Elementy języka **Haskell**

- Cechy języka
- Historia języka
- Proste przykłady
- Środowisko interakcyjne
- Typy i klasy
- Definiowanie funkcji
- Wyrażenia listowe
- Deklarowanie typów, danych i klas
- Monady

Cechy języka

- zwięzłe programy
- silny system typów
- wyrażenia listowe
- funkcje rekurencyjne
- funkcje wyższego rzędu
- czyste funkcje (bez efektów ubocznych)
- formalny opis efektów ubocznych
- obliczenia leniwe
- wnioskowanie z równań

Historia języka

- lata 30-te Alonzo Church formułuje rachunek lambda
- lata 50-te John McCarthy tworzy język Lisp
- lata 60-te Peter Landin tworzy czysty język funkcyjny ISWIM (bez przypisania zmiennych)
- lata 70-te John Backus tworzy język FP z funkcjami wyższego rzędu
- lata 70-te Robin Milner i inni tworzą pierwszy nowoczesny język funkcyjny ML z polimorfizmem i wnioskowaniem typów
- lata 80-te David Turner tworzy komercyjny język MIRANDA
- 1987 międzynarodowy komitet rozpoczyna prace nad językiem Haskell (logik Haskell Curry)
- lata 90-te Philip Wadler i inni opracowują koncepcję klas dla przeciążenia i monad dla obsługi wyjątków
- 2003 opublikowano Haskell Report
- 2010 poprawiono i zaktualizowano Haskell Report

Proste przykłady

```
sum []     = 0
sum (n:ns) = n + sum ns
```

Funkcja **sum** jest typu Num a => [a] -> a

```
qsort []     = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a <- xs, a <= x]
    larger  = [b | b <- xs, b > x]
```

Funkcja **qsort** jest typu Ord a => [a] -> [a]

Środowisko interakcyjne

- kompilator GHC (Glasgow Haskell Compiler)
- program GHCi
- uruchamianie:

```
$ ghci
> 2+3*4
14
> (2+3)*4
20
> sqrt (3^2+4^2)
5.0
> :quit
```

- wybrane funkcje standardowe (biblioteka *Standard Prelude*):

```
> head [1,2,3,4,5]
1
> tail [1,2,3,4,5]
[2,3,4,5]
> [1,2,3,4,5] !! 2
3
> take 3 [1,2,3,4,5]
[1,2,3]
> drop 3 [1,2,3,4,5]
[4,5]
```

```
> length [1,2,3,4,5]
5
> sum [1,2,3,4,5]
15
> product [1,2,3,4,5]
120
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

- zastosowanie funkcji ma wyższy priorytet niż inne operatory:

matematyka	Haskell
f(a, b) + cd	f a b + c*d
f(x)	f x
f(x, y)	f x y
f(g(x))	f (g x)
f(x, g(y))	f x (g y)
f(x)g(y)	f x * g y

- skrypty w Haskellu

```
-- test.hs
double x = x + x
quadruple x = double (double x)
```

```
$ ghci test.hs
> quadruple 10
40
> take (double 2) [1,2,3,4,5]
[1,2,3,4]
> :quit
```

- w osobnym terminalu można dopisać do pliku **test.hs** następujące definicje:

```
factorial n = product [1..n]
average ns  = sum ns `div` length ns
```

```
> :reload
> factorial 10
3628800
> average [1,2,3,4,5]
3
> :quit
```

Komenda	Działanie
:load name	załadowanie skryptu
:reload	przeładowanie bieżącego skryptu
:set editor name	zdefiniowanie polecenia edytora
:edit name	edycja skryptu
:edit	edycja bieżącego skryptu
:type expr	pokaż typ wyrażenia
:?	pokaż wszystkie komendy
:quit	wyjdź z GHCi

- grupowanie definicji na podstawię wcięć:

```
a = b + c                 a = b + c
  where                     where
    b = 1                     { b = 1;
    c = 3                       c = 2};
d = a * 2                 d = a * 2
```

```
a = b + c where {b = 1; c = 2};
d = a * 2
```

Unikaj tabulacji (stosuj spacje).

- komentarze

```
-- ble ble ble

{- ble ble ble
   ble ble ble -}
```

Typy i klasy

```
False :: Bool
True :: Bool
not :: Bool -> Bool

not False :: Bool
not True :: Bool
not (not False) :: Bool
```

$$\frac{f :: A \mapsto B \qquad e :: A}{f\ e :: B}$$

Wyrażenie **not 3** nie ma sensu. Podczas kontroli typu występuje *błąd typu* (argument funkcji **not** powinien być typu **Bool**). Kontrola typu odbywa się przed rozpoczęciem obliczeń i w ich trakcji błąd ten na pewno nie wystąpi.

Warunkowe wyrażenie:

```
if True then 1 else False
```

zawiera *type error* gdyż **1** i **False** powinny być tego samego typu.

```
> :type not
not :: Bool -> Bool
> :type False
False :: Bool
> :type not False
not False :: Bool
```

Podstawowe typy

- Bool — wartości logiczne
- Char — znaki
- String — łańcuchy znaków
- Int — liczby całkowite ustalonej precyzji (GHCi $-2^{63}..2^{63}-1$)
- Integer — liczby całkowite dowolnej precyzji
- Float — liczby zmiennopozycyjne pojedynczej precyzji
- Double — liczby zmiennopozycyjne podwójnej precyzji

```
> 2^63::Int
-9223372036854775808
> 2^63
9223372036854775808
> sqrt 2::Float
1.4142135
> sqrt 2::Double
1.4142135623730951
```

Listy

Typ **[T]** oznacza listę obiektów typu **T**.

```
[False,True,False] :: [Bool]
['a','b','c','d'] :: [Char]
["One","Two","Three"] :: [String]

> :type []
[] :: [t]
> :type [[]]
[[]] :: [[t]]
```

Krotki

Typ **(T1, T2, …, Tn)** oznacza krotkę obiektów typów **T1**, **T2**, …, **Tn**.

```
(False,True) :: (Bool,Bool)
(False,'a',True) :: (Bool,Char,Bool)
("Yes",True,'a') :: (String,Bool,Char)

> :type ()
() :: ()
```

specjalny typ oznaczający pustą krotkę

Funkcje

Typ **T1 -> T2** oznacza typ wszystkich funkcji odwzorowujących argument typu **T1** w wynik typu **T2**.

```
not :: Bool -> Bool

> :type even
even :: Integral a => a -> Bool

add :: (Int, Int) -> Int
add (x,y) = x + y

zeroto :: Int -> [Int]
zeroto n = [0..n]
```

Curried functions

```
add’ :: Int -> (Int -> Int)
add’ x y = x + y

> :type add’
add’ :: Int -> Int -> Int
> :type add’ 1
add’ 1 :: Int -> Int
> :type add’ 1 2
add’ 1 2 :: Int

mult :: Int -> (Int -> (Int -> Int))
mult x y z = x * y * z
```

Wyrażenie **mult x y z** oznacza to samo co **((mult x) y) z**.

Typy polimorficzne

```
> length [1,2,3,4,5]
5
> length [”yes”, ”no”]
2

length :: [a] -> Int
```

Zmienna typu a oznacza listę dowolnego typu.
Typ polimorficzny zawiera co najmniej jedną zmienną typu:

```
fst :: (a,b) -> a
head :: [a] -> a
tail :: [a] -> [a]
take :: Int -> [a] -> [a]
zip :: [a] -> [b] -> [(a,b)]
id :: a -> a
```

Przeciążanie

```
> 1 + 2
3
> 1.0 + 2.0
3.0

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

Num a wskazuje, że typ **a** powinien być typem numerycznym.

```
(*) :: Num a => a -> a -> a
negate :: Num a => a -> a
abs :: Num a => a -> a
```

Podstawowe klasy typów

- Eq — typy z równością
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  Bool, Char, String, Int, Integer, Float, Double, listy i krotki są klasy Eq.
- Ord — typy porządkowe
  (<), (<=), (>), (>=), min, max
- Show — typy prezentowalne
- Read — typy odczytywalne
- Num — typy numeryczne
  (+), (*), negate, abs
- Integral — typy całkowite
  div, mod
  Int i Integer są klasy Integral.
- Fractional — typy ułamkowe
  (/), recip

```
show :: a -> String

> show False
"False"
> show 'a'
"'a'"
> show 123
"123"
> show [1,2,3]
"[1,2,3]"
> show ('a',False)
"('a',False)"
```

```
read :: String -> a

> read "False" :: Bool
False
> read "'a'" :: Char
'a'
> read "123" :: Int
123
> read "[1,2,3]" :: [Int]
[1,2,3]
```

Definiowanie funkcji

```
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0

splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)

recip :: Fractional a => a -> a
recip n = 1 / n
```

Wyrażenie warunkowe

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n

signum :: Int -> Int
signum n = if n < 0 then -1
  else
    if n == 0 then 0
      else 1
```

Równania z wartownikiem

```
abs n | n >= 0    = n
      | otherwise = -n

signum n | n < 0     = -1
         | n == 0    = 0
         | otherwise = 1
```

Dopasowanie wzorca

```
not :: Bool -> Bool
not True = False
not False = True

(&&) :: Bool -> Bool - Bool
True && True   = True
True && False  = False
False && True  = False
False && False = False

True && True = True
_ && _ = False

True && b = b
False && _ = False
```

```
b && b = b
_ && _ = False
```

to nie jest poprawne
(zmienna **b** dwa razy
po lewej stronie równania)

```
b && c | b == c = b
       | otherwise = False
```

to jest poprawne

Wzorce na krotkach

```
fst :: (a,b) -> a
fst (x,_) = x
snd :: (a,b) -> b
snd (_,y) = y
```

Wzorce na listach

```
head :: [a] -> a
head (x:_) = x
tail :: [a] -> [a]
tail (_:xs) = xs
```

Lambda wyrażenia

```
\x -> x + x

> (\x -> x + x) 2
4

add :: Int -> Int -> Int
add x y = x + y

add :: Int -> (Int -> Int)
add = \x -> (\y -> x + y)

const :: a -> b -> a
const x _ = x

const :: a -> (b -> a)
const x = \_ -> x
```

```
odds :: Int -> [Int]
odds n = map f [0..n-1]
    where f x = x*2+1

odds :: Int -> [Int]
odds n = map (\x -> x*2+1) [0..n-1]
```

Wyrażenia listowe

Podstawy

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

> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,4),(3,5)]

> [(x,y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

```
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]

firsts :: [(a,b)] -> [a]
firsts ps = [x | (x,_) <- ps]

length :: [a] -> Int
length xs = sum [1 | _ <- xs]
```

Wartownicy

```
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]

> factors 15
[1,3,5,15]

> factors 7
[1,7]

prime :: Int -> Bool
prime n = factors n == [1,n]

> prime 15
False

>prime 7
True
```

Funkcja zip

```
> zip ['a','b','c'] [1,2,3,4]
[('a',1),('b',2),('c',3)]

pairs :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)

> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]

sorted :: Ord a => [a] -> Bool
sorted xs = and [x <= y | (x,y) <- pairs xs]

> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs = [i | (x',i) <- zip xs [0..], x == x']

> positions False [True,False,True,False]
[1,3]
```

Deklarowanie typów

Wprowadzenie nowej nazwy na istniejący typ

```
type String = [Char]

type Pos = (Int, Int)
type Trans = Pos -> Pos

type Tree = (Int, [Tree])
```

to nie jest poprawne
(deklaracja typu nie może być rekurencyjna)

Deklaracja parametryzowana

```
type Pair a = (a, a)

type Assoc k v = [(k, v)]

find :: Eq k => k -> Assoc k v -> v
find k t = head [v | (k’,v) <- t, k == k’]
```

Deklarowanie danych

```
data Move = North | South | East | West

move :: Move -> Pos -> Pos
move North (x,y) = (x,y+1)
move South (x,y) = (x,y-1)
move East  (x,y) = (x+1,y)
move West  (x,y) = (x-1,y)

moves :: [Move] -> Pos -> Pos
moves [] p = p
moves (m:ms) p = moves ms (move m p)
```

```
data Shape = Circle Float | Rect Float Float
```

Circle i **Rect** są konstruktorami

```
square :: Float -> Shape
square n = Rect n n

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y

> :type Circle
Circle :: Float -> Shape
> :type Rect
Rect :: Float -> Float -> Shape
```

Deklaracja parametryzowana

```
data Maybe a = Nothing | Just a

savediv :: Int -> Int -> Maybe Int
savediv _ 0 = Nothing
savediv m n = Just (m `div` n)

savehead :: [a] -> Maybe a
savehead [] = Nothing
safehead xs = Just (head xs)
```

Typy rekursywne

```
data Nat = Zero | Succ Nat

nat2int :: Nat -> Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n - 1))

add :: Nat -> Nat -> Nat
add m n = int2nat (nat2int m + nat2int n)

add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ m) n = Such (add m n)
```

```
data List a = Nil | Cons a (List a)

len :: List a -> Int
len Nil = 0
len (Cons _ xs) = 1 + len xs

data Tree a = Leaf a | Node (Tree a) a (Tree a)

occurs :: Eq a => a -> Tree a -> Bool
occurs x (Leaf y) = x == y
occurs x (Node l y r) = x == y ||
    occurs x l ||
    occurs x r

flatten :: Tree a -> [a]
flatten (Leaf x) = [x]
flatten (Node l x r) = flatten l ++ [x] ++ flatten [r]

occurs :: Ord a => a -> Tree a -> Bool
occurs x (Leaf y) = x == y
occurs x (Node l y r) | x == y    = True
                      | x < y     = occurs x l
                      | otherwise = occurs x r
```

Deklarowanie klas

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
```

- Aby typ **a** był przykładem klasy **Eq** musi obsługiwać równość i nierówność.
- Domyślna definicja dla **/=** została zawarta, zatem deklaracja przykładu wymaga tylko zdefiniowania **==**.

```
instance Eq Bool where
    False == False = True
    True  == True  = True
    _     == _     = False
```

Klasy można rozszerzać do nowych klas:

```
class Eq a => Ord a where
    (<), (<=), (>), (>=) :: a -> a -> Bool
    min, max             :: a -> a -> a

    min x y | x <= y = x
            | otherwise = y

    max x y | x <= y = y
            | otherwise = x

instance Ord Bool where
    False < True = True
    _     < _    = False
    b <= c = (b < c) || (b == c)
    b > c  = c < b
    b >= c = c <= b
```

Monady

Funktory

```
inc :: [Int] -> [Int]
inc [] = []
inc (n:ns) = n+1 : inc ns

sqr :: [Int] -> [Int]
sqr [] = []
sqr (n:ns) = n^2 : sqr ns

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

inc = map (+1)
sqr = map (^2)
```

Ogólnie: mapować funkcję po każdym elemencie struktury danych (nie tylko po listach).

Klasa typów, które umożliwiają mapowanie po strukturze nazywa się **funktorami**.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

typ parametryczny

Przykład:

```
instance Functor [] where
    -- fmap :: (a -> b) -> [a] -> [b]
    fmap = map
```

```haskell
data Maybe a = Nothing | Just a

instance Functor Maybe where
    -- fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap _ Nothing = Nothing
    fmap g (Just x) = Just (g x)

> fmap (+1) Nothing
Nothing
> fmap (*2) (Just 3)
Just 6
> fmap not (Just False)
Just True
```

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
    deriving Show

instance Functor Tree where
  -- fmap :: (a -> b) -> Tree a -> Tree b
  fmap g (Leaf x) = Leaf (g x)
  fmap g (Node l r) = Node (fmap g l) (fmap g r)

> fmap length (Leaf ”abc”)
Leaf 3
> fmap even (Node (Leaf 1) (Leaf 2))
Node (Leaf False) (Leaf True)
```

```
inc :: Functor f => f Int -> f Int
inc = fmap (+1)

> inc (Just 1)
Just 2
> inc [1,2,3,4,5]
[2,3,4,5,6]
> inc (Node (Leaf 1) (Leaf 2))
Node (Leaf 2) (Leaf 3)
```

Aplikacje

```
fmap0 :: a -> f a
fmap1 :: (a -> b) -> fa -> f b
fmap2 :: (a -> b -> c) -> f a -> f b -> fc
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
…
```

nieskończenie wiele przypadków!!!

```
> fmap2 (+) (Just 1) (Just 2)
Just 3
```

Skorzystamy z **curring**:

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b

g <*> x <*> y <*> z ≡ ((g <*> x) <*> y) <*> z
```

Typowe użycie **pure** i **<*>** (*styl aplikacyjny*):

```
pure g <*> x1 <*> x2 … <*> xn

 fmap0 = pure
 fmap1 g x = pure g <*> x
 fmap2 g x y = pure g <*> x <*> y
 fmap3 g x y z = pure g <*> x <*> y <*> z
 …
```

Klasa aplikacji:

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just g) <*> mx = fmap g mx

> pure (+1) <*> Just 1
Just 2
> pure (+) <*> Just 1 <*> Just 2
Just 3
> pure (+) Nothing <*> Just 2
Nothing
```

```
instance Applicative [] where
    -- pure :: a -> [a]
    pure x = [x]
    -- (<*>) :: [a -> b] -> [a] -> [b]
    gs <*> xs = [g x | g <- gs, x <- xs]

> pure (+1) <*> [1,2,3]
[2,3,4]
> pure (+) <*> [1] <*> [2]
[3]
> pure (*) <*> [1,2] <*> [3,4]
[3,4,6,8]
```

W powyższych przykładach, typ **[a]** rozumiemy jako uogólnienie typu **Maybe a** umożliwiające wielokrotne wyniki w przypadku powodzenia.

Lista pusta może wówczas oznaczać niepowodzenie.

```
prods :: [Int] -> [Int] -> [Int]
prods xs ys = [x*y | x <- xs, y <- ys]

prods :: [Int] -> [Int] -> [Int]
prods xs ys = pure (*) <*> xs <*> ys
```

Dzięki stylowi aplikacyjnemu dla list możliwe jest pisanie programów **niedeterministycznych**, w których możemy stosować czyste funkcje do wielowartościowych argumentów bez potrzeby obsługi wyboru wartości albo propagowania niepowodzenia.

```
instance Applicative IO where
    -- pure :: a -> IO a
    pure = return
    -- (<*>) :: IO (a -> b) -> IO a -> IO b
    mg <*> mx = do { g <- mg; x <- mx;
                     return (g x)}
```

Przykład: czytanie n znaków

```
getChars :: Int -> IO String
getChars 0 = return []
getChars n = pure (:) <*> getChar <*> getChars (n-1)
```

Monady

```
data Expr = Val Int | Div Expr Expr

eval :: Expr -> Int
eval (Val n) = n
eval (Div x y) eval x `div` eval y

> eval (Div (Val 1) (Val 0))
*** Exception : divide by zero

safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv n m = Just (n `div` m)
```

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval x of
    Nothing -> Nothing
    Just n -> case eval y of
        Nothing -> Nothing
        Just m -> safediv n m
```

wiele przypadków aby propagować niepowodzenie

to nie jest poprawne!!!
safediv jest typu Int -> Int -> Maybe Int
potrzebny jest typ Int -> Int -> Int

```
eval :: Expr -> Maybe Int
eval (Val n) = pure n
eval (Div x y) = pure safediv <*> eval x <*> eval y
```

Jak zapisać **eval** by było poprawne?

Użyjemy operatora >>= (*bind*):

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
mx >>= f = case mx of
    Nothing -> Nothing
    Just x -> f x

eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) =
    eval x >>= \n ->
        eval y >>= \m ->
            safediv n m
```

Wyrażenie:

```
m1 >>= \x1 ->
m2 >>= \x2 ->
…
mn >>= \mn ->
f x1 x2 … xn
```

można w Haskellu zapisać prościej:

```
do x1 <- m1
   x2 <- m2
   …
   xn <- mn
   f x1 x2 … xn

eval :: Expr -> Mayby Int
eval (Val n) = Just n
eval (Div x y) do n <- eval x
                  m <- eval y
                  safediv n m
```

Klasa monad:

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    return = pure
```

Przykłady:

```
instance Monad Maybe where
    -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Nothing >>= _ = Nothing
    (Just x) >>= f = f x

instance Monad [] where
    -- (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = [y | x <- xs, y <- f x]

pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = do x <- xs
                 y <- ys
                 return (x,y)
> pairs [1,2] [3,4]
[(1,3),(1,4),(2,3),(2,4)]
```