Functional programming languages

Functional programming languages

- **FP program – set of "pure" functions composed from expressions**
  - Principle of **referential transparency**
    - **Expression/function has always the same value for the same value of its arguments, independent on context in which expression/function is evaluated**
  - Function – **expression is assigned to the name** of function for some input parameters
  - Function gets a value when it is invoked by some concrete values of parameters, **no side-effects**
  - **Expression is application of a function** or operator on some arguments
    - Arguments can be expressions → make function compositions, recursive functions

Functional programing languages

- **Abstraction of flow of execution**
  - **No commands and variables**
    - Immutable function parameters
    - Immutable local variables
  - **Built-in mechanisms of expression evaluation, no need to know how it functions**
  - **Conditional expression** – expression value depends on value of some other sub-expression
  - **Recursion** instead of loops
- **Evaluation of FP program starts with a function application on concrete values of arguments**

Characteristics of Functional PL

- **FP abstracts the flow of program execution**
  - **Shorter and more concise programs** comparing to imperative programming
  - **Higher degree of abstraction** → smaller number of details → smaller possibility to make errors

- **Referential transparency of functions**
  - Smaller possibility to make errors
    - No side effects
  - **Better formal analysis** and validation of programs
  - **Greater possibility for program parallelization**
    - Subexpressions which are arguments of some other expression can be evaluated in parallel.

Higher-Order Functions

- **Higher-order functions can have functions as arguments, or their results are functions or both**
  - Example: derivation, integral
- Example.

```
function inc(x) = x + 1
function twice(f, x) = f(f(x))
twice(inc, 5)  → 7
```

- Three typical higher-order functions
  - **map f l** – apply function f on each element of its argument which is list l
  - **filter f l** – filter list l based on logical function f
  - **fold f l n** – reduces list l according to operator (binary function) f, n is neutral element of operator f
- Functions as elements of a data structure

Strict and ne-strict semantics

- **Strict semantics**
  - Expression (function) **can be evaluated in some value only if all its subexpressions (arguments) can be evaluated** in some values
  - ***Strict/eager evaluation, call by value***: expression value (function) can be evaluated after all its subexpressions (arguments) are evaluated
  - **Imperative programming languages are based on strict semantics, excluding logical expressions**
- **Non-strict semantics**
  - Expression (function) **can be evaluated even if some its subexpressions can not be evaluated**
  - ***Non-strict (lazy) evaluation, call by need***: Expression (function) is evaluated only if its value is needed
  - **Lazy FP languages: FP languages that support non-strict semantics (Miranda, Haskell)**

Strict and non-strict semantics

- **Examples.**
  - (x = 0) or (1 / x = 5)
    - for x = 0 expression has no value in strict semantics
    - In non-strict semantics it has value true
  - length [2, 2 + 4, 6 / 0, 2 + 3 * 4]
    - in strict semantics function can not be evaluated as third expression can not be evaluated
    - In non-strict semantics elements of list are not evaluated, as function returns length of the list
  - function sqr(x) = x * x, evaluate sqr(2 + 3)
    - **Eager evaluation.** sqr(2 + 3) → sqr(5) → 5 * 5 → 25
    - **Lazy evaluation.** sqr(2 + 3) → (2 + 3) * (2 + 3) → 5 * 5 → 25

Infinite Data Structures

- **Non-strict semantics offer possibility to work with infinite data structures**
- **Example**. An infinite list of 1s can be defined as an infinitely recursive function without arguments

  function Ones = 1 : Ones

  - Operator : (cons) – *x : y* form the list with head *x*, and tail *y*
  - Ones → 1 : Ones → 1 : 1 : Ones → …
- **function Head(h : t) = h**
- Eager evaluation

  Head(Ones) → Head(1 : Ones) → Head(1 : 1 : Ones)
  → Head(1 : 1 : 1 : Ones) →Head (1 : 1 : 1 : 1 : Ones) → …
- Lazy evaluation

  Head(Ones) → Head(1 : Ones) → 1

Lambda calculus

- Theory of functions proposed by *Alonzo Church* 30es of 20 century
- Lambda calculation is transformation of **lambda expression** using rules of lambda calculus
  - lambda expression is an **identifier**
  - If ***x is* identifier, *e* and *n* lambda expressions** then following are also lambda expressions
    - λx.e lambda abstraction
    - e n application (apply e on argument n)
- **Lambda abstraction is concept of anonymized function in FL**
  - `λx.x + 1`
  - `(λx.x + 1) 4` → `5`
  - `λx y.2x + y`
  - `(λx y.2x + y) 3 4` → `10`

Anonymized functions

- Often used as **parameters of higher-order functions**
- **Higher-order functions that return function as their value always return anonymized function**
- Without anonymized function

```
function inc(x) = x + 1
function twice(f, x) = f(f(x))
```

`twice(inc, 5)` → `7`

With anonymized function

```
function twice(f, x) = f(f(x))
```

`twice(λx.x + 1, 5)` → `7`

- **Example of function which returns function as its value:**

```
function incrementBy(x) = λy.y + x
```

Curry Functions

- **Currying: definition of function with *n* arguments as *n* nested functions with one argument (Haskell Curry)**

orginal function $\lambda x_1\ x_2\ \ldots\ x_n.e$

Curry function $\lambda x_1.(\lambda x_2.(\lambda x_3\ \ldots\ (\lambda x_n.e)))\ \ldots)$

- **Examples of Currying.**

```
function add(x, y) = x + y
function addCurry(x) = λy.x + y

addCurry(5)        → λy.5 + y
addCurry(5)(10)  → (λy.5 + y) 10 → 15
```

Partial function application

- Let *f* is function with *k* argumenats

- **Partial application of function *f* is application of function *f* with less than *k* argumens**

- **Example.**

```
function add(x, y, z) = x + y + z
add(1, 2, 3) → 6
add(1, 2)    → λz.3 + z
add(1)       → λy z.1 + y + z
```

- **Partial application ≡ currying, evaluation, de-Currying**

LISP (List Processing)

- First FP language has been developed in 60es, John McCarthy
- Only **one type for everything** – all data are s-expressions (symbolic expressions)
  - S - constants and numbers are s-expressions
  - If A and B are s-expressions then **(A . B)** is s-expressions - **pair**
  - If x_1 x_2 ... x_n s-expressions then **(x_1 x_2 ... x_n)** is s-expressions - **list**. () je empty list
    - List is sequence of nested pairs
      (1 2 3 4) ≡ (1 . (2 . (3 . (4 . ()))))
- **The same notation for data and functions/programs** – function definition and application are also s-expressions
  - (define (functionName arg1 arg2 ... argn) expression) ≡ definition f
  - (functionName arg1 arg2 ... argn) ≡ application f

LISP (List Processing)

- **Everything is s-expression**
  - **Built-in functions for checking types of s-expressions**: if s-expression is constant or number or pair or list or empty list,…
  - Quote (‘) function
    - ‘(+ 1 2) – it is s-expressions i.e. list with 3 elements
    - (+ 1 2) – s-expressions evaluated in 3 (application of function +)

- **Conditional expression**
  - **(if c e1 e2)** -- if c is true then value of whole expression is the same as value of e1, if c is false then value of whole expression is the same as value of e2
  - **If expression represents value, contrary to if command**
    (+ 5 (if (> 4 5) 1 2)) → (+ 5 (if false 1 2)) → (+ 5 2) → 7

```
(define (fibonacci n)
  (if (< n 2) n
      (+ (fibonacci (- n 1))
         (fibonacci (- n 2)))))
```

Successors of LISP

- ISWIM (if you see what I mean), Landin, ~1960
  - Infix notation instead of prefix notation for arithmetic-logic expressions
  - Constructions *let and where* local variables binding
  - SECD machine
- FP, Backus, ~1970
  - Functional programming as a composition of higher-order functions
- ML, Milner, ~1970
  - Parametric polymorphism, *type inference*
- SASL, KRC & Miranda, Turner
  - Lazy evaluation, ZF expressions for lists forming, Function definition as separate cases (sequence of equations) and *pattern matching*, *guard* expressions
- Haskell, 1987, international committee
  - "*Grand unification of functional languages*", *type classes*, *monads*

Haskell

- basic elements of language -

Haskell

- “Pure” functional programming language
- Haskell B. Curry (1900 – 1982), mathematician
- Basic language characteristics
  - **Lazy evaluation of expressions, non-strict semantics and infinite data structures**
  - Static type checking, *type inference* mechanism
  - **User defined types and parametric polymorphism** (generic types)
  - **Function** definition as ***cases* and *pattern matching***
  - **ZF expressions** and list forming
  - *type classes* and *type-safe* ad-hock polymorphism (*operator overloading*)

GHC (*Glasgow Haskell Compiler*) and Haskell platform

- GHC leading (*open source*) language implementation, part of Haskell platform
- https://www.haskell.org/platform/

- **GHC has compiler and interpreter for Haskell**

Types

- Each well defined expression in Haskell has a type
- **e :: t** – means that expression *e* can be evaluated in a value of type *t*
- Types are **determined automatically during compilation time**
- :t (:type) command determines expression type without its evaluation

```
> 2 < 5
True
> :t 2 > 5
2 > 5 :: Bool
> "Ana" ++ " voli" ++ " Milovana"
"Ana voli Milovana"
> :t "Ana" ++ " voli" ++ " Milovana"
"Ana" ++ " voli" ++ " Milovana" :: [Char]
```

Basic types in Haskell

- **Bool**
  - Logical value True i False
- **Char**
  - Characters: ‘a’, ‘b’, ‘c’,…
- **String ([Char])**
  - Strings realized as a list of characters (“a”, “Mika”, “Pera”, “Zika”, …)
- **Int**
  - Integers of fixed precision (30 bits, interval [-2^{29} .. 2^{29}-1])
- **Integer**
  - Integers of arbitrary precision (represented as a list of digits)
- **Float, Double**
  - Real numbers

Tuples

- *N*-tuple is sequence of *N* values that **can be of different types**
- (t_1, t_2, …, t_N) je tip *N*-tuple, types of components are t_1 to t_N
  - (False, True) :: (Bool, Bool)
  - (False, 1, ‘x’, True) :: (Bool, Int, Char, Bool)
- Number of components is determined by length of *N*-tuple
  - (Bool, Int) is 2-tuple (**pair**)
  - (Char, Int, Bool) is 3- tuple (**triplet**)
- Example
  - (1, (1, ‘x’), True, 5) :: (Bool, (Int, Char), Bool, Int)

Built-in functions on pairs

- **fst** – returns the **first component** of pair
  - `fst (1, 2)` → `1`
- **snd** – returns the **second component** of pair
  - `snd (1, 2)` → `2`

```
-- effects of  fst i snd
myfst (x, _) = x
mysnd (_, x) = x

-- extractions of triplet components
fst3 (x, _, _) = x
snd3 (_, x, _) = x
thr3 (_, _, x) = x
```

Lists

- List is a sequence of values that **must be of the same type**
- List can have arbitrary number of elements
- **[*t*] is type of the list which elements are of type *t***
  - [False, True, False, False] :: [Bool]
  - [1, 2, 9, 10] :: [Int]
  - [[1, 2], [1, 2, 4], [3, 4, 6, 1, 4]] :: [[Int]]
  - [(1, False, 3), (2, True, 6), (4, False, 4)] :: [(Int, Bool, Int)]
- [] is empty list

Built-in functions on lists

- **head – returns the first element**
  - head [1, 2, 3, 4, 5] → 1
  - head [1] → 1
  - head [] → exception
- **tail – returns the tail of the list**
  - tail [1, 2, 3, 4, 5] → [2, 3, 4, 5]
  - tail [1] → []
  - Tail [] → exception
- **!! – selects *k*-th element of list (indexing starts from 0)**
  - [10, 20, 30, 40] !! 0 → 10
  - [10, 20, 30, 40] !! 2 → 30

Built-in functions on lists

- **take – selects the first *k* elements**
  - take 3 [10, 20, 30, 40, 50] → [10, 20, 30]
- **drop – “removes” the first *k* elements**
  - drop 3 [10, 20, 30, 40, 50] → [40, 50]
- **length – returns the length of list**
  - length [10, 20, 30, 40, 50] → 5; length [] → 0
- **null – check is list empty**
  - null [10, 20, 30, 40, 50] → False; null [] → True
- **: (cons operator) – adds new element at the beginning of list**
  - 10 : [20, 30, 40] → [10, 20, 30, 40]
- **++ (append operator) – appends 2 lists**
  - [10, 20] ++ [30, 40, 50] → [10, 20, 30, 40, 50]

Basic types of Expressions

- **Arithmetical expression – composed form arithmetic operators, evaluated in some of numerical types**
  - +, -, *, /, `div`, `mod`
  - 2 + 3 * 4, (2 + 3) * 4
  - 1 / 4 (result is real number), 1 `div` 4 (integer division)
- **Logical expressions – composed from logical operators, evaluated in some of logical values**
  - && (conjunction), || (disjunction), not
- **Relational expressions – composed form relational operators, evaluated in some of of logical values**
  - <, >, <=, >=, == (equal), /= (non-equal)
  - (2 < 5 && "Ana" /= "Mina") || not (3 == 4)

Conditional Expression

- Conditional expression: **if e then p else q**
  - **“else branch” in conditional expression in Haskell is obligatory**
  - If value of e is true Then value of whole expression is equal to value of expression p, otherwise it has value of q
  - Expressions p and q must be evaluated in the same type

- **Examples:**
  - `if n > 0 then n else -n`
  - `if a > b then a else b`
  - `5 + if a > b then a else b`
  - `(if a > b then a else b) + 5`
  - `if mod x 2 == 0 then 2 * x else x`

Nested Conditional Expressions

- Conditional expression: **if e then p else q**
  - **p and/or q also can be conditional expressions**

- ```
  if a > b
    then if a > c then a else c
    else if b > c then b else c
  ```

- ```
  if c >= 'a' && c <= 'z`
    then “Small letter“
    else if c >= 'A' && c <= 'Z`
           then “Capital letter“
           else “Not eng. alphabet letter "
  ```

Application of function on arguments

- Let *f* is function with *k* argumenats
- $f\ a_1\ a_2\ a_3\ \ldots\ a_k$ – application of function *f* on arguments a_1 to a_k
- **Examples of built-in numerical functions.**
  - `truncate 12.78` evaluates in 12
  - `round 12.78` evaluates in 13
  - `gcd 75 100` evaluates in 25
- **Application of function on argumenta has higher priority than infix operators**
  - `f a + b is (f a) + b`, and not `f(a + b)`
  - `f a b + c d` is `(f a b) + (c d)`
- `` x `f` y `` is “syntactical sugar” for `f x y`

Application of function on arguments

Mathematics	Haskell
`f(x)`	`f x`
`f(x,y)`	`f x y`
`f(g(x))`	`f (g x)`
`f(x,g(y))`	`f x (g y)`
`f(x)g(y)`	`f x * g y`

Function Definition

- Function is defined by specifying **one or more declarations ("equations")** of the form **fname args = expr**
- **Names** of function and arguments **begins with a small letter**
- **For functions defined by different cases (multiple declarations): during execution system tries to find the first declaration that can be unified (matched) with arguments in function call** (***pattern matching***)
- To define **new functions** programmer can **use built-in functions or previously user-defined functions**
- **Examples.**
  - `double x = 2 * x`
  - `doubleEven x = if mod x 2 == 0 then 2 * x else x`
  - `maks2 a b = if a > b then a else b`
  - `maks3 a b c = maks2 (maks2 a b) c`

Examples of recursive functions - different cases

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

-- function to evaluate  k-th Fibonacci
number
fib 1 = 1
fib 2 = 1
fib n = fib (n - 1) + fib (n - 2)

-- function to evaluate a^k za k >= 0
step _ 0 = 1
step a 1 = a
step a k = a * step a (k - 1)
```

- ***k* - an argument in function definition is a pattern which can be matched with anything, *k* is binding to the argument from function call**
- **_ is pattern which can be matched with anything without binding**
  - pattern for function parameters that we do not use efectivelly

Tail-recursion

- **Function is tail-recursive if its evaluation is finished by recursive call (except of trivial cases)**
- Recursive functions with accumulated parameters are **tail-recursive**

```
-- non-tail-recursive function
fact 0 = 1
fact n = n * fact (n - 1)

-- function factAcc is tail-recursive
fact' n = factAcc n 1
factAcc 0 acc = acc
factAcc n acc = factAcc (n - 1) (n * acc)
```

Accumulation Parameter Technique

- Accumulating parameter technique offers possibility to write more efficient functions than just following its definitions
- **Example.** Fibonacci numbers

```
-- Fibonacci numbers by accumulating parameters technique
fib' 1 = 1
fib' 2 = 1
fib' n = fibAkum 1 1 2 n

{-
   function fibAkum is tail-recursive and of linear
complexity
 -}
fibAkum f1 f2 cnt n =
    if n == cnt
        then f2
        else fibAkum f2 (f1 + f2) (cnt + 1) n
```

Function Type

- By function type we **specify types of arguments and return values of the function**
- Function type **need not be explicitly specified** as we define a function, it will be determined by Haskell built-in mechanism.
- ***f :: x -> y –*** it is the function type of one argument, it maps elements of type ***x*** in elements of type ***y***
- Names of types always begins with a capital letter

```
-- function type
fib :: Int -> Int

-- function definition
fib 1 = 1
fib 2 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Function Type

- **All Haskell functions are Curry functions i.e. with one argument, they are Currying implicitly, can be partial applied**
  - Function type with two arguments a -> (b -> c)
  - Function type with three arguments a -> (b -> (c -> d))
- **Separator -> is right associative so parentheses can be deleted**
  - Function type with two arguments: a -> b -> c
  - Function type with three arguments: a -> b -> c -> d
  - …
  - Function type with k arguments: $x_1 \text{ -> } x_2 \text{ -> } x_3 \text{ -> } \ldots \text{ -> } x_k \text{ -> } y$
- Application of function on arguments:

  $f\ a_1\ a_2\ a_3 \ldots a_k \equiv (\ldots((f\ a_1)\ a_2)\ a_3) \ldots a_k)$

```
-- function that evaluates a^k za k >= 0
step :: Int -> Int -> Int
step _ 0 = 1
step a 1 = a
step a k = a * step a (k - 1)

-- function that multiplies three
numbers
mul :: Int -> Int -> Int -> Int
mul x y z = x * y * z

-- function of one argument realised
-- by partial application of function
mul
mul_3_5 :: Int -> Int
mul_3_5 = mul 3 5

> mul_3_5 10
150
```

Guarded Expressions (*guards*)

- Function can be defined using guarded expressions i.e. guards

$$\begin{aligned}\texttt{fname args } &| \; g_1 = expr_1 \\ &| \; g_2 = expr_2 \\ &\dots \\ &| \; g_k = expr_k \\ &| \; \texttt{otherwise} = expr_o\end{aligned}$$

```haskell
desc c
  | c >= 'a'  && c <= 'z'  = "Small letter"
  | c >= 'A'  && c <= 'Z'  = "Capital letter"
  | c >= '0'  && c <= '9'  = "Digit"
  | otherwise = "Special character"

char1 n
  | n < 0 = "negative"
  | n == 0 = "zero"
  | otherwise = "positive"
```

Let Expressions

- Let expression allows evaluation of an expression in **extended local environments**
- `let <bindings> in expr`
-
```
let
    v1 [args] = expr1        -- [] optional
    …
    vk [args] = exprk
in expr
```
- Identifiers **v_1, v_2, …, v_k** are variables or functions
- In expression **expr** identifiers **v_1, v_2, …, v_k** can appear, their values are obtained by evaluation of corresponding expressions
- Identifiers **v_1, v_2, …, v_k** are local, not visible outside the let block
- Values of let expression is the same as value of expression **expr**

Let Expressions

- ```
  let
      v1 [args] = expr1
      v2 [args] = expr2

      …

      vk [args] = exprk
  in expr
  ```
- In this case **v_1, v_2, …, v_k** **must be identically indented**...
- ... Or explicitly grouped
- ```
  let {
      v1 [args] = expr1; -- now indentation is not important
      v2 [args] = expr2;

      …

      vk [args] = exprk;
  } in expr
  ```

Let Expressions

```
foo a b c d =
  let
    y = a * b
    f x = (x + y) / y
    fact x = if x == 0 then 1 else x * fact (x - 1)
    z = a * k
    k = b
  in f c + fact d + z
```

- **Notice:**
  - In the definition of identifier f, identifier y is used, it is defined in let block previously
  - fact is recursive function
  - In the definition of identifier z identifier, k is used, it will be later definide in let block
- **For definition some identifiers in let block we can use all identifiers from let block + recursive definitions are allowed**

Let Expressions

```
cylinderSurface r h =
  let
    base = r * r * 3.14
    wrapper= 2 * r * 3.14 * h
  in 2 * base + wrapper

-- Function that returns pair of two
numbers
quadraticEquation a b c =
  let
    t = b * b - 4 * a * c
    s = sqrt t
    f1 = (-b + s) / (2 * a)
    f2 = (-b - s) / (2 * a)
  in (f1, f2)
```

Let Expressions

```
prime 2 = True
prime x =
  let
    -- operator . is composition of functions
    -- fromIntegral konverts Int in wider class
    -- of numbers Num
    -- it is necessary to apply sqrt function
    limit = (round . sqrt . fromIntegral) x
    noDivisors d numb
      | numb > limit = True
      | d `mod` numb == 0 = False
      | otherwise          = noDivisors d (numb + 1)
  in
    noDivisors x 2
```

Where block

- Where block also allows evaluation of an expression in environment extended by different definitions (identifiers, functions)
- **Differences between where and let block**
  - new identifiers and assigned expressions are given after the “main” expression/function
  - ***where* is not expression** but syntactical construction (has no value)
  - *where* can be introduced **after guarded expressions**

```
bar a b c d =
  f c + fact d + z
  where
    y = a * b
    f x = (x + y) / y
    fact x = if x == 0 then 1 else x * fact (x - 1)
    z = a * k
    k = b
```

Where block

- Example of where block introduced after guarded expression

```
-- weight in kg, hight in m
-- operator ^
-- take care of identation!
bodyMassIndex weight height
    | bmi <= skinny = "low"
    | bmi <= normal = "normal"
    | bmi <= fat    = "high"
    | otherwise     = "very high"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

Case expression

- **Conditional expression based on *pattern matching***

  case expr of

  pattern$_1$ → expr$_1$

  …

  pattern$_k$ → expr$_k$

  - Value of expression is **expr$_p$** where **pattern$_p$** is the first pattern which can be matched with **expr**

```
-- take care of identation!
fib' k =
  case k of
    1 -> 1
    2 -> 1
    _ -> fib' (k - 1) + fib' (k - 2)
```

Operators

- Operators are binary functions that can be applied in infix and prefix forms
  - For operator **$** appropriate function is written as **($)**
  - **2 + 3 (infix)** is equivalent with **(+) 2 3 (prefix)**

```
-- definition of operator as function
(\+) :: Bool -> Bool -> Bool
(\+) False b = b
(\+) True _ = True

-- pattern matching definition of operator
(\*) :: Bool -> Bool -> Bool
True \* b = b
False \* _ = False

twoDigits x = ((x >= 10)  \* (x < 100)) \+
              ((x <= -10) \* (x >= -100))
```

Anonymous functions

- Anonymous functions in Haskell are defined as follows.
  \args -> expr
  - `\x -> x + 1`
  - `(\x -> x + 1) 5` is evaluated in 6
  - `\x y -> x + y`
  - `(\x y -> x + y) 5 6` is evaluated in 11
  - `mult3 = \x y z -> x * y * z`
  - `mult3 1 2 3` is evaluated in 6

Partially applied operators

- Operators are binary functions and can be partially applied

Partial application	Result
`(+ 1)`	`\x -> x + 1`
`(1 +)`	`\x -> 1 + x`
`(== 5)`	`\x -> x == 5`
`(5 ==)`	`\x -> 5 == x`
`(/= 5)`	`\x -> x /= 5`
`(> 5)`	`\x -> x > 5`
`(5 >)`	`\x -> 5 > x`

Higher-order Functions

- **Higher-order functions can have functions as arguments, or their results are functions or both**

```
-- higher-order function
apply2 f x = f (f x)

> apply2 succ 5
7
> apply2 (+3) 10
16
> apply2 reverse "Ana voli Milovana"
"Ana voli Milovana"
> apply2 (++ " kul") "Haskell"
"Haskell kul kul"
> apply2 (10:) [1, 2, 3, 4]
[10,10,1,2,3,4]
```

Map Function

- Return as result list applying function (given as first argument) on each element of list given as the second argument

```
> map (+1) [1, 2, 3, 4]
[2,3,4,5]

> map (==1) [1, 2, 3, 4]
[True,False,False,False]

> map length ["foo", "bar", "mika", "ab"]
[3,3,4,2]

> map (\x -> x + 3) [1, 2, 3, 4]
[4,5,6,7]
```

Type classes

- **Type class is collection of types which perform operations adequate for that Type class**
- **Type can belong to one or more type classes**, in this case it can apply all operations adequate for these type classes
- Type classes allows **ad-hock polymorphism** (*operator overloading*)
- Examples of built-in Type classes
  - **Eq**: Types that realize operations (==) i (/=)
  - **Ord**: Types that realize operations (<), (<=), (>) i (>=)
  - **Num**: All types that realize arithmetic operations
- **Polymorph functions can have some restrictions depending on Type class**
  - `(+) :: Num a => a -> a -> a`
  - `(^) :: (Num a, Integral b) => a -> b -> a`

Type classes

```
-- function that gives the bigger of two comparable elemenets
maks :: (Ord t) => t -> t -> t
maks a b = if a > b then a else b
{-
   maks "Ana" "Mika"
   maks 12 34
   maks 13.4 56
   maks False True
   maks 'a' 'b'
-}

-- function that adds 5 to the bigger of two comparable elemene
maksPlus5 :: (Num t, Ord t) => t -> t -> t
maksPlus5 a b = (if a > b then a else b) + 5
{-
   maksPlus5 12 34
   maksPlus5 13.4 56
-}
```

User defined type classes

```
class Negation a where
  neg :: a -> a

instance Negation Integer where
  neg k = k * (-1)

instance Negation Bool where
  neg True = False
  neg False = True

> neg (1 > 2)
True
> neg (21 + 45)
-66
```