1. Programowanie deklaratywne
W językach deklaratywnych programista **deklaruje zależności oraz cele** nie podając wprost sposobu osiągnięcia wyniku. Innymi słowy opisujemy co chcemy osiągnąć, nie opisując sposobu.

2. Programowanie funkcyjne

- Jest podparadygmatem programowania deklaratywnego. Opisujemy pożądany wynik **przy pomocy funkcji**. Kompilator wyłącznie oblicza wartość funkcji głównej.
- **Funkcje czyste** zawsze przyjmują tę samą wartość dla tych samych argumentów - nie są funkcjami w rozumieniu programowania funkcyjnego.
- W programowaniu funkcyjnym podstawowym narzędziem jest **rekurencja**, a także **rachunek lambda** (definiowanie funkcji bez nadawanie jej nazwy) czy **przezroczystość referencyjna** (funkcja, która daje wynik, może być parametrem f(a) = b, g(b,c) = g(f(a),c) = d).
- **Leniwa ewaluacja**, czyli obliczanie wartości wyrażenia dopiero gdy jest ono potrzebne (umożliwia to stworzenie nieskończonych matematycznie struktur)
- **Silne typowanie i matematyczna ścisłość** (łatwość w dowodzeniu poprawności)

3. Operatory
Zapis: mod 4 3 = 4 `mod` 3 = 1 to samo dla div
Zapis: (+) 5 2 = 5 + 2 = 7 oraz (-, *, /, ==, <, <=, >=, /=, ^, **, mod, div, ||, &&, not)
Potęga całkowita: 2.5^3 potęga ułamkowa: 2.5**3.5
Dwuargumentowe: x & y = (x + y) / 2

4. Typy podstawowe
Int - z zakresu -2^29..29^28 **Integer** - dowolna liczba całkowita
Float - zmiennoprzecinkowa pojedynczej precyzji **Double** - podwójnej precyzji
Char - pojedynczy znak ‘x’ **String [Char]** - łańcuch znaków “xxx”
Bool - typ wyliczeniowy logiczny
Ordering - Typ relacji między elementami posiadający trzy wartości (LT, EQ, GT) - compare
Num - rodzina typów liczbowych **Eq** - rodzina typów dla których zdefiniowane jest == i /=

5. Typy strukturalne
Lista - elementy tego samego typu (rozmiar nieokreślony)
[Int] = [1, 2, 3] [Char] = [‘a’, ‘b’, ‘c’] [[Int]] = [[1], [1,4], []] [(String, Bool)] = [(“Ala” True)]
Krotka - elementy różnych typów (rozmiar określony - stały)
(Int, Char) = (1, ‘a’) ((Bool, String), Int) = ((True, “Ala”), 2) ([Int], Char) = ([1, 2], ‘c’)

6. Typy funkcji
nazwa funkcji :: typArg1 -> typArg2 -> typArgN -> TypWartościZwracanej
inc :: Int -> Int

7. Typy polimorficzne (rodzina typów)

kw :: Num a => a -> a
(gdzie a to zmienna typowa, ograniczona przez “Num a =>” co oznacza klasę typów liczbowych i kilku innych, :: znaczy “jest typu”)

8. Przykłady typów i inne wyniki operacji

Operator (typy jego argumentów) :t (||) :: Bool -> Bool -> Bool
Wynik funkcji :t True && False :: Bool
Krotka (typy jej składowych) :t (True, “pf”) :: (Bool, [Char])
Funkcji (typy wejściowe i wyjściowe) :t fst :: (a, b) -> a
Liczby całkowitej :t 6 :: Num a => a
Liczby zmiennoprzecinkowej :t pi :: Floating a => a
Znaku :t ‘a’ :: Char
Stringu :t “zdanie” :: [Char]
:t (+) :: Num a => a -> a -> a typ operatora/funkcji
:t (+) 2 3 :: Num a => a typ wyniku operatora/funkcji
map (^2) [1,2,7,5] będzie [1, 4, 49, 25]
map (>2) [1, 2, 7, 5] będzie [False, False, True, True]
filter (>2) [1, 2, 7, 5] będzie [7, 5]
:t map będzie (a -> b) -> [a] -> [b] czyli (a -> b) funkcja zmieniająca a na b, [a] jakaś tablica, [b] czyli wynik tej funkcji z tą tablicą
:t filter będzie (a -> Bool) -> [a] -> [a] czyli (a -> Bool) funkcja sprawdzająca dla elementu warunek i dająca Bool, [a] - dana tablica, [a] - wynik tej funkcji na tablicy
Konkatenacja do listy 2:lista (na początek listy) - działa tylko gdy konkatenujemy do listy
Konkatenacja [Char] “ALA”++”OLA”

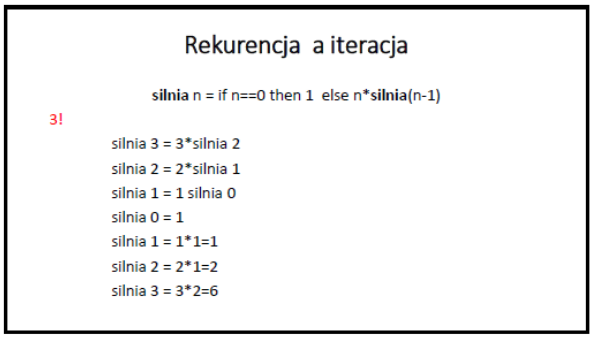
9. Sposoby definiowania funkcji

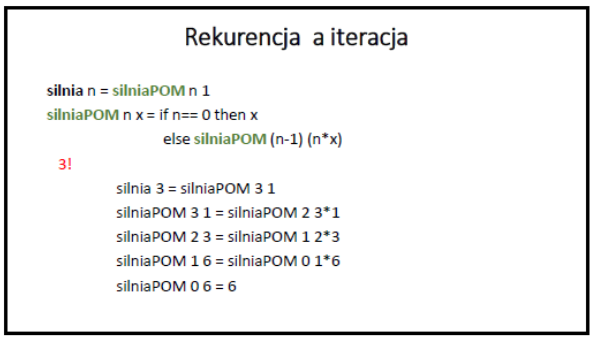
Przykłady

```
silnia1a n = if n == 0  then 1
                        else n * silnia1a (n − 1)
-----------------------------------------------------------------
silnia1b n                                              0!=1
  | n==0 = 1                                           n!=n*(n-1)!
  | otherwise = n * silnia1b (n − 1)
-----------------------------------------------------------------
silnia1c 0 = 1
silnia1c n = n * silnia1c (n - 1)
-----------------------------------------------------------------
silnia1d n = case n of  0 -> 1
                        _ -> n * silnia1d (n - 1)
```

10. Wersja akumulatorowa (rekurencja ogonowa)

Nie ma potrzeby budowania stosu, zastępuje pętle, jest wydajniejsza, akumulatory przechowują wyniki częściowe funkcji głównej, pierwszy argument pełni rolę licznika





1. Definiowanie listy

[1..5] = [1, 2, 3, 4, 5] **[1.0, 1.25.. 2.0]** = [1.0, 1.25, 1.5, 1.75, 2.0] **[1..]** = [1, 2, 3, …]
[x^2 | x<- [1 .. 5]] = x^2 : x e {1,....,5} = {1, 4, 9, 16, 25}
factors n = [x | x <- [1 .. n], mod n x == 0] **factors 20** = [1, 2, 4, 5, 10, 20]

2. Operacje na listach

:: [a] -> [a]
head (x:_) = x pierwszy element (x - pierwszy, _ - reszta elementów)
tail (_:xs) = xs ostatni element (_ - jakies elementy, xs - ostatni element)
init [x] = [] init (x:xs) = x : init xs wszystkie elementy bez ostatniego
last [x] = x last (_:xs) = last xs ostatni element

length lista - dlugosc listy **take** n lista - wez n elementow listy **sum** lista - suma listy
drop x lista - wyrzuc element x z listy **elem** x lista - sprawdz czy x jest w liscie
reverse lista - odwroc liste **min** a b - minimum z a i b

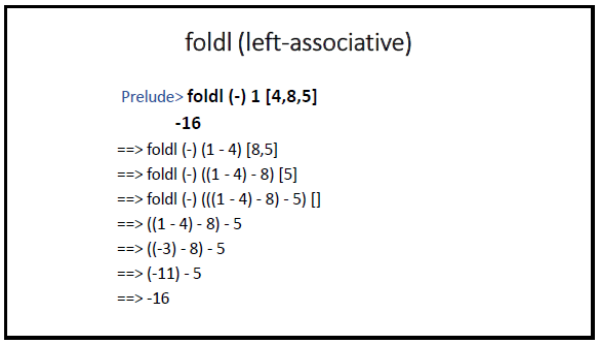
3. Funkcje wyższego rzędu

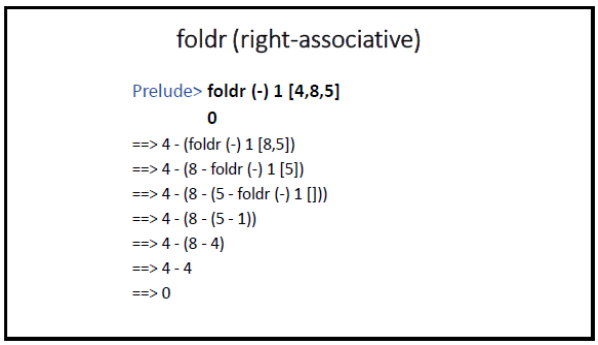
quicksort [] = []
quicksort (x:xs) = quicksort(**filter**(<x) xs) ++filter wybierze elementy < od x
[x]++
quicksort(filter(>=x) xs) filter wybierze elementy >= od x

map f [] = []
map f (x:xs) = (f x) : (map f xs)

filter _ [] = []
filter p (x:xs) = if (p x == True) then x: filter p xs
else filter p xs

1. Foldr i Foldl

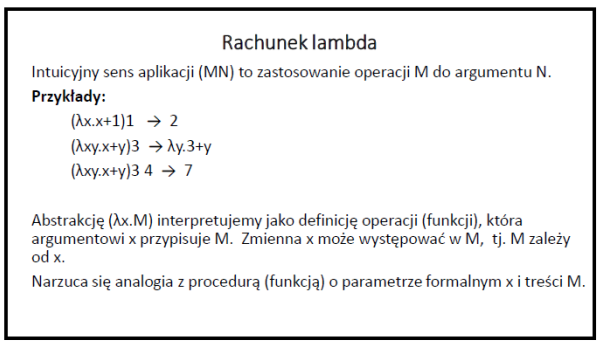


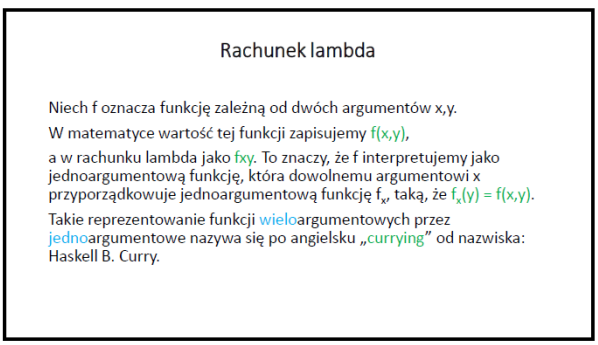


myFoldl f x [] = x
myFoldl f x (h:t) = myFoldl f (f x h) t

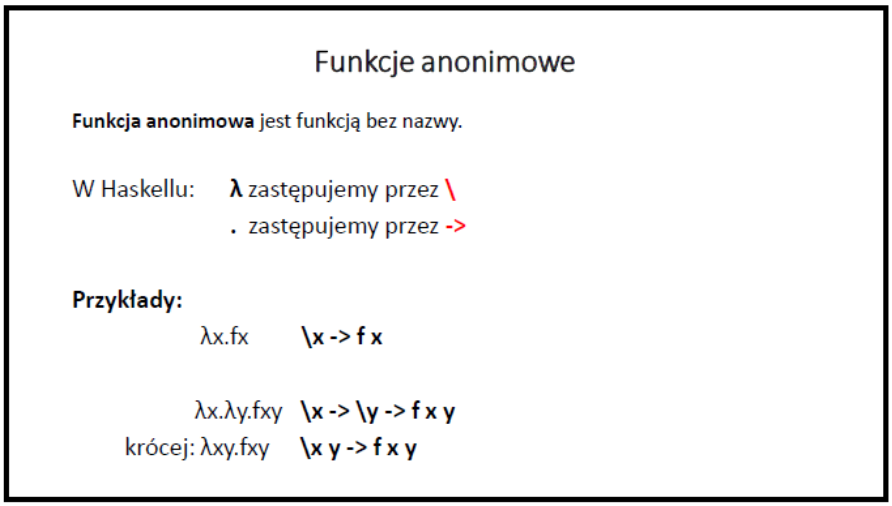
myFoldr f x [] = x
myFoldr f x (h:t) = f h (myFoldr f x t)

2. Rachunek Lambda





3. Funkcje anonimowe



Np.:
dodaj = \x y -> x+y
(\x y -> x*y) 2 5 = 10

4. Operator złożenia funkcji
reverse.reverse “abcde” = “abcde”

1. Własny typ
data TypKonstruktora = WartoscKonstruktora1 | WartoscKonstruktora2 | …

2. Konstruktory bezargumentowe

- data Bool = True | False
- data Marka = Opel | Ford | Fiat
- data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

```
przetlumacz :: Day -> String
przetlumacz d = case d of
                    Mon -> “Pon”
                    Tue -> “Wt”
                    ……..
```

3. Definicja własnego typu
data NazwaTypu = NazwaKonstruktora Arg1 Arg2 Arg3 …

data BookInfo = Book Int String [String]
Tworzymy instancje typu BookInfo (Konstruktor Indeks Tytuł Autorzy)
let book1 = Book 12345 “Harry Potter” [“J.K.R”, “J.R.R.T”]
:type book1 :: BookInfo

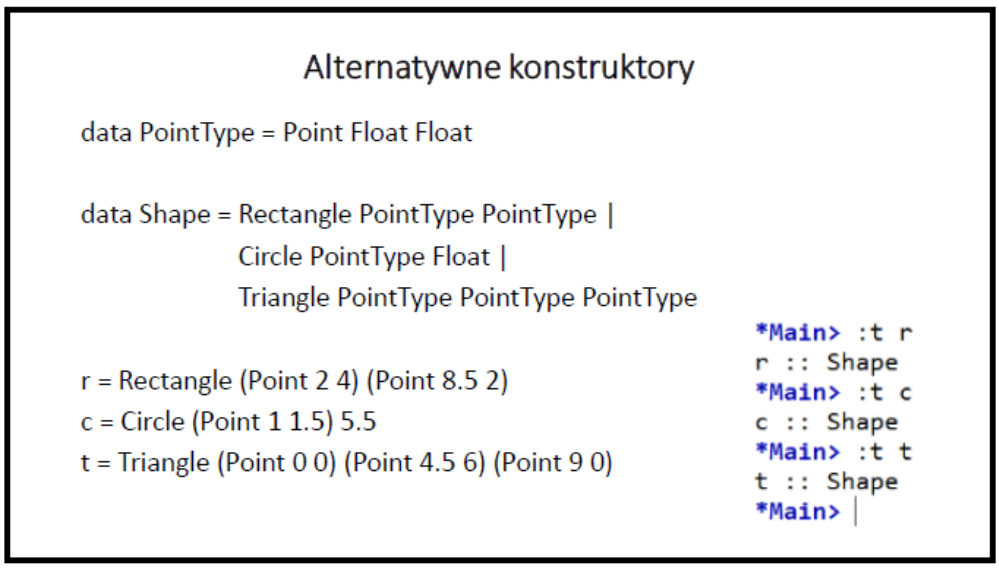
4. Synonimy typów

```
type NazwaTypu = TypJuzIstniejacy
type Nazwisko = String
```

```
type NazwaTypu = (Typ1, Typ2, …)
type But = (Char, Int)
```

```
type Punkt = (Double, Double)
odleglosc :: Punkt -> Punkt -> Double
odleglosc (x1, y1) (x2, y2) = sqrt ( (x1-x2)^2 + (y1-y2)^2 )
```

5. Alternatywne konstruktory



6. Typy parametryzowane

```
data NazwaTypu TypParametru = NazwaKonstruktora TypParametru1 TypParametru2 …
data PairType a = Pair a a
p = Pair 2 5
:type p :: PairType Integer
```

7. Typy rekurencyjne

```
data Naturalna = Zero | Succ Naturalna
n0 = Zero          n1 = Succ Zero          n2 = Succ (Succ Zero)
natToInt Zero = 0
natToInt (Succ n) = 1 + natToInt n
```

8. Typy rekurencyjne - listy

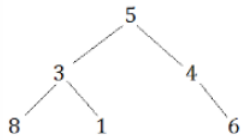
```
data NazwaTypu TypParametru = Konstruktor1 | Konstruktor2 Rekurencja
data List a = Empty | Cons a (List a)
l :: List Int
l = Cons 12 (Cons 8 (Cons 10 Empty))
len :: List a -> Int
len Empty = 0
len (Cons _ xs) = 1 + len xs
```

9. Typy rekurencyjne - drzewa

Typy rekurencyjne – drzewa

Drzewo jest puste albo składa się z wartości i dwóch poddrzew

```
data Tree a = Empty | Node a (Tree a) (Tree a)
t :: Tree Int
t = Node 5 (Node 3 (Node 8 Empty Empty)
           (Node 1 Empty Empty))
   (Node 4 Empty
           (Node 6 Empty Empty))
```




```
depth Empty = 0
depth (Node a l r) = 1 + max (depth l) (depth r)

preorder Empty = []
preorder (Node a l r) = [a] ++ preorder l ++ preorder r        //[5, 3, 8, 1, 4, 6]

inorder Empty = []
inorder (Node a l r) = inorder l ++ [a] ++ inorder r           //[8, 3, 1, 5, 4, 6[

postorder Empty = []
posterder (Node a l r) = posterder l ++ postorder r ++ [a]   //[8, 1, 3, 6, 4, 5]
```

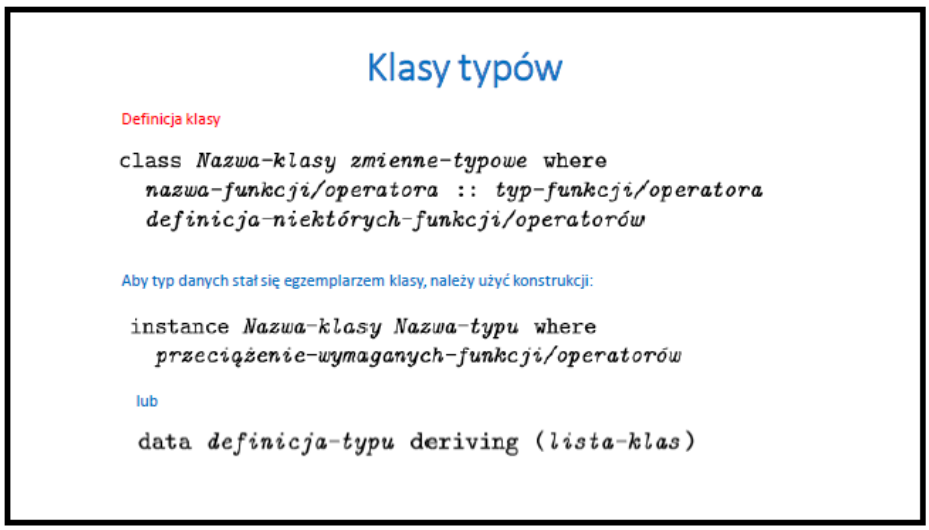
1. Moduły

module Nazwa (nazwy funkcji, nazwy typow) where
definicje klas typów, funkcji itp

Gdy (..) jest puste to wówczas wszystkie typy, klasy, funkcje są dostępne
Z modułów korzystamy za pomocą: import Nazwa

module M (A(..), B(Kb1, Kb3), f) where …
udostępnia typ danych ze wszystkimi jego konstruktorami, typ B z konstruktorami Kb1 i Kb3 oraz funkcje f

2. Klasy typów




```
class Eq a where
        (==), (/=) :: a -> a -> Bool

class Eq a => Ord a where                    //dziedziczenie
        max, min :: a -> a -> a
        ……..
```

3. Klasa Show, Read, Enum, Monady

4. Unie

```
data Unia typ1 typ2 = Jedno typ1 | Drugie typ2
                    deriving(Eq, Show)
```

5. Operacje wejścia/wyjścia

```
getChar
putChar c
getLine
putStr
putStrLn
return
firstLetter (x:xs) = return x
inicjaly = do
                    putStr "Podaj imie: "
                    a <- getLine
                    putStr "Podaj nazwisko: "
                    b <- getLine
                    putStr ("INICJALY: " ++ firstLetter a ++ "." ++ firstLetter b ++ ".")
```

6. Parser (przyjmuje napis i zwraca wartosc, wartość i nieskonsumowaną część napisu)

item - konsumuje pierwszy znak — item “anna” - [(‘a’,”nna”)]

failure - zawsze kończy się niepowodzeniem — failure “anna” - []

return v - zwraca v bez konsumowania wejścia return 1 “as” - [(1, “as”)]
p +++ q - zachowuje sie jak parser p w przypadku powodzenia, q w przypadku porażki