Wykład 8

Haskell – czysto funkcyjny język programowania z ewaluacją leniwą i silną statyczną typizacją

GHCi - praca interakcyjna w cyklu REPL

Skrypty

- Wiązanie identyfikatora z wyrażeniem
- Literały funkcyjne i definicje funkcji
- Listy i krotki

Funkcje wyższych rzędów i moduł Prelude

Pliki źródłowe (moduły) Haskella

- Definiowanie funkcji w modułach
- Wyrażenie „case”
- Definiowane funkcji z wykorzystaniem wzorców

Ewaluacja gorliwa i leniwa

Najważniejsze języki funkcyjne

Język	**Typizacja**	**Wyznaczanie zakresu**	**Ewaluacja**	**Efekty uboczne**
Lisp	dynamiczna	dynamiczne	gorliwa	tak
Scheme	dynamiczna	statyczne	gorliwa + leniwa (kontynuacje)	tak
Clojure (platforma JVM)	dynamiczna	statyczne lub dynamiczne	gorliwa	tak
Standard ML OCaml	statyczna, mocna	statyczne	gorliwa	tak
Haskell	statyczna, mocna	statyczne	leniwa	nie

Haskell

Haskell jest czysto funkcyjnym językiem programowania z leniwą ewaluacją i silną statyczną typizacją.

Nazwa upamiętnia amerykańskiego matematyka i logika Haskella Curry'ego.

Najpopularniejszym kompilatorem jest GHC (Glasgow Haskell Compiler). Jest on dystrybuowany z interakcyjnym interpreterem i debugerem GHCi.

Ta strona udostępnia wiele materiałów do nauki Haskella.

- https://wiki.haskell.org/Learning_Haskell

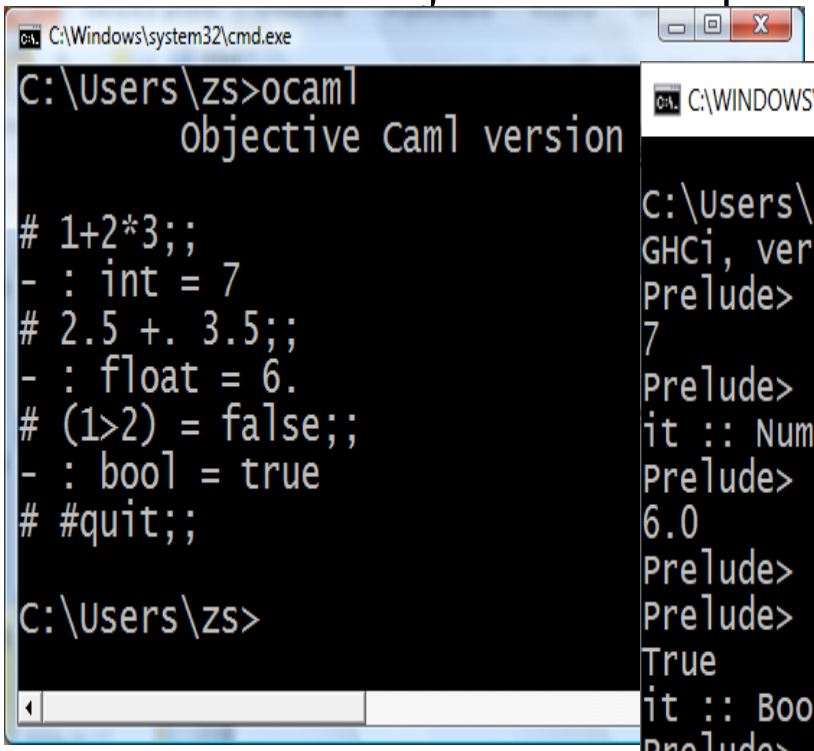
Tutaj też można znaleźć wiele interesujących materiałów.

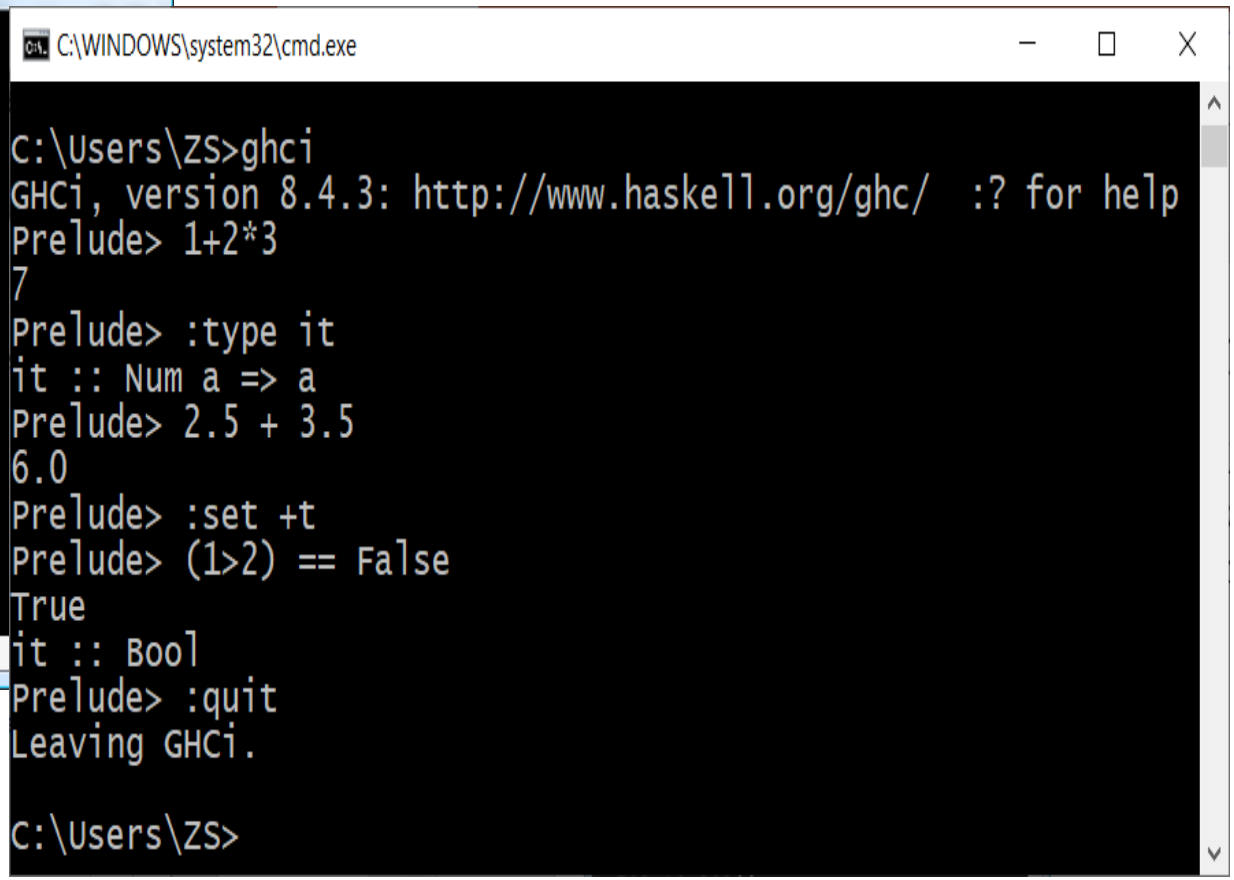
- https://www.haskell.org/documentation

Ten wykład jest zwięzłym wprowadzeniem do Haskella. Wykorzystywane są wiadomości i (przepisane w języku Haskell) przykłady z wykładów 1, 2, 3 i 5.

Praca interakcyjna w cyklu REPL

Każdy funkcyjny język programowania umożliwia pracę interakcyjną w cyklu REPL (REPL = Read-Evaluate-Print Loop). Najprostszym środowiskiem jest tu wiersz poleceń (okno terminala).





Haskell posiada też własne, nieco wygodniejsze środowiska do pracy interakcyjnej: WinGHCi (w systemie Windows).

GHCi

Najpopularniejszym kompilatorem Haskella jest GHC (Glasgow Haskell Compiler). Jest on dystrybuowany z interakcyjnym interpreterem i debugerem GHCi.

Emacs z wtyczką haskell-mode również używa GHCi (po wybraniu Haskell > Start interpreter). Jako symbol zachęty (ang. prompt) wykorzystywana jest tam grecka litera λ (lambda).

Praca interakcyjna w cyklu REPL w GHCi jest dość podobna do pracy interakcyjnej w OCamlu. Najważniejsze różnice są podane poniżej.

- GHCi po naciśnięciu klawisza Enter ewaluuje wprowadzone wyrażenie. W OCamlu ewaluacja była wykonywana dopiero po wprowadzeniu dwóch średników (;;).
- Interpreter OCamla razem z wartością wyrażenia wyświetla zawsze typ. GHCi nie wyświetla typu automatycznie, ale po ustawieniu flagi +t (za pomocą polecenia :set +t) typy będą zawsze wyświetlane.
- O typ konkretnego wyrażenia można zapytać za pomocą polecenia :t *wyrażenie*.
- Oprócz tego GHCi wiąże identyfikator it z ostatnio obliczoną wartością.
- GHCi oddziela wartość od typu dwoma dwukropkami (OCaml używał jednego dwukropka). Pojedynczy dwukropek denotuje w Haskellu infiksowy konstruktor listy niepustej.

Obcięcia (zawężenia) funkcji w Haskellu

W OCamlu i Haskellu można otrzymać z infiksowego operatora dwuargumentowego funkcję w postaci rozwiniętej:

(☺) ≡ λx → λy → x ☺ y

Haskell pozwala też dokonać obcięcia lub zawężenia takiej funkcji (ang. section), tzn. ustalenia jednego z jej argumentów:

(x ☺) ≡ λy → x ☺ y

(☺ y) ≡ λx → x ☺ y

Dotyczy to również infiksowych i miksfiksowych wbudowanych konstruktorów wartości.

W języku OCaml łatwo można ustalić lewy argument:

(☺) x ≡ fun y → x ☺ y

Ustalenie prawego argumentu wymaga napisania wyrażenia funkcyjnego:
fun x → x ☺ y

Typy kwalifikowane w Haskellu

Jaki powinien być typ funkcji dodawania?

OCaml

```
# (+);;
- : int -> int -> int = <fun>
# (+.);;
-: float -> float -> float = <fun>
```

W języku OCaml mechanizm przeciążania (ang. overloading) jest stosowany wyłącznie w operatorach porównania: =, <>. ==, !=, <, <= itd., np. (=): 'a -> 'a -> bool .

Haskell

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
```

Typy kwalifikowane (ang. qualified types) pozwalają na kontrolowane wprowadzenie przeciążania do języka Haskell. Dodawanie jest zdefiniowane dla wszystkich typów numerycznych (należących do klasy typów Num).

```
(==) :: Eq a => a -> a -> Bool
(<) :: Ord a => a -> a -> Bool
```

Num, Eq i Ord są klasami typów. Typy kwalifikowane w Haskellu będą przedstawione poźniej.

Skrypty

Skrypt (ang. script) jest plikiem z ciągiem poleceń, przeznaczonych do sekwencyjnego wykonania przez jakiś program.

Plik z kodem źródłowym w języku OCaml powinien mieć rozszerzenie .ml (np. foo.ml). Skrypt w języku Haskell może mieć dowolne rozszerzenie, ale zwykle stosuje się standardowe rozszerzenie dla plików źródłowych z programami Haskella .hs (np. foo.hs).

Skrypt z pliku foo.ml można załadować i wykonać w konsoli interaktywnej za pomocą polecenia #use, np.

```
#  #use "foo.ml";;
```

Skrypt z pliku foo.hs można załadować i wykonać w konsoli interaktywnej za pomocą polecenia :script, np.

```
Prelude> :script foo.hs
```

Na SKOSie jest umieszczony plik w1skrypt.hs, zawierający przykłady z wykładu 1, przetłumaczone możliwie dokładnie na język Haskell (jeśli to było możliwe).

Wiązanie identyfikatora z wyrażeniem

W skrypcie (i w pracy interakcyjnej w cyklu REPL) do wiązania identyfikatora z wyrażeniem można używać słowa kluczowego let (podobnie jak w OCamlu). Od wersji 8.0.1 nie jest to już konieczne. W poniższych przykładach let będzie jednak używane, ponieważ w pracowniach pod Ubuntu mamy zainstalowaną wersję 7.10.3.

```
Prelude> :set +t
Prelude> let x = 3+2
x :: Num a => a
Prelude> x' = 3+2                -- od GHC 8.0.1 można też tak
x' :: Num a => a
Prelude> x'
5
it :: Num a => a
Prelude>
```

Literały funkcyjne i definicje funkcji

Definicje funkcji (również rekurencyjnych) to szczególny przypadek wiązania identyfikatora z wartością. Do tworzenia literału funkcyjnego Haskell używa ukośnika \, ponieważ przypomina grecką literę λ, a rachunek lambda leży u podstaw wszystkich języków funkcyjnych.

```
Prelude> let double = \x -> 2*x
double :: Num a => a -> a
Prelude> -- lub   double = \x -> 2*x    od GHC 8.0.1
Prelude> double 6
12
it :: Num a => a
Prelude> (\x -> 2*x) 6                  -- funkcja anonimowa = literał funkcyjny
12
it :: Num a => a
Prelude> let twice x = 2*x              -- jest to wygodny skrót notacyjny
twice :: Num a => a -> a
Prelude> twice 2+3                      -- aplikacja wiąże najmocniej
7
it :: Num a => a
Prelude> let plus = \x y -> x+y
plus :: Num a => a -> a -> a
Prelude>
```

Funkcja error

Funkcja biblioteczna error :: String -> a wyświetla komunikat o błędzie i kończy program.

Długa, nie pokazująca struktury, jednowierszowa definicja funkcji silnia wynika z ograniczeń GHCi, o których była mowa wcześniej. Definicje funkcji należy umieszczać w modułach.

```
Prelude> let silnia n = if n==0 then 1 else n*silnia(n-1)
silnia :: (Eq p, Num p) => p -> p
Prelude> let silnia n = if n==0 then 1 else if n>0 then n*silnia(n-1) else error "ujemny argument"
silnia :: (Num p, Ord p) => p -> p
Prelude> silnia (-4)
*** Exception: ujemny argument
CallStack (from HasCallStack):
  error, called at <interactive>:36:67 in interactive:Ghci29
Prelude>
```

Listy

Pojedynczy dwukropek denotuje w Haskellu infiksowy konstruktor listy niepustej.

```
Prelude> let l1 = 1:2:3:[]          -- pełna notacja
l1 :: Num a => [a]
Prelude> let l2 = [1, 2, 3]         -- notacja skrócona
l2 :: Num a => [a]
Prelude> l1 == l2                   -- równość strukturalna
True
it :: Bool
Prelude> 2 /= 2                     -- nierówność strukturalna
False
it :: Bool
Prelude> [1,2]++[2,3]               -- konkatenacja list
[1,2,2,3]
it :: Num a => [a]
Prelude> tail [1,2,3]
[2,3]
it :: Num a => [a]
Prelude> reverse [1,2,3]
[3,2,1]
it :: Num a => [a]
```

Pary i krotki

```
Prelude> (8, "osiem")                          -- nawiasy są konieczne
(8,"osiem")
it :: Num a => (a, [Char])                     -- napisy są listami znaków
Prelude> (1,1.0,"jeden")
(1,1.0,"jeden")                                -- to jest krotka trzyelementowa
it :: (Fractional b, Num a) => (a, b, [Char])
Prelude> (1,(1.0,"jeden"))                     -- to jest para
(1,(1.0,"jeden"))
it :: (Fractional a1, Num a2) => (a2, (a1, [Char]))
Prelude> fst (8, "osiem")
8
it :: Num a => a
```

Polecenie :info

Polecenie :info wyświetla informacje o zadanym argumencie. Jest to szczególnie użyteczne w przypadku operatorów.

```
Prelude> :info (+)
class Num a where
  (+) :: a -> a -> a
 ...
        -- Defined in `GHC.Num'
infixl 6 +
Prelude> :info (*)
class Num a where
  ...
  (*) :: a -> a -> a
  ...
        -- Defined in `GHC.Num'
infixl 7 *
Prelude> :info (^)                    -- potęgowanie
(^) :: (Num a, Integral b) => a -> b -> a       -- Defined in `GHC.Real'
infixr 8 ^
Prelude>
```

Dopasowanie do wzorca w definicjach zmiennych

```
Prelude> let x = (False,10)
x :: (Bool, Integer)
Prelude>  let (z,y)=x
y :: Num t => t
z :: Bool
Prelude> y
10
it :: Num t => t
Prelude> z
False
it :: Bool
Prelude> let (False,y)=x
y :: Num t => t
Prelude> y
10
it :: Num t => t
Prelude> let (True,y)=x
y :: Num t => t     -- skompiluje się i wykona (leniwa ewaluacja!)
Prelude> y          -- dopiero teraz jest potrzebna wartość y, co wymaga ewaluacji prawej strony,
                    -- dopasowania wzorca i w efekcie spowoduje błąd wykonania
*** Exception: <interactive>:1:4-13: Irrefutable pattern failed for pattern (True, y)
```

Funkcje wyższych rzędów i moduł Prelude

W Haskellu wiele z przedstawionych na poprzednich wykładach funkcjonałów należy do standardowego preludium.

```
curry :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c
(.) :: (b -> c) -> (a -> b) -> a -> c          -- operator składania funkcji
($) :: (a -> b) -> a -> b                      -- operator aplikacji  (wiąże w prawo)
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldl :: (a -> b -> a) -> a -> [b] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Standardowe preludium (moduł Prelude) w Haskellu jest zawsze bezpośrednio dostępne i zawiera bardzo dużo użytecznych definicji.

Informacje o najczęściej używanych można znaleźć w pliku Prelude.pdf na SKOSie.

Pliki źródłowe (moduły) Haskella

W GHCi (i w skryptach Haskella) można było definiować funkcje, ale nie jest to wygodne, choćby z tego powodu, że trzeba się zmieścić w jednym wierszu. W praktyce wyklucza to wykorzystanie wzorców w definicjach funkcji.

Pliki źródłowe (moduły) Haskella powinny mieć rozszerzenie .hs (np. foo.hs) i są one ładowane za pomocą polecenia :load (np. :load foo.hs). Więcej informacji o modułach będzie na kolejnych wykładach.

W plikach źródłowych nie wolno do wiązania identyfikatorów z wyrażeniami używać słowa kluczowego let. Wyrażenie let ... in ... jest oczywiście dopuszczalne. Nie wolno też powtórnie wiązać identyfikatora z wyrażeniem (co jest dopuszczalne w pracy interakcyjnej) ani umieszczać wyrażeń do ewaluacji.

```
-- plik: add.hs
add x y = x + y
```

Załadowanie pliku add.hs w GHCi oraz wykorzystanie funkcji add wygląda następująco:

```
Prelude> :load add.hs
[1 of 1] Compiling Main             ( add.hs, interpreted )
Ok, one module loaded.
*Main> :type add
add :: Num a => a -> a -> a
*Main> add 2 9
11
```

Definiowanie funkcji w modułach

W Haskellu zalecane jest poprzedzanie definicji funkcji jej typem. Stanowi to bardzo użyteczną dokumentację, ale jest też wykorzystywane przez kompilator do weryfikacji typu, wygenerowanego automatyczne przez system inferencji typów. Podobnie jak w OCamlu, programista może ewentualnie zawęzić wygenerowany najogólniejszy typ.

```
-- plik: add.hs
add :: Int -> Int -> Int
add x y = x + y
```

Załadowanie pliku add.hs w GHCi oraz wykorzystanie funkcji add wygląda następująco:

```
Prelude> :load add.hs
[1 of 1] Compiling Main             ( add.hs, interpreted )
Ok, one module loaded.
*Main> :type add
add :: Int -> Int -> Int
*Main> add 2 9
11
```

Wyrażenie „case” (1)

case *wyr* **of**
  w1 -> wyr1
  ⋮
| *wn -> wyrn*

Porównaj z wyrażeniem match w OCamlu (wykład 2, str. 16)

```
imply :: (Bool, Bool) -> Bool
imply pb =
  case pb of
    (True,True)   -> True
    (True,False)  -> False
    (False,True)  -> True
    (False,False) -> True

imply2 pb =
  case pb of
    (True, x)  -> x
    (False, x) -> True
```

Wyrażenie „case” (2)

```
imply3 pb =
  case pb of
    (True, x)  -> x
    (False, _) -> True

imply4 pb =
  case pb of
    (True, False) -> False
          _       -> True

imply5 (True, False) = False
imply5       _       = True

-- postać rozwinięta, operator infiksowy
(==>) True False = False
(==>)  _      _    = True
-- False ==> True
```

Definiowane funkcji z wykorzystaniem wzorców

Wyrażenie case jest w Haskellu stosunkowo rzadko wykorzystywane (w odróżnieniu od wyrażenia match w OCamlu). Wygodniejsza jest składnia, pokazana niżej na przykładach.

```
imply5 :: (Bool, Bool) -> Bool
imply5 (True, False) = False
imply5       _           = True

-- postać rozwinięta, operator infiksowy
(==>) :: Bool -> Bool -> Bool
(==>) True False = False
(==>)    _     _      = True
-- użycie   False ==> True
```

Ewaluacja gorliwa i leniwa

Najważniejsze strategie ewaluacji (wartościowania, obliczania), stosowane w językach programowania to ewaluacja gorliwa i leniwa. Poniżej przypomniane są informacje z wykładu 5.

- *Ewaluacja gorliwa* lub ewaluacja sterowana danymi (ang. eager evaluation, strict evaluation, data-driven evaluation, supply-driven evaluation).
  Jest stosowana najczęściej.
- *Ewaluacja leniwa* lub ewaluacja sterowana popytem (ang. lazy evaluation, non-strict evaluation, demand-driven evaluation).

Stosowana strategia ewaluacji stanowi jedną z najważniejszych charakterystyk języka programowania. Zwykle obok głównej strategii, języki programowania udostępniają mechanizmy, umożliwiające lokalne stosowanie drugiej strategii.

OCaml i Scheme wykorzystują ewaluację gorliwą, natomiast Haskell – leniwą.

Poniżej idea obu strategii zostanie wyjaśniona na przykładzie definiowania zmiennej. Precyzyjna definicja strategii wartościowania wymagałaby użycia pewnych formalizmów, np. rachunku lambda.

Definicja zmiennej

Definicja zmiennej składa się z trzech faz:

(1) deklaracja zmiennej
(2) zdefiniowanie wyrażenia, z wartością którego zmienna ma być związana
(3) ewaluacja wyrażenia i związanie zmiennej z obliczoną wartością

Powyższe fazy mogą być wykonywane jednocześnie lub kolejno.
We wszystkich językach funkcyjnych (1) i (2) **muszą** być wykonane jednocześnie.

- W językach funkcyjnych z ewaluacją gorliwą (OCaml, SML, Scheme, …) wykonywana jest od razu faza (3) i zmienna jest wiązana z wartością.
- W językach funkcyjnych z ewaluacją leniwą (Haskell, …) faza (3) jest wykonywana dopiero wtedy, kiedy wartość zmiennej jest potrzebna.

Analogicznie:

- W językach funkcyjnych z ewaluacją gorliwą argument funkcji jest obliczany zawsze, a do funkcji jest przekazywana jego wartość.
- W językach funkcyjnych z ewaluacją leniwą argument funkcji jest przekazywany do funkcji jako wyrażenie, którego wartość jest obliczana dopiero wtedy, kiedy jest potrzebna.

Leniwa ewaluacja koniunkcji i alternatywy

Większość współczesnych gorliwych języków programowania, m.in. Scala, OCaml, Java, C++, ewaluuje koniunkcję i alternatywę leniwie. W Haskellu to jest oczywiste.

---------- OCaml

```
# true || failwith "error";;
- : bool = true
# false && failwith "error";;
- : bool = false
```

---------- Haskell

```
Prelude> True || error "error"
True
Prelude> False && error "error"
False
```

Struktury nieskończone w języku Haskell

Haskell jest językiem funkcyjnym z ewaluacją leniwyą, dzięki czemu struktury nieskończone otrzymujemy w nim „za darmo”. Nieskończoną listę, składającą się z samych jedynek definiujemy następująco:

```
ones :: [Int]
ones = 1:ones

*Main> take 5 ones
[1,1,1,1,1]
```

Ciąg rosnących liczb całkowitych zaczynający się od *k* to po prostu [*k*..], np.

```
*Main> take 5 [30..]
[30,31,32,33,34]
```

Dostępny jest też bardzo czytelny sposób konstrukcji list (ang. list comprehensions), np.

```
*Main> [x^2 | x <- [1..5]]
[1,4,9,16,25]
```

Idea i nazwa pochodzą z teorii mnogości, gdzie analogiczny zbiór można zdefiniować tak: $\{x^2 \mid x \in \{1..5\}\}$.

Mechanizm konstrukcji list w Haskellu

W języku Haskell dostępna jest bardzo użyteczna abstrakcja lingwistyczna – mechanizm konstrukcji list (ang. list comprehension). Oto dowód, że jest to rzeczywiście tylko abstrakcja lingwistyczna.

```
[ x | x <- xs]              =  xs
[ e |  x <- xs]             =  map (\x->e) xs
[ e |  x <- xs, p x]        =  [e | x <- filter p xs]
[ e | x <- xs, y <- ys]     =  concat [ [e | y <- ys] | x <- xs]
                                concat :: [[a]] -> [a]   konkatenuje (spłaszcza) listę list
```

Przykłady.

```
[x^2 | x <- [1..5]]  =  map (\x-> x^2) [1..5] = [1,4,9,16,25]
xys  = [(x,y) | x <- [1..3], y <- ['a','b']]
     = [(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
xys' = concat [[(x,y) | y <- ['a','b']] | x <- [1..3]]
     = [(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

```
factors :: Int -> [Int]
factors n = [k | k <- [1..n], n `mod` k == 0]     -- lista wszystkich podzielników n
factors 15 => [1,3,5,15]
```

Definiowanie ciągów arytmetycznych w Haskellu

To jest tylko lukier syntaktyczny dla operacji z klasy `Enum`.

[a, b ..] ≡ a : a+d : a+2d : a+3d : … gdzie d = b-a

[a, b .. c] ≡ a : a+d : a+2d : a+3d : … : c : [] gdzie d = b-a

c nie musi się znaleźć w utworzonej liście (patrz przykłady poniżej)

[-1.5, 0 .. 7] ≡ [-1.5, 0.0, 1.5, 3.0, 4.5, 6.0, 7.5]

[2, 5 .. 13] ≡ [2, 5, 8, 11]

[a ..] ≡ a : a+1 : a+2 : a+3 : …

[a .. c] ≡ a : a+1 : a+2 : a+3 : … : c : []

c nie musi się znaleźć w utworzonej liście (patrz przykłady poniżej)

[2.5 .. 5.45] ≡ [2.5, 3.5, 4.5, 5.5]

[2.5 .. 5.55] ≡ [2.5, 3.5, 4.5, 5.5]

Przykład.

```
factorial :: Int -> Int
factorial n = foldl (\i acc -> i*acc) 1 [1..n]
factorial 5 => 120
```

Sito Eratostenesa w języku Haskell

Nieskończoną listę liczb pierwszych, otrzymaną za pomocą sita Eratostenesa, definiujemy następująco:

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

*Main> take 6 primes
[2,3,5,7,11,13]
```

Wykorzystywanie list nieskończonych wymaga ostrożności. Na przykład w wyniku ewaluacji wyrażenia filter (<= 5) [1..] wyprodukowany zostanie początek listy wynikowej [1,2,3,4,5

po czym proces zapętla się w poszukiwaniu liczb spełniających predykat w ogonie nieskończonej listy wejściowej. Biblioteka standardowa Haskella zawiera funkcję takeWhile :: (a -> Bool) -> [a] -> [a] , której ewaluacja kończy się po sfalsyfikowaniu predykatu.

```
Prelude> takeWhile (<= 5) [1..]
[1,2,3,4,5]
```

Ciąg liczb Fibonacciego w języku Haskell

Poniższe wyrażenie generuje liczby Fibonacciego w sposób, który może zaskoczyć programistów, nieprzywykłych do ewaluacji leniwej, ale jest naturalny w Haskellu.

Dwie początkowe liczby Fibonacciego zostały umieszczone jawnie w nieskończonej liście. Reszta jest konstruowana następująco. Tworzona jest lista par, zbudowana za pomocą zip (patrz wykład 2, zip i zipWith są w standardowym preludium) z dwóch list liczb Fibonacciego, przesuniętych o jedną pozycję. Pierwsza para to (0,1). Funkcjonał map bierze taki nieskończony strumień par i zwraca nieskończony strumień sum elementów tych par, co jest zgodne z definicją kolejnych liczb Fibonacciego.

Typ Integer dla liczb całkowitych dowolnej precyzji umożliwia wyliczenie dowolnie dużych liczb Fibonacciego. Biblioteki wszystkich współczesnych języków programowania udostępniają takie typy, np. OCaml – moduł Big_int, Java – klasa BigInteger, Scala – klasa BigInt.

```
fibs :: [Integer]
fibs = 0 : 1 : map (\(a,b) -> a+b) (zip fibs (tail fibs))
-- lub zwięźlej
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
*Main> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

Haskell – ewaluacja leniwa

Leniwa ewaluacja może spowodować subtelne efekty w definiowanych funkcjach.

```
ff  0   _  = True
ff  _   0  = True
Ff  _   _ = False

Ff1  _  0  = True
ff1  0   _  = True
ff1   _  _  = False
```

Funkcje ff i ff1 dla argumentów o określonych wartościach zachowują się identycznie.

Dla argumentów, których ewaluacja nie doprowadza do obliczenia wartości, ich zachowanie może być różne.

```
*Main> ff  0  (error "Och!")
True
*Main> ff1  0  (error "Och!")
*** Exception: Och!
*Main> ff  (error "Och!")  0
*** Exception: Och!
*Main> ff1  (error "Och!")  0
True
```

Funkcja ff jest "bardziej zdefiniowana" względem swojego drugiego argumentu,

natomiast funkcja ff1 jest "bardziej zdefiniowana" względem swojego pierwszego argumentu.