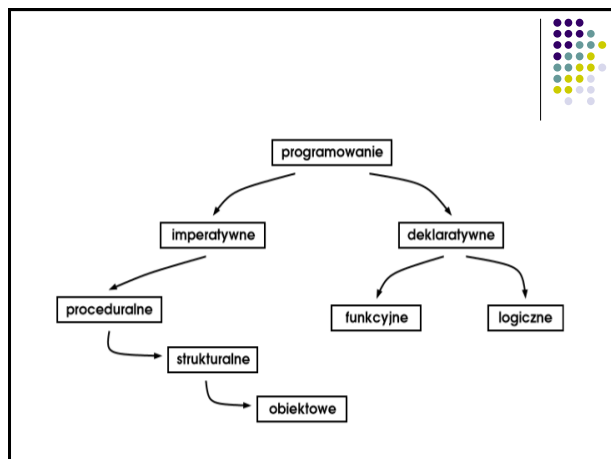
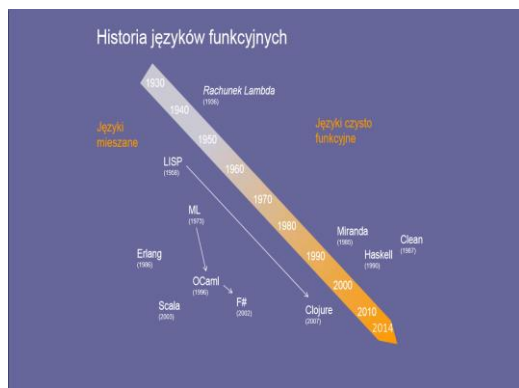


# Programowanie funkcyjne

## HASKELL cz.1



### Historia języków funkcyjnych



Podstawy matematyczne programowania funkcyjnego stanowi **rachunek funkcyjny lambda**.

Rachunek lambda pozwala m.in. zdefiniować funkcję bez nadawania jej nazwy.



(Alonzo Church, 1903-1995)

## Haskell

Haskell to język:

- silnie typowany
- stosuje obliczanie leniwe
- czysto funkcyjny



Haskell Brooks Curry (1900-1982)

## Haskell

- Praca interaktywna: zwana **sesją**, polega na wpisywaniu wyrażeń, które Haskell oblicza i wypisuje ich wynik (w najprostszym przypadku możemy używać Haskella podobnie jak kalkulatora)
- Tworzenie definicji złożonych funkcji, które później wykorzystujemy w obliczeniach. Zbiór definicji podanych Haskellowi, w odpowiedniej dla niego składni, zwany jest **skryptem**. Skrypt zapisuje się w pliku z rozszerzeniem **.hs**.

```

Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> 5*100
7888609052210118054117285652827862296732064351090230047702789306640625
Prelude> 5**100
7.888609052210118e69
Prelude> 5**1.5
11.180339887498949
Prelude> "Poznań"++"wart"++"poznania"
"Poznań324wartpoznania"
Prelude> "Poznań"++"wart"++"poznania"
"Poznańwartpoznania"
Prelude> "Poznań"++"wart "++"poznania"
"Poznań wart poznania"
Prelude> 'P'++"oznań"

```

```

Prelude> 5+2
7
Prelude> (+) 5 2
7
Prelude> sin (pi * 0.5)
1.0
Prelude> 2*e

<interactive>:1:2: Not in scope: 'e'
Prelude> let e=exp 1
Prelude> 2*e
5.43656365691809
Prelude> let a=2*e*pi
Prelude> 5*a
42.89078155253941
Prelude> a
8.578156310507882
Prelude>

```

#### Operatory:

==, >, <, >=, <=, /= (nie równa się)

**Wartości logiczne:** True, False

#### Operatory logiczne:

&& (and) np. True && False  
 || (or) np. False || True  
 not (negacja) np. not True

## Typy podstawowe

- Każda wartość (jak również funkcja) w Haskellu ma ściśle określony typ. Jawnie definiowanie typów nie jest konieczne, ponieważ Haskell sam rozpoznaje typ wartości.
- Sprawdzanie typu dowolnego wyrażenia  
`:type` lub `:t`
- Symbol `::` jest odczytywany jako "jest typu"  
`x :: y` oznacza „x jest typu y”

## Typy podstawowe

- Typy całkowite**
  - Int** (fixed-precision) - liczby całkowite z zakresu  $[-2^{29}.. 2^{29}-1]$ ,
  - Integer** (arbitrary-precision) - wartością Integer może być dowolna liczba całkowita (zarówno ujemna jak i dodatnia).
- Typy rzeczywiste**
  - Float** - liczba zmiennoprzecinkowa pojedynczej precyzji,
  - Double** - liczba zmiennoprzecinkowa podwójnej precyzji.
- Typ znakowy**
  - Char** - typ pojedynczego znaku. Jest to typ wyliczeniowy (pojedyncze znaki (Char) są w apostrofach, natomiast łańcuchy znaków (String) w cudzysłowach)

## Typy podstawowe

- Typ logiczny Bool**  
 Typ Bool jest typem wyliczeniowym zawierającym dwie wartości **False** (0) i **True** (1).
- Typ relacji między elementami**  
**Ordering**, typ relacji. Jest to typ wyliczeniowy posiadający trzy wartości:
  - LT** (less then - mniejszy niż)
  - EQ** (equal - równy)
  - GT** (greater then - większy niż)
 Wartości tego typu są zwracane między innymi przez funkcję `compare` porównującą dwa elementy:
 

```
> compare 1 2
LT
```

## Typy strukturalne

- **Listy** – ciąg elementów tego samego typu. Rozmiar listy nie jest określony - można dołączać do niej kolejne elementy.  
`> :t ['a','b','c']`  
`['a','b','c'] :: [Char]`
- **Krotki** – elementy krotki mogą być różnych typów. Rozmiar krotki jest ściśle określony podczas jej tworzenia. Nie jest możliwe dołączanie elementów do istniejącej krotki.  
`> :t (True,"Haskell",1)`  
`(True,"Haskell",1) :: (Num t) => (Bool, [Char], t)`

## Typy funkcji

- Na typ funkcji składają się typy przyjmowanych przez nią parametrów oraz typ wartości zwracanej przez funkcję. Typy te podajemy w następujący sposób:

**`nazwa_funkcji :: TypParam_1 -> TypParam_2 -> ... -> TypParam_n -> TypWartosciZwracanej`**

Przykład:

- definicja funkcji **inc** zwiększającej wartość liczby `Int` o jeden, wyglądają następująco:  
`inc :: Int -> Int`
- definicja funkcji **add** dodającej dwie liczby `Double`:  
`add :: Double -> Double -> Double`

## Typy polimorficzne

Typ polimorficzny oznacza rodzinę typów.  
Np.

**`kw :: Num a => a -> a`**  
`kw x = x * x`

**a** jest zmienną typową, której zakres ograniczamy przez klauzulę „`Num a =>`”, oznaczającą klasę typów liczbowych: `Integer`, `Float`, `Double` i parę innych.

Zatem funkcja `kw` podnosi do kwadratu dowolną liczbę, a wynik ma taki sam typ jak argument.

## Listy

`[]`,  
`[1,2,3]`,  
`["ab","bc","cd"]`

.. wyliczenie  
`[1..10]` ozn. [1,2,3,4,5,6,7,8,9,10]  
`[1.0,1.25..2.0]` ozn. [1.0,1.25,1.5,1.75,2.0]  
`[1,4..15]` ozn. [1,4,7,10,13]  
`[10,9..1]` ozn.[10,9,8,7,6,5,4,3,2,1]

## Listy nieskończone

Jeżeli ostatni element listy nie zostanie podany, Haskell utworzy listę o "nieskończonej" długości. Jest to możliwe dzięki leniwemu wartościowaniu. Wyznaczony zostanie tylko ten element listy, który będzie w danej chwili potrzebny.

`[1..]` ozn. [1,2,3,4,5,6,...]  
`take 3 [1..]` ozn. [1,2,3]

## Operacje na listach

- **++** konkatenacja  
`[3,1,3]++[3,7]`    wynik: [3,1,3,3,7]
- **concat** tworzy listę list (wszystkie tego samego typu) i łączy w jedną listę  
`concat [[1,2],[1,5]]`    wynik: [1,2,1,5]
- **reverse** odwraca kolejność elementów
- **length** liczba elementów listy
- **:** dodaje element na początek listy  
`1:[2,3]`    wynik: [1,2,3]

## Operacje na listach

- **head** zwraca głowę listy  
head [1,2,3,4] wynik: 1
- **tail** zwraca ogon listy  
tail [1,2,3,4] wynik: [2,3,4]
- **last** ostatni element listy  
last [1,2,3,4] wynik: 4
- **init** odwrotność last  
init [1,2,3,4] wynik: [1,2,3]

## Operacje na listach

- **take** zwraca  $n$  pierwszych elementów listy  
take 2 [1,7,8,4] wynik: [1,7]
- **drop** zwraca wszystkie oprócz  $n$  pierwszych elementów listy  
drop 2 [1,7,8,4] wynik: [8,4]

### Operacje na krotkach:

- **fst** zwraca pierwszy element pary  
fst(10,'a') wynik: 10
- **snd** zwraca drugi element pary  
snd(10,'a') wynik: 'a'

## Definiowanie funkcji

- Dodawanie dwóch liczb  
**add a b = a+b**
- Mnożenie dwóch liczb  
**mnoz (x, y) = if x == 0 || y == 0  
then 0 else x \* y**
- Sprawdzanie nieparzystości liczby  
**isOdd n = mod n 2 == 1**

## Definiowanie funkcji

```
sgn1 :: Integer -> Integer
sgn1 n = if n > 0 then 1 else if n == 0 then 0 else -1
```

Alternatywą dla konstrukcji warunkowych if-then-else jest zapis definicji z **dozorami**:

```
sgn2 :: Integer -> Integer
sgn2 n
| n > 0 = 1
| n == 0 = 0
| n < 0 = -1
```

## Definiowanie funkcji

Kolejna konstrukcja, która pozwala rozważać różne przypadki, to **dopasowywanie do wzorca**. Można je zresztą łączyć z innymi konstrukcjami, np.

```
sgn3 :: Integer -> Integer
sgn3 0 = 0
sgn3 n
| n > 0 = 1
| n < 0 = -1
```

Pierwszym wzorcem jest po prostu liczba zero. Jeśli parametr aktualny do niej pasuje (czyli jest zerem), to wykorzystywany jest ten właśnie wariant definicji. Drugi wzorec to  $n$ , do którego pasuje każda liczba.

## Leniwe wartościowanie

$$ax^2+bx+c=0$$

```
roots(a,b,c) = if d<0 then error "pierwiastki
zespolone"
```

```
else (r1,r2) where
  r1=e+sqrt d / (2*a)
  r2=e+sqrt d / (2*a)
  d=b*b-4*a*c
  e=(-b)/(2*a)
```

## Literatura



- B.O'Sullivan, J.Goerzen, D.Stewart, Real World Haskell, O'REILLY, 2008.
- K.Doets, J.van Eijck, The Haskell Road to Logic, Math and programming, 2004.
- G.Brzykcy, A.Meissner, Programowanie w Prologu i programowanie funkcyjne, Wyd.PP, 1999.
- Miran Lipovaca, Learn You a Haskell for Great Good!