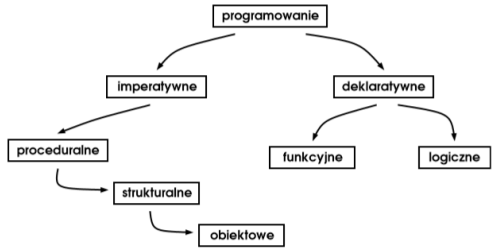
Programowanie funkcyjne

HASKELL



Dwie główne grupy języków programowania:

1. imperatywne

Języki imperatywne charakteryzują się tym, że programista wyraża w nich czynności (w postaci rozkazów), które komputer ma w pewnej kolejności wykonywać. Program jest więc listą rozkazów do wykonania, stąd nazwa imperatywne („rozkazowe").

2. deklaratywne

W językach deklaratywnych programista pisząc program podaje (deklaruje) komputerowi pewne zależności oraz cele, które program ma osiągnąć. Nie podaje się jednak wprost sposobu osiągnięcia wyników.

Języki imperatywne mówią komputerowi, jak ma osiągnąć wynik (choć samego wyniku nie określają wprost), języki deklaratywne opisują, co ma być osiągnięte (choć nie podają na to bezpośredniego sposobu).

Funkcyjny paradygmat programowania

jest podparadygmatem **programowania deklaratywnego.**

W programowaniu funkcyjnym (także: **funkcjonalnym**) nie podajemy maszynie czynności do wykonania, ale opisujemy pożądany wynik: tutaj przy pomocy **funkcji**. W związku z tym, zadaniem interpretera lub kompilatora języka funkcyjnego jest wyłącznie obliczenie wartości funkcji głównej, a więc pewnego wyrażenia. Ewentualne wejście/wyjście programu jest jedynie efektem ubocznym wykonywania obliczeń.

Programowanie czysto funkcyjne

- **funkcje czyste (referencyjnie przezroczyste),** które zawsze przyjmują tę samą wartość dla tych samych argumentów (nie zależą w żaden sposób od stanu maszyny, od urządzeń wejścia/wyjścia, użytkownika, pamięci zewnętrznej...), od akcji, które mogą powodować i wykorzystywać efekty uboczne, gdy są wykonywane, ale nie są funkcjami w rozumieniu programowania funkcyjnego.
- Funkcja jest tutaj więc rozumiana całkowicie **matematycznie** — jako przyporządkowanie pewnych wartości pewnym argumentom.

Programowanie czysto funkcyjne

- W związku z pojęciem programu jako złożenia pewnych funkcji, **nie występują zmienne znane z programowania imperatywnego** (bo są one abstrakcją stanu maszyny, do którego funkcja dostępu nie ma) ani tradycyjne pętle (potrzebujące do kontroli swojego działania dostępu do stanu maszyny), zamiast których używa się rekurencji.
- Języki funkcyjne traktują funkcje jako **wartości pierwszego rzędu**, czyli mogą być one takimi samymi wartościami argumentów i wyników innych funkcji jak dane „prostsze" — liczby, napisy, listy...
- Podstawy programowania funkcyjnego (w szczególności czysto funkcyjnego) są także ściśle matematyczne: opiera się ono zwykle teoretycznie na **przeźroczystości referencyjnej** oraz **λ-rachunku**, który pozwala na wprowadzenie najściślejszej możliwej teoretycznie kontroli typów wraz z automatycznym o nich wnioskowaniem.

Cechy charakterystyczne języków czysto funkcyjnych

Przezroczystość referencyjna (przezroczystość odwołań)

Polega ona na tym, że wyrażenie mające tę własność może być w każdym swoim użyciu zastąpione swoją wartością.

Typowym przykładem wyrażeń przezroczystych referencyjnie są aplikacje funkcji w matematyce:
jeśli funkcja f : A → B jest matematycznie dobrze zdefiniowana i zachodzi **f(a) = b,** to wszędzie gdzie jest **f(a)** możemy wstawić **b** bez zmiany sensu wyrażeń;
tak więc następujące wyrażenia są równoważne:

g(b,b),
g(f(a),b),
g(b,f(a)),
g(f(a),f(a)).

Trwałe struktury danych

Trwałe struktury danych to takie, które nie są zmieniane w trakcie przetwarzania i mogą być wielokrotnie wykorzystywane. Mogą też być fragmentami współdzielone, co znacznie polepsza efektywność ich przetwarzania (zarówno czasową jak i przestrzenną).

Automatyczne zarządzanie pamięcią

Jak wiele języków wyższego poziomu, języki funkcyjne same zarządzają pamięcią przez pewien rodzaj zbierania nieużytków (programista i tak nie ma bezpośredniego dostępu do stanu maszyny, w tym do pamięci).
Zbieranie nieużytków (także *odśmiecanie*, ang. garbage collection) polega na takim zarządzaniu alokacją pamięci na stercie, by można było wykrywać i zwalniać obszary zaalokowane, ale już nieużywane.
Wszystkie języki wysokiego poziomu (na przykład Java, C#, Python, Haskell, Prolog, Lisp, Scheme), które oferują niejawną alokację danych na stercie, muszą jakiś sposób ich niejawnych dealokacji mieć w postaci zbierania nieużytków. Jest kilka odmian zbierania nieużytków.

Leniwa ewaluacja

W językach czysto funkcyjnych można obliczać wartości wyrażeń **leniwie, czyli w ostatniej chwili**, gdy są już potrzebne.
W językach imperatywnych dominuje ***gorliwa ewaluacja***, czyli obliczanie wyrażeń jak tylko zostaną napotkane.

Kolejność wykonywania czynności obliczeniowych

W programowaniu imperatywnym jest ona z góry określona przez układ programu, natomiast w programowaniu czysto funkcyjnym interpreter/kompilator sam może wybierać czynności do wykonania tak, by było to optymalne.

Struktury potencjalnie nieskończone

Leniwe obliczenia umożliwiają definiowanie struktur, które są, matematycznie rzecz biorąc, nieskończone (choć oczywiście dostęp do ich elementów w praktyce jest realizowany w miarę potrzeb, więc nigdy nie trzeba całej takiej struktury w pamięci fizycznie przechowywać).

Silne typowanie

Większość języków czysto funkcyjnych jest bardzo silnie typowana, co znaczy, że każda dana (także nieznana przed uruchomieniem programu) może zostać w trakcie kompilacji (także częściowej) zakwalifikowana do konkretnego typu lub klasy typów.

Funkcje jako wartości pierwszego rzędu

Funkcje są zwykłymi danymi, więc inne funkcje mogą na nich działać, jak i zwracać je jako wyniki.

Zwięzłość

Zapis funkcyjny programu jest zwykle bardziej zwięzły niż imperatywny.

Matematyczna ścisłość

Dzięki czystości funkcyjnej programy funkcyjne są ścisłe i dobrze zdefiniowane w sensie matematycznym, co daje możliwość prostego (czasem niemal automatycznego) dowodzenia poprawności programów, a także utrzymania tej poprawności przy jego modyfikowaniu/rozszerzaniu.

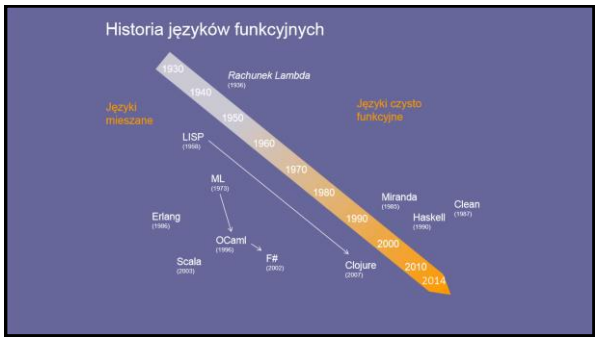
Wady

Trudność w wyrażeniu niektórych operacji imperatywnych

Niektóre operacje — jak interakcja z zewnętrzem (plik, użytkownik, urządzenia we/wy) czy zmiany/uwzględnianie pewnych globalnych danych/stanów/ustawień (liczby pseudolosowe, nietrwałe struktury danych, wyjątki, efekty uboczne) — nie mogą same w sobie być zaimplementowane w postaci funkcji przezroczystych referencyjnie. Istnieją jednak metody ich prezentowania przez pewne złożenia funkcji działających na określonych typach (monady) lub łańcuchy implikacji (jak w programowaniu logicznym).

Brak iteracji

Nie ma w językach funkcyjnych takich czynności jak pętle. W związku z tym, wszelkie powtórzenia muszą być definiowane rekurencyjnie — podobnie jak w programowaniu logicznym. Rekurencja potrafi być dużo bardziej nieefektywna od iteracji. (dzięki przezroczystości referencyjnej — możemy korzystać z rekurencji ogonowej, która może być obliczona efektywnie (to jest bez użycia stosu i kolejnych wywołań podprogramu).
W rekurencji ogonowej, funkcja zdefiniowana rekurencyjnie występuje w swojej własnej definicji jako funkcja zewnętrzna, a więc jest ostatnim wywołaniem potrzebnym do obliczenia wyniku. Przy takiej konstrukcji nie ma potrzeby budowania stosu.



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**.

Operatory:

==, >, <, >=, <=, /=

+, -, *, /, **, ^

mod, div

```
Prelude> mod 4 3
1
Prelude> 4 `mod` 3
1
Prelude> div 7 2
3
Prelude> 7 `div` 2
3
```

```
Prelude> 5^100
78886090522101180541172856528278622967320643510902300477027893
06640625
Prelude> 5**100
7.88860905221012e69
Prelude> 5**1.5
11.180339887498949
Prelude> 5/6
0.8333333333333334
Prelude> 234==2
False
```

```
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>
```

Wartości logiczne: True, False

Operatory:

&& (and)
|| (or)
not (negacja)

```
Prelude> True && True
True
Prelude> True && False
False
Prelude> True || True
True
Prelude> True || False
True
Prelude> False || False
False
Prelude> not True
False
Prelude> not (True && False)
True
```

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.

[Int]	[1, 2, 3]
[Char]	['a', 'b', 'c', 'd']
[[Int]]	[[1], [1, 4], []]
[(String, Bool)]	[("Ala" , True), ("kot", False)]

Typy strukturalne

Krotki – elementy krotki mogą być różnych typów.

Rozmiar krotki jest ściśle określony podczas jej tworzenia.

Nie jest możliwe dołączanie elementów do istniejącej krotki.

(Int, Char)	(1, 'a')
(Int, Char, Float)	(1, 'a', 3.4)
((Bool, String), Int)	((True, "Ala"), 2)
([Int], Char)	([-1, 4, 2], 'c')

```
> :t (True,"Haskell",1)
  (True,"Haskell",1) :: (Num t) => (Bool, [Char], t)
```

Typy funkcji

Na typ funkcji składają się typy przyjmowanych przez nią argumentów oraz typ wartości zwracanej przez funkcję.

Typy te podajemy w następujący sposób:

nazwa_funkcji :: TypArg_1 -> TypArg_2 -> ... -> TypArg_n ->**TypWartosciZwracanej**

Przykład:

- definicja typu funkcji **inc** zwiększającej wartość liczby Int o jeden, wyglądają następująco:

  inc :: Int -> Int

- definicja typu 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.

Operator specyfikowania typu

Każda wartość (jak również funkcja) w Haskellu ma ściśle określony typ. Jawne definiowanie typów nie jest konieczne.

Uwaga. Specyfikowanie typu jest zazwyczaj opcjonalne. Haskell sam wywnioskuje typ danego identyfikatora.

:type lub **:t**

```
Prelude> let c=[1,2,3]++[5,4,3,2]
Prelude> :t c
c :: Num a => [a]
Prelude> let f x = x*x
Prelude> :type f
f :: Num a => a -> a
```

Symbol **::** jest odczytywany jako "jest typu"

x :: y oznacza „x jest typu y"

```
Prelude> :t 2+2
2+2 :: Num a => a
Prelude> :t 2.0+2.4
2.0+2.4 :: Fractional a => a
Prelude> :t sin
sin :: Floating a => a -> a
Prelude> :t length
length :: [a] -> Int
Prelude> :t length [1,2,3]
length [1,2,3] :: Int
Prelude> :t elem
elem :: Eq a => a -> [a] -> Bool
Prelude> :t elem 2 [1,2,8,4,1]
elem 2 [1,2,8,4,1] :: Bool
Prelude> :t (2+9)/3
(2+9)/3 :: Fractional a => a
```

Funkcje

Definiowanie wartości

let pi = 3.141592653589793

Definiowanie funkcji

square x = x ^ 2

triangleArea a h = 0.5 * a * h

Prelude> square 2

4.0

Prelude> triangleArea 4 2

4.0

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

Wyrażenie warunkowe if ... then ... else

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

Uwaga: Nie ma konstrukcji if ... then

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

Strażnicy (*guards*)

```
sgn x | x < 0   = -1
      | x == 0  = 0
      | x > 0   = 1
```

$$sgn(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}$$

```
sgn x | x < 0     = -1
      | x == 0    = 0
      | otherwise = 1
```

$$sgn(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & wpp \end{cases}$$

Kolejność strażników ma znaczenie:

```
f a b | a >= b    = "wieksze lub rowne"
      | a == b    = "rowne"
      | otherwise = "niewieksze"
      | a < b     = "mniejsze"
```

```
ghci> f 1 1
"wieksze lub rowne"
ghci> f 1 5
"niewieksze"
```

Uwaga

```
otherwise = True
```

Dopasowanie wzorca (pattern matching)

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

```
sgn :: Integer -> Integer
sgn 0 = 0
sgn 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 wzorzec to n, do którego pasuje każda liczba.

```
1 conj a b = if a == True && b == True
               then True else False

2 conj True True = True
  conj a    b    = False

3 conj True True = True
  conj _    _    = False
```

Kolejność wzorców ma znaczenie:

```
conj _    _    = False
conj True True = True
```

```
ghci> conj True True
False
```

Klauzula where

Definiuje wartości i funkcje użyte wewnątrz wyrażenia:

```
1 volume r = a * pi * cube
      where a = 4 / 3
            cube = r ^ 3
2 volume r = a * pi * cube r
      where a = 4 / 3
            cube x = x ^ 3
```

Wyrażenie let ... in

```
volume r = let a = 4 / 3
               cube = r ^ 3
           in a * pi * cube
```

Lokalność definicji po where (let)

```
a = 1
volume r = ...
    where a = 4 / 3
fun x = a * x

ghci> fun 5
5
```

where i let razem (Definicje po let przesłaniają te po where)

```
fun x = let y = x + 1
        in y
        where y = x + 2

ghci> fun 5
6
```

Wyrażenie case ... of

```
1 fun 1 =  2
  fun 2 =  3
  fun _ = -1

2 fun n = case n of
            1 ->  2
            2 ->  3
            _ -> -1
```

Argumentem **case** może być dowolne wyrażenie:

```
fun n = case n < 0 of
          True  -> "ujemna"
          False -> "dodatnia"
```

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)
```

```
silnia2 : : Integer – > Integer
silnia2  0 = 1
silnia2  n = n * silnia2 (n – 1)
-----------------------------------------------------------------------------
silnia3 n = silniaPOM n 1
            where silniaPOM 0 x = x
                  silniaPOM n x = silniaPOM (n-1) (x*n)
-----------------------------------------------------------------------------
silnia4 :: Integer -> Integer
silnia4 n = silniaPOM n 1
            where silniaPOM 0 x = x
                  silniaPOM n x = silniaPOM (n-1) (x*n)
```

```
Ok, modules loaded: Main.
*Main> silnia1 10
3628800
*Main> silnia2 10
3628800
*Main> silnia3 10
3628800
*Main> silnia4 10
3628800
*Main> silnia2 2.5

<interactive>:14:9:
    No instance for (Fractional Integer) arising from the literal ‘2.5’
    In the first argument of ‘silnia2’, namely ‘2.5’
    In the expression: silnia2 2.5
    In an equation for ‘it’: it = silnia2 2.5
*Main> silnia4 2.5

<interactive>:15:9:
    No instance for (Fractional Integer) arising from the literal ‘2.5’
    In the first argument of ‘silnia4’, namely ‘2.5’
    In the expression: silnia4 2.5
    In an equation for ‘it’: it = silnia4 2.5
*Main> silnia3 2.5
```

```
*Main> :type silnia1
silnia1 :: (Num a, Eq a) => a -> a
*Main> :type silnia2
silnia2 :: Integer -> Integer
*Main> :type silnia3
silnia3 :: (Num a, Eq a) => a -> a
*Main> :type silnia4
silnia4 :: Integer -> Integer
*Main>
```

Num - rodzina typów liczbowych
Eq – rodzina (klasa) typów, dla których zdefiniowane jest porównywanie (operatory == i /=)

*Main> silnia3 2.5

nigdy się nie skończy - ani poprawnym wynikiem, ani przepełnieniem stosu, bo stosu nie używa (!)

W definicjach funkcji silnia3 i silnia4 użyto *rekurencji ogonowej*, bo funkcja zdefiniowana rekurencyjnie występuje w swojej własnej definicji jako funkcja, a więc jest ostatnim wywołaniem potrzebnym do obliczenia wyniku.

Przy takiej konstrukcji (i przy referencyjnej przezroczystości) nie ma potrzeby budowania stosu.

Rekurencja ogonowa zastępuje efektywnie pętle - podobnie jak one nie zużywa dodatkowej pamięci za każdym „obrotem".

Rekurencja a iteracja

silnia n = if n==0 then 1 else n***silnia**(n-1)

3!

silnia 3 = 3*silnia 2
silnia 2 = 2*silnia 1
silnia 1 = 1 silnia 0
silnia 0 = 1
silnia 1 = 1*1=1
silnia 2 = 2*1=2
silnia 3 = 3*2=6

Rekurencja a iteracja

silnia n = **silniaPOM** n 1
silniaPOM n x = if n== 0 then x
else **silniaPOM** (n-1) (n*x)

3!

silnia 3 = silniaPOM 3 1
silniaPOM 3 1 = silniaPOM 2 3*1
silniaPOM 2 3 = silniaPOM 1 2*3
silniaPOM 1 6 = silniaPOM 0 1*6
silniaPOM 0 6 = 6

Rekurencja a iteracja

W definicji „iteracyjnej" wprowadzamy dodatkową funkcję, której argumenty odgrywają rolę **akumulatorów**, czyli służą do przechowywania wyników częściowych funkcji.

Dodatkowa funkcja: **silniaPOM**.

- pierwszy z argumentów dodatkowej funkcji silniaPOM odgrywa rolę licznika i zmienia się co do wartości od *n* do *0* (w kolejnych wywołaniach funkcji silniaPOM mamy *n-1*),
- w drugim argumencie przechowywane są iloczyny częściowe *n*x*.

Obliczanie n-tego wyrazu ciągu Fibonnaciego

(definicja rekurencyjna)

fib n = if n==0 then 1
else if n==1 then 1
else **fib**(n-1) + **fib**(n-2)

(wersja „akumulatorowa")

fib n = **fibPOM** n 1 1
fibPOM n f1 f2 = if n==1 then f1
else **fibPOM** (n-1) (f1+f2) f1

- Rolę licznika odgrywa pierwszy argument dodatkowej funkcji fibPOM, a jego wartość zmienia się od *n* do *1*.
- W akumulatorach reprezentowanych przez dalsze argumenty funkcji przechowywane są wartości dwóch kolejnych wyrazów ciągu Fibonacciego.

Definiowanie operatorów dwuargumentowych

```
x & y  = (x + y) / 2
x #^ y = x == y - 1

ghci> 1 & 2
1.5
ghci> 1 #^ 2
True
```

Uwagi

- Operatory dwuargumentowe są funkcjami „zapisywanymi" pomiędzy argumentami
- Nazwy operatorów składają się z jednego lub więcej symboli

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!