

# Języki i Paradygmaty Programowania

Marcin Benke

MIM UW

27 lutego 2017

# Języki i Paradygmaty

Jakie to ma znaczenie,  
jakiego języka używamy?

## Jakie to ma znaczenie, jakiego języka używamy?

*Język kształtuje nasz sposób myślenia i określa, o czym możemy myśleć*

— Benjamin Lee Whorf

*Granice mojego języka oznaczają granice mojego świata*

— Ludwig Wittgenstein

# Język

- ▶ Komunikacja
  - ▶ z komputerem
  - ▶ z ludźmi

*Programs must be written for people to read,  
and only incidentally for machines to execute.*

# Język

- ▶ Komunikacja
  - ▶ z komputerem
  - ▶ z ludźmi

*Programs must be written for people to read,  
and only incidentally for machines to execute.*

- ▶ Ekspresja
  - ▶ program jako konstruktywny (wykonywalny) zapis idei
  - ▶ siła wyrazu: zwięźle i czytelnie wyrazić naszą ideę programu (algorytmu, systemu)?
  - ▶ estetyka

# Język

- ▶ Komunikacja
  - ▶ z komputerem
  - ▶ z ludźmi

*Programs must be written for people to read,  
and only incidentally for machines to execute.*

- ▶ Ekspresja
  - ▶ program jako konstruktywny (wykonywalny) zapis idei
  - ▶ siła wyrazu: zwięźle i czytelnie wyrazić naszą ideę programu (algorytmu, systemu)?
  - ▶ estetyka

- ▶ Abstrakcja
  - ▶ mechanizmy uogólniania istniejących konstrukcji i tworzenia nowych

*By relieving the brain of all unnecessary work, a good notation sets it  
free to concentrate on more advanced problems, and in effect increases  
the mental power*

— A. N. Whitehead

# Komunikacja: programowanie jako dzieło zespołowe

## Komunikacja

- ▶ z komputerem
- ▶ z ludźmi
- ▶ kod jest głównym kanałem komunikacji

*Any fool can write code that a computer can understand.  
Good programmers write code that humans can understand.*

— Martin Fowler

# Programowanie jako dzieło osobiste — ekspresja

- ▶ program jako konstruktywny (wykonywalny) zapis idei
- ▶ siła wyrazu:  
zwięźle i czytelnie wyrazić ideę programu (algorytmu, systemu)
- ▶ estetyka

*Every computer program is a model, hatched in the mind, of a real or mental process... If art interprets our dreams, the computer executes them in the guise of programs!*

— Abelson & Sussman



# Abstrakcja

- ▶ Rozpoznawanie wzorców, powtarzających się elementów
- ▶ Tworzenie nowych pojęć przez uogólnianie
- ▶ Kluczowy element efektywnej komunikacji
- ▶ Wsparcie abstrakcji miarą **siły wyrazu** języka
- ▶ W jakim stopniu język programowania pozwala na zdefiniowanie języka problemu i rozwiązań?

*Jakub spotkał na pustyni lwa. – Ja wiem – odezwał się Jakub. – Ty jesteś lew i ty możesz mnie pożreć, ale ja mogę jedną rzecz, której ty nie możesz. Ja mogę ciebie nazwać. Ty jesteś lew.*

— Tomasz Mann

# Zarządzanie złożonością

- ▶ Abstrakcja pozwala na opanowanie złożoności, niejako opakowując złożone mechanizmy, pozwalając na manipulację nimi w prostszy sposób
- ▶ Przykłady
  - ▶ tablice wielowymiarowe
  - ▶ podprogramy
  - ▶ rekurencja
  - ▶ obiekty
- ▶ Abstrakcje są często *nieszczelne*: izolacja jest niedoskonała i złożoność “wycieka”.

*Software systems are perhaps the most intellectually complex artifacts created by humans*

— Grady Booch, wykład Turinga

# Paradygmat

Sposób widzenia świata (tu: obliczeń)

- ▶ Metafora

- ▶ Opanowanie złożoności przez odwołanie do rzeczy znanych
- ▶ Mentalny model obliczeń

- ▶ Abstrakcja

- ▶ Uogólnianie powtarzających się schematów
- ▶ Opanowanie złożoności przez podział problemu na części i budowę rozwiązania z elementów.

- ▶ Idiom — swoisty dla danej kultury sposób ekspresji

Np. `(x:) <$> g` zamiast `do { xs <- g; return (x:xs) }`

Dostarczane przez biblioteki raczej niż język

# Paradygmaty a metafory

- ▶ Funkcje i operacje matematyczne

$$\theta^* = \eta \circ \theta \circ \eta^{-1}$$

*revwords = unwords ∘ reverse ∘ words*

*revwords "Ala ma kota"  $\rightsquigarrow$  "kota ma Ala"*

- ▶ Ciąg poleceń (instrukcji):

```
i=20; L: draw(i); i--; if i goto L
```

- ▶ Instrukcja zmienia stan maszyny

- ▶ Formuły logiczne

```
path(X,X) .
```

```
path(X,Z) :- edge(X,Y) , path(Y,Z) .
```

- ▶ Komunikujące się obiekty

```
BrowserWindow new openUrl: 'http://xkcd.com'
```

# Języki i paradygmaty programowania

Podstawowe paradygmaty programowania, z ich mechanizmami wyrazu, komunikacji, abstrakcji.

Programowanie:

- ▶ funkcyjne
- ▶ imperatywne
- ▶ w logice
- ▶ obiektowe

Języki programowania

- ▶ Haskell (świetny język funkcyjny)
- ▶ Haskell (świetny język imperatywny)
- ▶ Prolog
- ▶ Smalltalk

# Po co jest ten przedmiot?

*Lisp is worth learning for [...] the profound enlightenment experience you will have when you finally get it.*

*That experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot [...] the same can be said of Haskell, and for very similar reasons.*

— esr, How to Become a Hacker, On Learning Haskell

*Język, który nie zmienia naszych poglądów na programowanie  
nie jest wart poznania*

— Alan Perlis

# Plan wykładu

- ▶ Modele obliczeń i paradygmaty programowania
- ▶ Haskell i programowanie funkcyjne
- ▶ Składnia i analiza składniowa
- ▶ Typy, kontrola typów
- ▶ Semantyka, interpretery i kompilatory
- ▶ Prolog i programowanie w logice
- ▶ Smalltalk i programowanie obiektowe

# Zasady zaliczania

- ▶ Egzamin 52p
- ▶ Trzy małe programy zaliczeniowe po 8p
- ▶ Duży program zaliczeniowy 24p
- ▶ Suma tych punktów wyznacza ocenę
- ▶ Dla zaliczenia laboratorium trzeba
  - ▶ oddać wszystkie cztery programy w terminie (tolerujemy jedno spóźnienie do 48h, z karą 2p),
  - ▶ z każdego uzyskać co najmniej 25%,
  - ▶ łącznie uzyskać z nich minimum 60% punktów.



# Inne informacje

- ▶ Materiały do wykładu pojawiają się sukcesywnie na `moodle.mimuw.edu.pl`
- ▶ Klucz do kursu: JPP1617.5350
- ▶ Ostateczna wersja notatek po wykładzie.
- ▶ Notatki nie zastępują wykładu.
- ▶ Kontakt ze mną: `ben@mimuw.edu.pl`
- ▶ Konsultacje czwartki o 1200 pokój 5750, możliwe inne terminy.

**Proszę zadawać pytania!**

# Haskell

- ▶ Nazwa — Haskell Brooks Curry (1900–1982)
- ▶ Czysty język funkcyjny
  - ▶ Bez efektów ubocznych
  - ▶ Ułatwia wnioskowanie o programach
- ▶ Leniwy
  - ▶ Wyrażenia nie są obliczane wcześniej niż potrzeba
  - ▶ Umożliwia programowanie z (potencjalnie) nieskończonymi strukturami
  - ▶ Daje pełną kompozycyjność
- ▶ Zaprojektowany w 1990 jako standard leniwego języka funkcyjnego

```
primes :: [Integer]
primes = sieve [2..] where
  sieve (p:xs) = p : sieve[x | x<-xs, x`mod`p /=0]
```

# Czy ktoś go używa?

- ▶ AT&T
- ▶ Barclay
- ▶ Facebook (Sigma,  $10^6$  zapytań/sek.)
- ▶ Microsoft
- ▶ New York Times
- ▶ Standard Chartered
- ▶ ...i wielu innych

# Typy w językach funkcyjnych

## Języki funkcyjne

- ▶ typowane dynamicznie, gorliwe: Lisp, Clojure,
- ▶ typowane statycznie, gorliwe, nieczyste: ML, Scala
- ▶ typowane statycznie, leniwe, czyste: Haskell, Agda, Idris

## Haskell

- ▶ Kontrola (i rekonstrukcja) typów w czasie kompilacji.
- ▶ Niepoprawne typowo programy są odrzucane
- ▶ Typy są językiem specyfikacji i projektowania

*I like to think of types as warping our gravity,  
so that the direction we need to travel becomes "downhill"*

— @pigworker (Conor McBride)

# Funkcje

Jak opisać funkcję?

Zbiór par, n.p.  $\{(0, 1), (1, 2), \dots\}$

$$f(x) = x + 1$$

$$g(x) = x + 1$$

“optycznie” są to dwie funkcje:  $f$  i  $g$ , ale naprawdę to jedna funkcja

$$\lambda x. x + 1$$

Możemy wyznaczyć jej wartość dla danego argumentu

$$(\lambda x. x + 1)(2) \rightsquigarrow 2 + 1$$

# Rachunek lambda

- ▶ Alonzo Church 1932 (i inni)
- ▶ Minimalistyczny uniwersalny język programowania
  - ▶ Można zdefiniować każdą funkcję obliczalną (Turing-complete)
  - ▶ Tylko trzy konstrukcje:

$M$	$\rightarrow$	$x$	zmienna
		$\lambda x.M$	definicja funkcji
		$M_1(M_2)$	użycie funkcji

- ▶ ... i jedna reguła obliczenia:

$$(\lambda x.M)N \rightsquigarrow M[N/x]$$

- ▶ Leży u podstaw programowania funkcyjnego.

Notacja: aplikacja wiąże w lewo, zamiast  $(f(x))(y)$  piszemy " $f x y$ "

# Rachunek lambda — przykłady

$$(\lambda x.M)N \rightsquigarrow M[N/x]$$

$I = \lambda x.x$  — identyczność —  $I\ 5 = (\lambda x.x)\ 5 = x[5/x] = 5$

$K = \lambda x\ y.x = \lambda x.(\lambda y.x)$  — generator funkcji stałych —  $K\ 5\ y = 5$   
(możemy je też zdefiniować równaniami  $I\ x = x$  oraz  $K\ x\ y = x$ )

```
true = zero = K
false = K I = \x y.y
if cond e1 e2 = cond e1 e2
one = \f x. f x
succ n = \f x. f (n f x)
plus m n = \f x. m f (n f x)
plus one = \n f x. one f (n f x)
           = \n f x. f (n f x) = succ
```

# Po pierwsze: wartości

Wykonanie programu polega na obliczaniu wartości wyrażeń dla uzyskania ich wartości.

```
let x=2 in x+x  $\rightsquigarrow$  4
```

W czystym języku funkcyjnym:

- ▶ nie ma przypisania ani w ogóle zmiennych, których wartość może się zmieniać;
- ▶ zmienne są lokalnymi nazwami dla wartości;
- ▶ obliczenia nie mają efektów ubocznych;
- ▶ globalne są tylko stałe (nazwy funkcji, typów, etc);
- ▶ podstawowym mechanizmem jest zastosowanie funkcji do argumentów (wywołanie funkcji).



# Po drugie: przejrzystość

## Zasada przejrzystości

1. Zastąpienie dowolnego wyrażenia innym wyrażeniem o tej samej wartości daje równoważny program.
2. Wartość funkcji zależy tylko od wartości jej argumentów.

W Ocamlu `(print_string "foo")` ma tę samą wartość co `(print_string "bar")`, ale łatwo widzieć, że nie są równoważne (powód: efekty uboczne).

W C funkcja `int f(int x) {return (a+=x);}` nie jest przejrzysta (powód: zmienne globalne i przypisanie).

Przejrzystość ułatwia modularyzację, testowanie i wnioskowanie o programach

# Po trzecie: funkcje

Funkcje są “pełnoprawnymi obywatelami”:

- ▶ wartości mogą być funkcjami
- ▶ funkcje mogą być argumentami funkcji
- ▶ funkcje mogą być wynikami funkcji

Funkcje mogą być anonimowe:  $\lambda x \rightarrow x + 1$  oznacza funkcję, która dla argumentu  $x$  daje wartość  $x + 1$

Często definiujemy funkcje w terminach innych funkcji

```
f . g = \x -> f(g(x))      -- złożenie funkcji f i g
revwords = unwords . reverse . words
odd = not . even
heal = modify (hero . health) (+10)
writeln = liftIO putStrLn
```

# Przekazywanie argumentów

...czyli Schoenfinkelizacja i częściowe aplikacje

Dlaczego piszemy  $f\ x\ y = \dots$  a nie  $f\ (x, y) = \dots$  ?

Pierwsza forma pozwala zastosować  $f$  do jednego argumentu

$(f\ x)$  jest funkcją;  $f\ x\ y \equiv (f\ x)\ y$

Takie zastosowanie  $f$  nazywamy *częściową aplikacją*.

Z kolei  $g\ (x, y) = \dots$  definiuje funkcję jednoargumentową (oczekującą argumentu w postaci pary).

Izomorfizm tych postaci wyznaczają operacje (nazywane curry/uncurry, ale wprowadzone przez Schoenfinkela):

$\text{curry}\ g\ x\ y = g\ (x, y)$

$\text{uncurry}\ f\ (x, y) = f\ x\ y$

# Typy

Każda wartość ma swój typ  
(intuicyjnie możemy myśleć o typach jako o zbiorach wartości), np:

```
True  :: Bool
5     :: Int
'a'   :: Char
[1, 4, 9] :: [Int]           -- lista
"hello" :: String           -- (String = [Char])
("ala", 4) :: (String, Int) -- para
```

Funkcje oczywiście też mają typy:

```
isLetter :: Char -> Bool
ord      :: Char -> Int
```

W większości wypadków Haskell potrafi sprawdzić poprawność typów bez żadnych deklaracji (rekonstruuje typy)

# Polimorfizm

Funkcja

$$\text{fst } (x, y) = x$$

ma typ

$$(a, b) \rightarrow a$$

co oznacza:

“Dla dowolnych typów  $a$  i  $b$ , funkcja dająca dla argumentu typu  $(a, b)$  wynik typu  $a$ .”

Na przykład  $\text{fst } ("luty", 2014) \rightsquigarrow "luty" :: \text{String}$   
 $\text{fst } (\text{isdivby}, f) \rightsquigarrow \text{isdivby} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}$

# Polimorfizm

Jakie typy mają funkcje `curry`/`uncurry`?

Dla **dowolnych** typów  $a, b, c$  oraz funkcji  $g :: (a, b) \rightarrow c$ , wartością wyrażenia `curry g` jest funkcja typu  $a \rightarrow b \rightarrow c$

`curry :: ((a, b) -> c) -> a -> b -> c`

`uncurry :: (a -> b -> c) -> (a, b) -> c`

Ten rodzaj polimorfizmu czasem nazywany jest parametrycznym.

Funkcja polimorficzna działa tak samo niezależnie od typu argumentu.

Dlatego często z typu funkcji możemy się domyslić jej działania, np funkcja typu

$$(a, b) \rightarrow a$$

jeśli daje jakiś wynik, to musi to być pierwszy element pary (czyli ekstensjonalnie musi być funkcją `fst`).

# Przekroje

Operatory infiksowe są z natury swej dwuargumentowe. Podając operatorowi jeden z argumentów możemy uzyskać funkcję jednoargumentową.

Konstrukcja taka nazywa się *przekrojem* (section) operatora.

```
Prelude> (+1) 2
```

```
3
```

```
Prelude> (1+) 3
```

```
4
```

```
Prelude> (0-) 4
```

```
-4
```

Przekrojów używamy przeważnie, gdy chcemy taką funkcję przekazać do innej funkcji.

# Funkcje wyższego rzędu

Funkcjami wyższego rzędu nazywamy funkcje operujące na funkcjach, np złożenie (w **Prelude** operator `(.)`)

$\circ :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$   
`(f `o` g) x = f (g x)`

```
*Main> (+1) `o` (*2) $ 3
7
*Main> (+1) . (*2) $ 3
7
```

albo `flip`, które zamienia kolejność argumentów funkcji:

$\text{flip} :: (a \rightarrow b \rightarrow c) \rightarrow (b \rightarrow a \rightarrow c)$   
`flip f x y = f y x`



# Funkcje map i filter

**map** zamienia funkcję na elementach w funkcję na listach:

```
*Main> :info map
map :: (a -> b) -> [a] -> [b]
*Main> map (*2) [1,2,3]
[2,4,6]
```

**map f xs** daje listę złożoną z zastosowania funkcji **f** do każdego elementu **xs**

```
*Main> :info filter
filter :: (a -> Bool) -> [a] -> [a]
*Main> filter (>5) [1..8]
[6,7,8]
```

**filter p xs** daje listę złożoną z tych elementów **xs**, które spełniają predykat **p**

# Quicksort i partition

```
qs [] = []  
qs (x:xs) = (qs (filter (<=x) xs))  
            ++ [x]  
            ++ (qs (filter (>x) xs))
```

Ale można inaczej:

```
partition :: (a->Bool) -> [a] -> ([a],[a])  
partition p [] = ([],[])  
partition p (x:xs)  
    | p x = (x:ys,zs)  
    | otherwise = (ys,x:zs)  
    where (ys,zs) = partition p xs
```

```
qs' [] = []  
qs' (x:xs) = (qs' ys) ++ [x] ++ (qs' zs)  
    where (ys,zs) = partition (<x) xs
```

# Funkcja foldr

Policzmy sumę i iloczyn listy

```
suma [] = 0
```

```
suma (x:xs) = x + suma xs
```

```
prod [] = 1
```

```
prod (x:xs) = x * prod xs
```

Mozemy uogólnić

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
foldr op z [] = z
```

```
foldr op z (y:ys) = y `op` foldr op z ys
```

```
suma = foldr (+) 0
```

```
prod = foldr (*) 1
```

# Funkcje **foldr** i **foldl**

W ogólności **foldr** redukuje listę “pRawostronnie”

$$\text{foldr } \oplus z [x_1, x_2, \dots, x_n] = x_1 \oplus (x_2 \oplus \dots (x_n \oplus z) \dots)$$

Możemy rozważyć, poniekąd dualną, operację **foldl**:

$$\text{foldl } \oplus z [x_1, x_2, \dots, x_n] = (\dots ((z \oplus x_1) \oplus x_2) \oplus \dots) \oplus x_n$$

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z0 xs0 = lgo z0 xs0
  where
    lgo z []      = z
    lgo z (x:xs) = lgo (f z x) xs
```

Dla operatorów łącznych wynik **foldr** i **foldl** powinien być ten sam.

# foldr — ilustracja

$$\text{foldr } (\#) \ z \left( \begin{array}{c} \vdots \\ \swarrow \quad \searrow \\ x_1 \quad \vdots \\ \swarrow \quad \searrow \\ x_2 \quad [] \end{array} \right) = \begin{array}{c} \# \\ \swarrow \quad \searrow \\ x_1 \quad \# \\ \swarrow \quad \searrow \\ x_2 \quad z \end{array}$$

# Przykłady

Łatwo zauważyć, że

$$\text{foldr } (:) [] xs == xs$$

Ale co robią poniższe funkcje?

```
a xs ys = foldr (:) ys xs
```

```
s n = foldl (*) 1 [1..n]
```

```
r xs = foldl (flip(:)) [] xs
```

# Leniwe obliczenia

Mówimy że Haskell jest językiem leniwym:

- ▶ wyrażenia nie są obliczane wcześniej niż potrzeba,
- ▶ programowanie z (potencjalnie) nieskończonymi strukturami

```
from n = n:from(n+1)
-- const x y = x
Prelude> const 0 (from 1)
0
Prelude> take 5 (from 1)
[1,2,3,4,5]
```

- ▶ `from` buduje (potencjalnie) nieskończoną listę.
- ▶ Obliczenie wartości `const 0 (from 1)` nie wymaga obliczenia `from 1` (nie jest więc ono obliczane).
- ▶ Obliczenie `take 5 xs` wymaga 5 pierwszych elementów `xs`.

# Kiedy wykonywane są obliczenia?

- ▶ W językach gorliwych (C, Java, Ocaml) wartości argumentów są obliczane przed wywołaniem funkcji (wywołanie przez wartość itp).
- ▶ W językach leniwych nie przekazujemy wartości, ale “cuś” (ang. thunk), co pozwoli ją obliczyć.
- ▶ Funkcja wywoływana oblicza wartość jeśli i kiedy tego potrzebuje (ale tylko raz!)
- ▶ W praktyce obliczenie wartości jest niezbędne przy dopasowaniu do wzorca (ale tylko na tyle, na ile wzorzec tego potrzebuje — nie trzeba obliczać całej listy, żeby ustalić, czy jest ona pusta).



# Wymuszanie obliczeń

Operator ( $\$!$ ) oznacza podobnie jak  $\$$  aplikację funkcji, ale aplikację gorliwą — wymuszającą uprzednie obliczenie wartości argumentu.

Jest on zdefiniowany przy pomocy wbudowanej funkcji

```
seq :: a -> t -> t
```

Która oblicza swój pierwszy argument i daje w wyniku drugi.

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

NB gorliwa wersja **foldl** — **foldl'**:

```
foldl' f z []      = z  
foldl' f z (x:xs) = let z' = z `f` x  
                    in seq z' $ foldl' f z' xs
```

# Własności strategii obliczeń

- ▶ **Twierdzenie Churcha-Rossera:** jeśli dwie różne strategie (kolejności) obliczeń dadzą wynik, to będzie to ten sam wynik.
- ▶ Jeśli jakaś strategia obliczeń prowadzi do wyniku to leniwa też.
- ▶ Strategia gorliwa nie ma tej własności.

Leniwość daje pełną kompozycjonalność

$$k \ x \ y = x$$

Dla wszystkich  $x, y$  zachodzi  $k \ x \ y = x$

# Własności strategii obliczeń

- ▶ **Twierdzenie Churcha-Rossera:** jeśli dwie różne strategie (kolejności) obliczeń dadzą wynik, to będzie to ten sam wynik.
- ▶ Jeśli jakaś strategia obliczeń prowadzi do wyniku to leniwa też.
- ▶ Strategia gorliwa nie ma tej własności.

Leniwość daje pełną kompozycyjność

$$k \ x \ y = x$$

Dla wszystkich  $x, y$  zachodzi  $k \ x \ y = x$

to wydaje się oczywiste, ale co z

`k "OK" (error "crash!")`

# Własności strategii obliczeń

- ▶ **Twierdzenie Churcha-Rossera:** jeśli dwie różne strategie (kolejności) obliczeń dadzą wynik, to będzie to ten sam wynik.
- ▶ Jeśli jakaś strategia obliczeń prowadzi do wyniku to leniwa też.
- ▶ Strategia gorliwa nie ma tej własności.

Leniwość daje pełną kompozycjonalność

$$k \ x \ y = x$$

Dla wszystkich  $x, y$  zachodzi  $k \ x \ y = x$

to wydaje się oczywiste, ale co z

`k "OK" (error "crash!")`

Haskell: OK; ML, Scala, C: crash!

# Rekursja ogonowa

Rozważmy dwie funkcje liczące długość listy:

```
length1 [] = 0
length1 (x:xs) = 1 + length1 xs
```

```
length2 xs = len xs 0
  where
    len :: [a] -> Int -> Int
    len []      a = a
    len (_:xs) a = len xs (a+1)
```

Która z nich jest lepsza?

# Rekursja ogonowa

```
benchmarking length1 10^6
time                28.16 ms      (26.77 ms .. 29.28 ms)
```

```
benchmarking length2 10^6
time                4.260 ms      (4.199 ms .. 4.326 ms)
```

length2 ma lepsze parametry, gdyż rekurencja w len jest *ogonowa*:  
wywołanie rekurencyjne jest ostatnią czynnością do wykonania.  
Kompilator zwykle potrafi zamienić taką rekursję na iterację.