the bit or byte, not bothering with low-level implementation.
- Use of types may allow a compiler to detect meaningless or invalid code.
  - Especially true for pure functional languages, where it is hard *to make mistake* in a program that has correct types.

Basic functionality of a type system (in Haskell) I

- **Haskell** has static, strong and safe type system. Moreover, it supports polymorphism.
  - Static typing *(C, C++, Java, Haskell...)*
    - Type checking is performed (mostly) during compile-time.
    - Static typing is a limited form of program verification.
    - It allows many errors to be caught early in the development cycle.
    - Static type checkers are conservative - they will reject some programs that may be well-behaved at run-time, but that cannot be statically determined to be well-typed.
  - Dynamic typing *(Javascript, Python, PHP...)*
    - Majority of its type checking is performed at run-time.
    - Dynamic typing can be more flexible than static typing. For example by allowing programs to generate types based on run-time data.
    - Run-time checks can potentially be more sophisticated, since they can use dynamic information as well as any information that was present during compilation.

```
var x := 5;      // (1)  (x is an integer)
var y := "37";   // (2)  (y is a string)
var z := x + y;  // (3)  (? - Visual Basic = 42, Javascript "537")
```

Basic functionality of a type system (in Haskell) II

- Strongly typed languages - do not allow undefined operations to occur.
- Weak typing means that a language implicitly converts (or casts) types when used.
- Type safe - is language if it does not allow operations or conversions which lead to erroneous conditions.
- Memory safe - for example it will check array bounds (resulting to compile-time and perhaps run-time errors).

```
int x = 5;
char y[] = "37";
char* z = x + y; //z points five characters after y
```

- Polymorphism
  - The ability of code (in particular functions, methods or classes) to act on values of multiple types, or the ability of different instances of the same data-structure to contain elements of different types.
  - Type systems that allow polymorphism generally do so in order to improve the potential for code re-use.

Basic functionality of a type system (in Haskell) III

```
Animal obj = new Horse();
obj.sound();
```

```haskell
length :: [a] -> Int -- a is a type variable
length [] = 0
length (x:xs) = 1 + length xs
```

- **Type checking** - the process of verifying and enforcing the constraints of types.
- **Type interference**
  - Strongly statically typed languages
  - Automatic deduction of the data types
  - Hindley-Milner type system

Basic data types I

- `1::Int`

  `+, -, *, ^, div, mod, abs, negate, ==`
- `'a'::Char`
  - Special characters: `'\t'`, `'\n'`, `'\\'`,`'\''`, `'\"'`
  - Prelude functions:

    `ord :: Char -> Int, chr :: Int -> Char, toUpper, isDigit`
  - Library `Char`
- `True,False::Bool`

  `&&, ||, not, ==`
- `3.14::Double` (`3.14::Float`)

Basic data types II

```
+, -, *, /, ^, **,
==, /=, <, >, <=, >=
abs, acos, asin, sin, cos,
celing, floor, exp, fromInt, log, negate, pi
```

- `"Hello"::String`
  - Defined as: `type String = [Char]` - list of characters, lists will be explained later.
  - Example: `"Hello world"`

Summary, how to work with these basic data types

- Each type is defined by its *unique* name (starting with capital letter in Haskell) - `String`.
- There are some build-in types, later we will learn a way to build our own types.
- There is a way, how to write a constant value (literal) of these types.
- *Values* belongs to exactly one type.
- There are basic functions and operators working with these basic data types.
- If you want to define a type (for a value or a function), you can use: `1::Int`

Function's definition and its type

- Definition

```
name :: Type
name parameters = expression
```

- Example

```
square :: Int -> Int  -- optional!
square n = n * n

sum :: Int -> Int -> Int
sum x y = x + y
```

- Function application

```
square 5 = 5 * 5
square (2+4) = (2+4) * (2+4)
sum 4 5
```

So, how to really write a function in Haskell?

$$f : \mathbb{N} \to \mathbb{N}$$
$$f(x) = x + 5$$

```
f :: Int -> Int
f x = x + 5
```

$$g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$g(x, y) = x + y$$

```
g :: (Int, Int) -> Int
g (x,y) = x + y
```

- But on previous slide, there was: `sum :: Int -> Int -> Int`
- *Real way*, how to write binary function in Haskell.

So, how to really write a function in Haskell?

- What is: `->`
  - In fact, it is *an operator* with **right** associativity.
  - So, `sum :: Int -> Int -> Int` is equivalent to `sum :: Int -> (Int -> Int)`.
  - How can we understand it? $\rightarrow$ Remember, functions are also values $\rightarrow$ It is a function that returns a function!
- Functional programming languages are *based* on Lambda calculus.
  - Lambda abstraction $\Leftrightarrow$ anonymous function: $\lambda x.x + 5$
  - `sum x y = x + y` $\Leftrightarrow \lambda x.(\lambda y.x + y) \Leftrightarrow \lambda xy.x + y$
- What can we do with such lambda expressions? $\rightarrow$ We can apply them on some values.
  - How it is evaluated? $\rightarrow$ We *substitute* the corresponding variable with the value.
  - $(\lambda x.x + 5 \quad 2) \rightarrow (x + 5)[x := 2] \rightarrow 2 + 5 \rightarrow 7$
  - $((\lambda x.(\lambda y.x + y) \quad 2) \quad 3) \rightarrow ((\lambda y.x + y)[x := 2] \quad 3) \rightarrow (\lambda y.2 + y \quad 3) \rightarrow (2 + y)[y := 3] \rightarrow 2 + 3 \rightarrow 5$
- Functions application is left associative.
  - `((sum 2) 3)` $\Leftrightarrow$ `sum 2 3`

So, how to really write a function in Haskell?

- Where are we now?
  - We can write a function using the basic data types.
  - The function will be using basic operators and functions associated with these types.
  - We know, how to create a *n-ary* function and how to use it.
- What are the most fancy things we can do now?

```
applyTwice :: (Double -> Double) -> Double -> Double
applyTwice f x  = f (f x) -- applyTwice sin 1
```

  - What if we want to use it with `Int` and function `negate`?
  - We talked about polymorphism → We can use parameters (variables) in type definition.

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x  = f (f x) -- applyTwice sin 1, applyTwice negate 1
```

  - Still, we need to be careful, types needs to be valid.

So, how to really write a function in Haskell?

- Can we use more parameters in type definition?
- Function composition: $h = g \circ f$ such that: $h(x) = g(f(x))$.

```
compose :: (b -> c) -> (a -> b) -> (a -> c)
compose g f = -- ???
```

- We still do not have tools to build more complex functions...

So, how to really write a function in Haskell?

- Can we use more parameters in type definition?
- Function composition: $h = g \circ f$ such that: $h(x) = g(f(x))$.

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

- We still do not have tools to build more complex functions...

Interlude - Type classes - what does the => mean?

- Assume the following function sum, what is its type?

```
sum x y = x + y
```

```
Prelude> :type sum
sum :: Num a => a -> a -> a
```

- Num is a type class for all numbers, a is a type variable. It can be any *numeric* type, for example Double or Int.

```
Prelude> :type (sum (4::Int) 4)
(sum (4::Int) 4) :: Int
Prelude> sum 1.1 1
```

- We can restrict the function type.

```
sum :: Int -> Int -> Int
sum x y = x + y
```

Practical demonstration

- Identifiers in Haskell
- What is a Haskell program?
- Primitive data types.
- How to write an expression.
- Calling some standard functions.
- Using some operators for primitive data types.
- Checking types of these functions.
  - What is the meaning of *the nonsense* that operator + returns as its type?

Better functions

- What we are missing for writing more advanced functions? $\rightarrow$ **Branching**

$$abs(x) = \begin{cases} x, & \text{if } x \geq 0 \\ -x, & \text{otherwise} \end{cases}$$

$$min(x, y) = \begin{cases} x, & \text{if } x \leq y \\ y, & \text{if } x > y \end{cases}$$

Syntax of function's definition

- **Pattern matching**
- Guard expressions

```
max :: Int -> Int -> Int
max x y | x>=y = x
        | otherwise = y
```

- Local definitions - `where`
  - Local definitions need to have *bigger indentation*.
  - ```
    initials :: String -> String -> String
    initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
         where f = head firstname
               l = head lastname
    ```
- *All* these syntax constructs can be used to define a single function.
    ```

Pattern matching

- Several function definitions (equations) with different *patterns*.

```
f  pat11  pat12  ...  = rhs1
f  pat21  pat22  ...  = rhs2
...
```

- First equation that can be unified with given parameters is chosen.

```
f  value1 value2 ...
```

- If there is none → **error**
- The most basic patterns are:
  - **constants**;
  - **variables**.

Pattern matching - naive example

- Basic example:

```
sayMe 1 = "One!"
sayMe 2 = "Two!"
```

- Some function applications:

```
*Main> sayMe 1
"One!"
*Main> sayMe 3
"*** Exception: input.hs:(1,1)-(2,16): Non-exhaustive patterns in
function sayMe
```

- First equation that can be unified with given parameters is chosen.

```
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe x = "Something else"
```

Better functions

- A function can have multiple definitions, they must differ in their parameters - *patterns*.
  - More general patterns (containing variables) must be defined after more specific patterns (with constants).
- Each such definition can use guard expressions.
- Each such definition can have its local `where` section.
- Definitions are then processed from top to bottom, for each set of input parameters **exactly one** right side is chosen.

```
funny 0 y z | z < y = z
            | otherwise = y
funny 1 y z | y == z = abs z where
  abs x | x < 0 = -x
        | x >= 0 = x
funny x y z = x + y + z
```

- We are finally ready to talk about recursion :-)

Recursion

- Recursion generally contains:
  - A simple base case (or cases) — a terminating scenario that does not use recursion to produce an answer.
  - A recursive step — a set of rules that reduces all successive cases toward the base case.
- Recursion is frequent occurrence in math.
  - Many axioms are recursive - natural numbers.
  - Profs - mathematical induction.
- Fractals - can usually be drawn using recursion.

Pattern matching - better example

- The answer to most questions in Haskell is recursion.
- Recursive function is a function that *calls* itself.
  - Very nice is a *tail recursion*..
- Simple example of recursion is **factorial** in math.
  $n! = n \times (n-1) \times (n-2) \times (n-3) \times \cdots \times 3 \times 2 \times 1$

$$fact(n) = \begin{cases} 1, & \text{if } n = 0 \\ n * fact(n-1), & otherwise \end{cases}$$

- How to write it in Haskell?

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

- Pattern matching is a syntactic sugar based on case expression.

Programs evaluation

- Program's evaluation is equivalent to evaluation of the expression on the right side of $main$ function.
- How to evaluate expression?
  - Repetitively, we take the operation with the biggest priority and solves it.
  - If there are more items with the same priority, associativity is used to determine the operation to process.
  - While programs compose from functions, the most interesting operation is *function's application*.
- Function application have the highest priority and it is left associative.
- Function application generally replaces function call with its right-hand side expression (substituting the parameters).
  - If there are multiple definitions, right-hand side expression is chosen based on parameters.

Interlude - operators and priority

- What about the following expression, how to understand it? Which brackets are necessary?

  ```
  n * factorial (n-1)
  ```

- In Haskell, like in other languages (C++), there are functions and operators.
- Operators are composed from characters:

  ```
  ! # & $ % * + - . / < > = ? \ ^ | : ~
  ```

- Operators are using *infix* notation (`5 + 3`) and are strictly binary.
- Priority rules:
  - Function application has a highest priority.
  - Operator * have a higher priority then +.
- Operators and their priority will be explained later. **If not sure use brackets!**

Pattern matching - application

- Factorial definition

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

- Functions application step by step.

```
factorial 5 = 5 * factorial (5-1) = 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
```

Recursion

- In computer science, recursion is a method of solving a computational problem.
  - A common algorithm design tactic is to divide a problem into sub-problems of the same type as the original, solve those sub-problems, and combine the results (divide and conquer).
  - Another common algorithm design tactic is dynamic programming - we save the intermediate steps in recursion to simplify further computation.
- Type of recursion:
  - Single recursion *vs.* Multiple recursion
  - Direct recursion *vs.* Indirect recursion
  - Structural recursion *vs.* Generative recursion

Fibonacci numbers

- Factorial is simple, but consider Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13 ...
- Lets define a function that computes n^{th} number in the sequence.

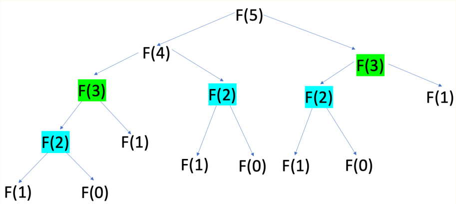
$$fib(x) = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{if } x = 1 \\ fib(x-1) + fib(x-2), & otherwise \end{cases}$$

- Is there only one way to solve the problem?
  - For most *problems* we have several *algorithms* solving this problem.
  - Even if use recursion, there can be more recursive algorithms solving this problem.
- Solving the problem, we can follow the definition.

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Fibonacci numbers

- Lets check our solution, it is even a good solution?
- How many *steps* are we expecting and how many will be performed by our code?




```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

- What's wrong in our solution? $\rightarrow$ We are computing the same intermediate steps.
- *Roughly speaking*, in terms of the computer science, we have created an algorithm with exponential time complexity.
  - Fibonacci numbers grow at an exponential rate equal to the golden ratio $\varphi = (1 + \sqrt{5})/2 \cong 1.61803$.

Fibonacci numbers

- Can we do better? How we (*humans*) solve it with pen and paper?
  $(0,1) \to (1,1) \to (1,2) \to (2,3) \to (3,5) \to (5,8) \to (8,13)$...
- So, how can we improve our solution?
  1. We can save the intermediate steps in some kind of *dictionary* (we do not have skills to do that).
  2. We can rewrite the solution using the strategy *from bottom to top* instead of *from top to bottom*.

```
fib n = fst (fibCount n) where
  fibCount 0 = (0,1)
  fibCount n = fibStep (fibCount (n-1))
  fibStep (x,y) = (y, x+y)
```

Fibonacci numbers

```
fib n = fst (fibCount n) where
  fibCount 0 = (0,1)
  fibCount n = fibStep (fibCount (n-1))
  fibStep (x,y) = (y, x+y)
```

- How it will be evaluated?

```
fib 3 = fst (fibCount 3)
      = fst (fibStep (fibCount 2))
      = fst (fibStep (fibStep (fibCount 1)))
      = fst (fibStep (fibStep (fibStep (fibCount 0))))
      = fst (fibStep (fibStep (fibStep (0,1))))
      = fst (fibStep (fibStep (1,1)))
      = fst (fibStep (1,2))
      = fst (2,3)
```

Syntax for expressions

- Expressions can be used **anywhere**!
- We already know expressions - function's application and a usage of operator.
- `if` *expression* - it is similar to ternary `?:` operator from C++ (`(x>y)? x : y`)

```
max x y = if x > y then x else y
```

- `case` expression

```
describe :: Int -> String
describe n = "The number is " ++ case n of
       0 -> "zero."
       1 -> "small."
       x -> "large."
```

- `let` expression

```
cylinder r h = let sideArea = 2 * pi * r * h
                   topArea = pi * r ^2
               in  sideArea + 2 * topArea
```

Practical demonstration

- Priority in expressions.
- Implementing some simple functions.
- Defining function's type.
- Type inheritance.
- Where can I store some data if needed? `-- How to declare variables?`
- How can I write *a cycle*? `-- I NEED my cycles!`

Lists

- Probably the most used data structure in functional Languages (for C++ the equivalent will be array).
- Lists are a **homogeneous** data structures.
  - A list can contain only elements with the same data types.
  - `[1,2,3] -- OK`
  - `[1,'a',3] -- Wrong`
  - `"hello" == ['h','e','l','l','o']`
- Informally, the term **syntactic sugar** refers to **a nicer way** how to write something.

List - Basic concept

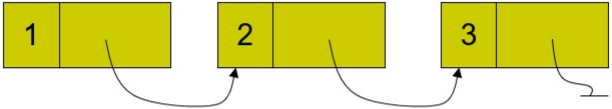


Figure: A scheme of the list.

- Element in a list contains the data and a reference to other elements.
- The last element points *nowhere* (usually a sort of **null** reference).

- How to work with such a list?
  - What if i want to get n^{th} element in the list?

List - Basic concept

	Array	List	Winner
Get n^{th} element	pointer arithmetic's	need to go trough the list	Array
Add element at the beginning	new memory, copy all	easy, just add	List
Add element at other positions	new memory, copy all	rebuild the list	Tie
Remove element at the beginning	new memory, copy relevant	easy, get second element	List
Remove element at other positions	new memory, copy relevant	rebuild the list	Tie
Modify first element	get and modify	new first, stitch the tail	List
Modify any other element	get and modify	rebuild the list	Array

Table: Informal comparison of an array and a list

Efficient usage of lists usually requires different algorithms (approach).

Lists in Haskell

- Definition

```
data List a = Cons a (List a)
            |  Nil
```

- Application of (syntactic) sugar
  - `List a` → `[a]`
  - `Cons a` → `: -- a -> [a] -> [a]`
  - `Nil` → `[]`
- List literals
  - `[1,2,3] :: [Int]`
  - `1:2:3:[] :: [Int]`
- Patterns for lists
  - Empty list `[]`
  - Non-empty list `(x:xs)`

Function working with lists

- List `length` function

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

- Application of this function

```
length [1,2,3] = 1 + length [2,3]
               = 1 + 1 + length [3]
               = 1 + 1 + 1 + length []
               = 1 + 1 + 1 + 0 = 3
```

- What is the type of this functions?
  - Do we even know or care about the type of the element in the list?

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

Tuples

- Haskell have build in ordered tuples `(a,b,c,d,...)`

```
(1,2) :: (Int,Int)
(1,['a','b'], "abc")::(Int, [Char], String)
() :: ()
```

- Unlike homogeneous lists, tuples can have elements of different types.
- Example of a *pattern* for tuples:

```
addThem :: (Int, Int) -> Int
addThem (x,y) = x + y
```

- Build in functions working with tuples.

```
addThem :: (Int, Int) -> Int
addThem x = fst x + snd y
```

Type classes 101 (more details later)

- Not the same as classes form Java or C++.
- Type of the operator ==

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

- Definition of the type class Eq

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y =  not (x == y)
```

- A type can become a member of this class, if it provides functions and operators that the class defines.

Practical demonstration

- `List`
  - Basic type storing *bigger* data - `List -- Who needs arrays...`
  - Functions for *all* lists (type variables → polymorphism).
  - Simple functions *going trough* the list.
  - Nice patterns available to handle lists.
- Tuples

Basic type classes

- Class `Eq` - `==` `/=`
- Class `Ord` - `>` `<` `>=` `<=` `compare`
- Class `Show` - `show :: a -> String`
- Class `Read` - `read :: (Read a) => String -> a`
  - Why is it not working?

```
ghci> read "4"
<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
      `Read a' arising from a use of `read' at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```

  - We can repair it by: `read "4" :: Int`
- Class `Enum` - `succ, pred`
- Class `Bounded` - `minBound, maxBound :: (Bounded a) => a`

Basic type classes for numbers

- Basic relations between numeric classes `Num` (not all numeric classes are mentioned)
  - `Num` → `Real`, `Fractional`
  - `Real` → `Integral`, `RealFrac`
  - `Fractional` → `RealFrac`, `Floating`

  - `Integral` → `Int`, `Integer`
  - `Floating` → `Float`, `Double`

Conversion between numbers

- There are functions taking a value and pushing it *higher* in the type hierarchy.

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

fromIntegral (length [1,2,3,4]) + 3.2
```

- There are functions changing the type class to a class in the same *level*

```
realToFrac :: (Real a, Fractional b) => a -> b
```

- Special functions

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

- There is a lot of functions converting types of numeric values.

```
fromInteger, toInteger, fromRational, toRational, ceiling,
floor, truncate
```

Partially applied functions

- In theory, every Haskell function only takes one parameter.
  - But we were using functions with several parameters? → **curried functions**
  - Definition `max :: (Ord a) => a -> a -> a` can be rewritten as: `max :: (Ord a) => a -> (a -> a)`.
- What really happens when a function is applied? → `(max 2) 3`
  - It will work even if we specify just one parameter `max 2` → **partially applied function**

```
Prelude> :t max 2
max 2 :: (Ord a, Num a) => a -> a
```

- **Functions can return other functions**

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
-- compareWithHundred x = compare 100 x
compareWithHundred = compare 100
```

High order functions

- Functions can have other functions as parameters.

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

- Useful example of a high order function.

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

- How it is used?

```
map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]  -- [1,3,6,2,2]
```

Practical demonstration

- What is a type class?
- Useful type classes.
- Function is a first class citizen in Haskell.
  - Partial application, curried functions.
  - Usage of high order functions.
- Some tips how to write complex functions.
  - Dividing complex computations into smaller functions.
  - Construct `let ... in`

Operators in Haskell

- Operators are composed from characters: `!#&$%*+-./<>=?\^: |`
- Operators are using *infix* notation (`5 + 3`).
- Important for operators are priority and *associativity*.
- Operators can be used as functions.

```
(+) 1 2
```

- Functions can be use as operators.

```
5 `mod` 3
```

- This change affects also the priority!
  - We can define the priority of an operator created from a function.

```
infixl 7 `mod`
```

Standard operators

<table>
<tr><th>Precedence</th><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td>9</td><td><code>.</code></td><td>Function composition</td><td>Right</td></tr>
<tr><td>8</td><td><code>^,^^,**</code></td><td>Power</td><td>Right</td></tr>
<tr><td>7</td><td><code>*,/,`quot`,`rem`,`div`,`mod`</code></td><td></td><td>Left</td></tr>
<tr><td>6</td><td><code>+,-</code></td><td></td><td>Left</td></tr>
<tr><td>5</td><td><code>:</code></td><td>Append to list</td><td>Right</td></tr>
<tr><td>4</td><td><code>==,/=,<,<=,>=,></code></td><td>Compare-operators</td><td></td></tr>
<tr><td>3</td><td><code>&&</code></td><td>Logical AND</td><td>Right</td></tr>
<tr><td>2</td><td><code>||</code></td><td>Logical OR</td><td>Right</td></tr>
<tr><td>1</td><td><code>>>,>>=,=<<</code></td><td></td><td></td></tr>
<tr><td>0</td><td><code>$,$!,`seq`</code></td><td></td><td>Right</td></tr>
</table>

Creating new operators

- Operators are defined similarly to functions.

```
(&&&) :: Int -> Int-> Int
x &&& y = x + y
```

- We can change the precedence and associativity.

```
infixl 6 &&&
```

- Associativity can be changed by: `infix`, `infixl`, `infixr`
  - Keyword `infix` defines *no associativity*. We need explicit parenthesis.

Numeric lists

- `[m..n]`

  `[1..5] -- [1,2,3,4,5]`

- `[m1,m2..n]`

  `[1,3..10] -- [1,3,5,7,9]`

- Never-ending list - `[m..]`

  `[1..] -- [1,2,3,4,5,...]`

- `[m1,m2..]`

  `[5,10..] -- [5,10,15,20,25,...]`

List comprehension

- Consider mathematical definition → Define a set containing even natural numbers smaller then or equal to 10.

```
[n | n <- [1..10], n `mod` 2 == 0]
```

- Examples

```
[x*2 | x <- [1..10]]  -- [2,4,6,8,10,12,14,16,18,20]
[x*2 | x <- [1..10], x*2 >= 12] -- [12,14,16,18,20]
[ x*y | x <- [2,5,10], y <- [8,10,11]] -- [16,20,22,40,50,55,80,100,110]
allEven xs = xs == [x | x<-xs, isEven x]
allOdd xs = xs == [x | x<-xs, not(isEven x]
length' xs = sum [1 | _ <- xs]
```

Never-ending (infinite) lists

- Can not *show* the list `[1..]` but we can still use it *(lazy evaluation)*.
- Consider following function **zip**.

```
zip :: [a] -> [b] -> [(a, b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

zip [1,2,3] "ABCD" -- [(1,'A'),(2,'B'),(3,'C')]
zip [1..] "ABCD" -- [(1,'A'),(2,'B'),(3,'C'),(4,'D')]
```

Practical demonstration

- Defining new operators.
- List comprehensions:
  - it can simplify the solution;
  - nice examples of its usages.
- Nice examples of usages of infinite lists. `--Are they even USEFUL?`
- Lambda expressions.

Basic functions for lists I

- Accessing list elements

```
head [5,4,3,2,1] -- 5
tail [5,4,3,2,1] -- [4,3,2,1]
last [5,4,3,2,1] -- 1
init [5,4,3,2,1] -- [5,4,3,2]
[1,2,3] !! 2  -- 3
length [5,4,3,2,1] -- 5
null [1,2,3] -- False
null [] -- True
```

- Merging lists

Basic functions for lists II

```
[1,2,3] ++ [4,5] -- [1,2,3,4,5]
concat [[1,2],[3],[4,5]] -- [1,2,3,4,5]
zip [1,2] [3,4,5] -- [(1,3),(2,4)]
zipWith (+) [1,2] [3,4] -- [4,6]
```

- Computing with lists

```
minimum [8,4,2,1,5,6] -- 1
maximum [1,9,2,3,4] -- 9
sum [5,2,1,6,3,2,5,7] -- 31
product [6,2,1,2] -- 24
```

- Taking a part of a list

Basic functions for lists III

```
take 3 [5,4,3,2,1] -- [5,4,3]
drop 3 [8,4,2,1,5,6] -- [1,5,6]
takeWhile (> 0) [1,3,0,4] -- [1,3]
dropWhile (> 0) [1,3,0,4] -- [0,4]
filter (> 0) [1,3,0,2,-1] -- [1,3,2]
```

- Transforming a list

```
reverse [5,4,3,2,1] -- [1,2,3,4,5]
map (*2) [1,2,3] -- [2,4,6]
```

- Selected *nice* functions

```
4 `elem` [3,4,5,6] -- True
replicate 3 10 -- [10,10,10]
-- cycle and repeat returns infinite  list
take 10 (cycle [1,2,3]) -- [1,2,3,1,2,3,1,2,3,1]
take 10 (repeat 5) -- [5,5,5,5,5,5,5,5,5,5]
```

Folding a list I

- In general, folding functions transform `Foldable` structure to a *value*.
- `Foldable` structure can be for example a tree → for such a structure we need to define how to traverse it.
- We will be using it only for lists → lists are *foldable* structures.

```
class Foldable (t :: * -> *) where
  foldl :: (b -> a -> b) -> b -> t a -> b
  foldr :: (a -> b -> b) -> b -> t a -> b
  foldr1 :: (a -> a -> a) -> t a -> a
  foldl1 :: (a -> a -> a) -> t a -> a
```

- Examples

Folding a list II

```
sum' :: (Num a) => [a] -> a
sum' x = foldl (+) 0 x

product' :: (Num a) => [a] -> a
product' x = foldr1 (*) x
```

- Functions `scanl`, `scanr`, `scanl1`, `scanr1` are like their *fold* counterparts, only they report all the intermediate accumulator states in the form of a list.

```
scanl (+) 0 [3,5,2,1] -- [0,3,8,10,11]
scanr (+) 0 [3,5,2,1] -- [11,8,3,1,0]
```

Lambda expressions

- Lambdas are basically anonymous functions.
  - They are used only once → so they do not need even a name.
- Syntax

```
\x y -> x + y
```

- Examples

```
map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)] -- [3,8,9,8,7]

reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc) False ys
```

Function application with $

- Definition:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

- Differences with function application.
  - Function application is left-associative (`((f a) b) c`), $ right-associative.
  - Function application have has a highest precedence, $ has the lowest precedence.
- Why it is useful? → It eliminates many parentheses.

```
sum (map sqrt [1..130])  = sum $ map sqrt [1..130]
sqrt (3 + 4 + 9) = sqrt $ 3 + 4 + 9
```

- It also means, that function application can be treated just like another function!

```
ghci> map ($ 3) [(4+), (10*), sqrt]
```

Function composition

- Definition:

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

- The meaning is the same as in math - compose a function that takes the input, applies g and then on the result f.
- It is right-associative and it has a high precedence.

```
(\x -> negate (abs x)) = (negate . abs)
fn = ceiling . negate . tan . cos . max 50
```

User defined data types - introduction

- Type synonyms (preserve type classes)

```haskell
type String = [Char]
type Table a = [(String, a)]
type AssocList k v = [(k,v)]
```

- New (algebraic) data type

```haskell
data Bool = False | True
data Color  = Black | White | Red

isBlack :: Color -> Bool
isBlack Black = True
isBlack _ = False
```

  - Color – type constructor
  - Red / Green / Blue – data (nullary) constructor

User defined data types - more advanced I

- Parametric data types

```
data Point = Point Float Float
```

- Data (Value) constructor's type

```
ghci> :t Point
Point :: Float -> Float -> Point
```

- Usage

```
dist (Point x1 y1) (Point x2 y2) = sqrt ((x2-x1)**2 + (y2-y1)**2)
```

```
ghci> dist (Point 1.0 2.0) (Point 4.0 5.0) = 5.0
```

- Polymorphic data types

User defined data types - more advanced II

```
data Point a = Point a a
```

- Constructor: `Point :: a -> a -> Point a`
- Better examples (build in types)

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)

sqrt' :: Float -> Maybe Float
sqrt' x | x < 0       = Nothing
        | otherwise = Just (sqrt x)
```

Recursive data types

- We already know recursive data type - List

```
data List a = Null
            | Cons a (List a)

lst :: List Int
lst = Cons 1 (Cons 2 (Cons 3 Null))
```

- Better example - binary tree

```
data Tree1 a = Leaf a | Branch (Tree1 a) (Tree1 a)
data Tree2 a = Leaf a | Branch a (Tree2 a) (Tree2 a)
data Tree3 a = Null | Branch a (Tree3 a) (Tree3 a)

t2l :: Tree1 a -> [a]
t2l (Leaf x) = [x]
t2l (Branch lt rt) = (t2l lt) ++ (t2l rt)
```

Automatically deriving type classes

- Consider following example:

```
data Color = Black | White
list :: [Color]
list = [Black, Black, White]
```

```
ghci> list
<interactive>:15:1: error:
    * No instance for (Show Color) arising from a use of `print'
    * In a stmt of an interactive GHCi command: print it
```

- A solution can be let Haskell automatically derive type classes.

```
data Color = Black | White deriving (Show, Eq, Ord, Read)
```

```
ghci> list
[Black,Black,White]
```

Record syntax

- Named fields in a data type definition.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Show)
```

```
ghci> :t firstName
firstName :: Person -> String
```

```
description :: Person -> String
description p = firstName p ++ " " ++ lastName p
```

```
description Person {firstName = "John" , lastName="Doe", age = 40}
```

Practical demonstration

- New user defined data types.
  - Algebraic types
  - Parametric data types
  - Polymorphic data types
  - Simple recursive data structures (list).
- Data structures handling frequently encountered problems.
  - Maybe
  - Either
  - Tree - Expressions
- Record syntax

Modules 101

- Module definition
  - All definitions are visible.

    ```
    module A where   -- A.hs, A.lhs
    ```
  - Restricted export

    ```
    module Expr ( printExpr, Expr(..) ) where
    ```
  - Data types restrictions

    ```
    Expr(..) -- exports also constructors
    Expr -- exports data type name only
    ```
- Restricted import

  ```
  import Expr hiding( printExpr )
  import qualified Expr -- Expr.printExpr
  import Expr as Expression -- Expression.printExpr
  ```

Type classes 102 I

- Type class definition

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

  - Default definitions are overridden by instance definition.
  - At least one must be defined.

- Adding a type into a class.

```
instance Eq Color where
    Black == Black = True
    White == White = True
    _ == _ = False
```

Type classes 102 II

- Adding a data type with parameters.

```haskell
instance (Eq a) => Eq (Maybe a) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

- There can be relations between data type classes - class `Ord` inherits the operations from class `Eq`.

```haskell
class Eq a => Ord a where
        (<),  (<=), (>), (>=) :: a -> a -> Bool
        max, min :: a -> a -> a
        compare        :: a -> a -> Ordering
```

Example of a user defined type class I

```haskell
class Visible a where
        toString :: a -> String
        size :: a -> Int
instance Visible Char where
        toString ch        = [ch]
        size _ = 1
instance Visible Bool where
        toString True = "True"
        toString False = "False"
        size = length . toString
```

Example of a user defined type class II

```
instance Visible a => Visible [a] where
        toString            = concat . map toString
        size            = foldr (+) 0 . map size

class (Ord a, Visible a) => OrdVisible a where
    ...
```

Abstract data type

- Definition: An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects.
- Imperative style
  - Class of objects whose logical behavior is defined by a set of values and a set of operations.
  - ADT is a *mutable* structure (mutable structure has inner state, its behaviour can change in time).
- Functional style
  - Each state of the structure as a separate entity.
  - ADT is modeled as a mathematical function that takes the old state as an argument, and returns the new state as part of the result.
  - Unlike the imperative operations, these functions have no side effects.

Example of queue usage in Java

```java
public class QueueExample
{
  public static void main(String[] args)
  {
    Queue<Integer> q = new LinkedList<>();

    for (int i=0; i<5; i++)
     q.add(i);

    int remove = q.remove();
    int head = q.peek();
    int size = q.size();
  }
}
```

Queue implementation in Haskell I

- Initialization: `emptyQ :: Queue a`
- Test if queue is empty: `isEmptyQ :: Queue a -> Bool`
- Inserting new element at the end: `addQ :: a -> Queue a -> Queue a`
- Removing element from the beginning: `remQ :: Queue q -> (a, Queue a)`

```haskell
module Queue(Queue, emptyQ, isEmptyQ, addQ, remQ) where
    data Queue a = Qu [a]

    emptyQ :: Queue a
    emptyQ = Qu []

    isEmptyQ :: Queue a -> Bool
    isEmptyQ (Qu q) = null q
```

Queue implementation in Haskell II

```haskell
addQ :: a -> Queue a -> Queue a
addQ x (Qu xs) = Qu (xs++[x])

remQ :: Queue a -> (a,Queue a)
remQ q@(Qu xs) | not (isEmptyQ q) = (head xs, Qu (tail xs))
               | otherwise        = error "remQ"
```

Practical demonstration

- Functional style for handling data:
  - defining new data types;
  - how to handle these new data;
  - modules. `--Just to avoid long files?`

Thank you for your attention

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

September 26, 2023

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE