Języki i paradygmaty programowania

Programowanie funkcyjne – Haskell

Marcin Szpyrka
Krzysztof Balicki

Katedra Informatyki Stosowanej AGH
Interdyscyplinarne Centrum Modelowania Komputerowego UR

2018

Literatura

1. Graham Hutton: Programming in Haskell. Cambridge University Press 2007
2. Bryan O'Sullivan, Don Stewart, John Goerzen: Real World Haskell. O'Reilly Media 2008 http://book.realworldhaskell.org
3. Paul Hudak: The Haskell Scholl of Expressions. Learning Functional Programing through Multimedia. Cambridge University Press 2007
4. Kees Doets, Jan van Eijck: The Haskell Road to Logic, Math and Programming. King's College Publications 2004 (pdf)
5. Joeren Fokker: Functional Programming. Department of Computer Science, Utrecht University 1995 (pdf)
6. Hal Daume III, et. al.: Yet Another Haskell Tutorial. 2004 (pdf)
7. http://www.haskell.org
8. http://www.haskell.org/ghc/docs/latest/html/users_guide/ghci.html

Haskell – wprowadzenie

- Haskell jest nowoczesnym, leniwym, czysto-funkcyjnym językiem programowania, nazwanym tak na cześć amerykańskiego matematyka Haskella Curry'ego.
- Termin leniwy oznacza, że wartość żadnego wyrażenia nie jest wyznaczana dopóki nie jest potrzebna. – Umożliwia to na przykład stosowanie niekończącej się rekurencji, przy czym wyznaczane są tylko te wartości, które są aktualnie potrzebne. Dla porównania w językach ścisłych (np. C, C++, Java itd.) wyznacza jest wartość każdego wyrażenia, niezależnie od tego czy jest to potrzebne, czy nie.
- Termin czysty oznacza, że jest to język bez efektów ubocznych. – Za efekty uboczne wykonania funkcji uważamy wszystkie operacje przez nią wykonywane nie związane bezpośrednio z obliczeniem jej wartości, np. wypisanie komunikatu na ekranie, modyfikacja globalnej zmiennej itp.
- Haskell jest językiem stosującym silne typowanie. – Niemożliwa jest na przykład automatyczna (przypadkowa) konwersja `Double` do `Int`.
  Ponadto w Haskellu typy są automatycznie rozpoznawane, co oznacza, że bardzo rzadko konieczne jest deklarowanie typu funkcji.
- Haskell stosuje system monad do oddzielenia *nieczystych* operacji (np. operacji wejścia-wyjścia) od reszty programu.

Kompilator, praca interaktywna

Do pracy z językiem Haskell będziemy używać kompilator GHC – the Glasgow Haskell Compilation system. W systemie GNU/Linux (Ubuntu) wymaga to instalacji pakietu ghc.
Kod napisany w języku Haskell może być wykonywany w trybie wsadowym (kompilowanie) lub interaktywnym (interpretowanie).

Praca interaktywna – przykład sesji:

```
$ ghci
GHCi, version 7.6.2: http://www.haskell.org/ghc/  :? for help
Loading package ...
Prelude> "Haskell is fun"
"Haskell is fun"
Prelude> (23-17)*2^7
768
Prelude> sqrt 3
1.7320508075688772
Prelude> let f x = x^3 -2*x + 3
Prelude> f 4
59
Prelude> :q
Leaving GHCi.
$
```

Podstawy programowania – funkcje

- Program w Haskellu składa się z jednej lub więcej funkcji. Podczas wykonywania programu funkcji dostarczane są argumenty i wyznaczana jest jej wartość.
- W Haskellu rozróżniane są małe i wielkie litery. Nazwy funkcji i parametrów muszą zaczynać się z małej litery. Ponadto dozwolone jest stosowanie liter, cyfr, pojedynczego apostrofu i symbolu podkreślenia. Wyjątek stanowią funkcje konstruujące, których nazwy zaczynają się z wielkiej litery. Ponato z wielkiej litery piszemy nazwy typów.
- Funkcje można definiować bezpośrednio podczas sesji interaktywnej stosując polecenie `let`.
- Zazwyczaj definicje funkcji umieszczane są w plikach o rozszerzeniu hs. Podczas pracy z interpreterem, definicje zawarte w pliku wczytuje się stosując polecenie `:load` (skrót `:l`), którego parametrem jest nazwa wczytywanego pliku. Ponowne wczytanie ostatnio załadowanego pliku (np. po zmianach jego zawartości) realizowane jest przez polecenie `:reload` (skrót `:r`).
- W Haskellu nie występują zmienne. Można definiować funkcje bezargumentowe, np.: `e = 2.718282`, które można traktować jak stałe.
- W programach pisanych w Haskellu dostępne są bezpośrednio funkcje i operatory (jest ich w sumie ponad 200) zdefiniowane w pliku prelude.hs.

Prelude – przykłady funkcji

`abs`	wartość bezwzględna,
`pi`	liczba π,
`id`	identyczność,
`signum`	znak liczby,
`sqrt`	pierwiastek kwadratowy,
`gcd`	największy wspólny dzielnik,
`lcm`	najmniejsza wspólna wielokrotność,
`min, max`	minimum i maksimum dwóch liczb,
`sin, cos, tan`	funkcje trygonometryczne,
`asin, acos, atan`	funkcje cyklometryczne,
`log`	logarytm naturalny,
`logBase`	$\log_a b$,
`exp`	e^x
`round`	zaokrąglenie liczby zmiennoprzecinkowej do całkowitej,
`truncate`	obcięcie części ułamkowej,
`ceiling, floor`	funkcje *sufit* i *podłoga*,
`not`	negacja,
`even`	sprawdzenie, czy liczba jest parzysta.

Operatory

W Haskellu operator jest to funkcja przyjmująca dwa argumenty, której nazwa jest umieszczana pomiędzy argumentami, a nie przed nimi. Ponadto nazwa operatora składa się z jednego lub większej liczby symboli.

Symbole, z których można budować operatory

`: # % & * + - = . / \ < > ? ! ^ | @`

- Operator umieszczony w nawiasach okrągłych zachowuje się jak zwykła funkcja – umieszczany jest przed argumentami.
- Funkcja dwuargumentowa umieszczona w odwrotnych apostrofach zachowuje się jak operator – umieszczana jest między argumentami.

```
Prelude> (*) 2 6
12
Prelude> 8 `gcd` 12
4
```

Operatory predefiniowane – priorytety

Po lewej priorytet, inaczej moc wiązania, dla operatorów po prawej:

9 `.` (składanie funkcji), `!!` (pobranie elementu z listy)

8 `^`, `^^`, `**`

7 `*`, `/`, `` `quot` ``, `` `rem` ``, `` `div` ``, `` `mod` ``

6 `+`, `-`

5 `:` (dodanie elementu do listy), `++` (konkatenacja list)

4 `==`, `/=`, `<`, `<=`, `>`, `>=`, `` `elem` `` (należy do listy), `` `notElem` `` (nie należy do listy)

3 `&&`

2 `||`

1 `>>`, `>>=`

0 `$`, `$!`, `` `seq` ``

Najwyższy priorytet ma wywołanie funkcji. Wyrażenie `sqrt 3 + 4` jest zatem traktowane jako $\sqrt{3} + 4$.

Konstruując wyrażenia można stosować nawiasy okrągłe, co pozwala precyzyjnie określić kolejność wykonywania poszczególnych działań.

Definiowanie modułu – przykład

```
-- plik functions.hs
module Functions where

f x = x^2 -3*x + 1
g x = sin x + sqrt x
h x y = x^2 + y^2 - pi*x*y

{- nie podajemy tutaj definicji funkcji pi.
   bo jest ona zdefiniowana w pliku prelude.hs -}
```

- Pliki z kodem mogą zawierać dwa typy komentarzy: od znaku `--` do końca linii oraz wieloliniowe umieszczone między znakami `{-` i `-}`.
- Definicja funkcji składa się z: nazwy funkcji, nazwy parametrów formalnych, znaku = i wyrażenia określającego jak należy wyznaczyć wartość funkcji.

```
Prelude> :l functions
[1 of 1] Compiling Functions        ( functions.hs, interpreted )
Ok, modules loaded: Functions.
*Functions> f 2
-1
*Functions> h 1 2
-1.2831853071795862
```

Definiowanie funkcji

Nową funkcję możemy zdefiniować stosując funkcje biblioteczne i funkcje wcześniej zdefiniowane. Definicje funkcji `f2` i `f3` są równoważne. W obu przypadkach definiujemy w zasadzie alias dla funkcji **`sqrt`**. W takim przypadku można pominąć parametry formalne. **Uwaga:** Definicja `f1 = sin + cos` jest **niepoprawna**!

```
f1 x = sin x + cos x
f2 y = sqrt y
f3 = sqrt
```

Wywołanie funkcji z mniejszą liczbą argumentów aktualnych niż jej liczba argumentów formalnych nazywane jest częściową aplikacją (ang. currying). Wynikiem takiego wywołania jest funkcja, która ma co najmniej jeden argument formalny.

```
h1 x = mod 12 x
h2 = mod 12           -- równoważne h1
h3 x = mod x 12       -- tutaj nie możemy pominąć x
h4 = (`mod` 12)       -- równoważne h3 -> patrz następny slajd
```

Sekcje

Dla częściowo zaaplikowanych operatorów dostępne są dwie notacje zwane sekcjami:

- $(\oplus\ a)$ – operator $\oplus$ jest częściowo zaaplikowany wartością a z prawej strony
- $(a\ \oplus)$ – operator $\oplus$ jest częściowo zaaplikowany wartością a z lewej strony

```
successor = (+ 1)
double    = (2 *)
half      = (/ 2.0)
revNumber = (1.0 /)
square    = (^ 2)
twoPower  = (2 ^)
oneDigit  = (<= 9)
isZero    = (== 0)
```

```
-- successor 34     35
-- double 45        90
-- half 5           2.5
-- revNumber 4      0.25
-- square 34        1156
-- twoPower 12      4096
-- oneDigit 7       True
-- isZero 9         False
```

Uwaga: Powyższe funkcje można zdefiniować nie stosując sekcji i jawnie podając parametry formalne, np.

```
successor' x = x + 1
double' x = 2 * x
```

Typy danych

Każde wyrażenie w Haskellu ma swój typ. Fakt, że wartość `v` ma typ `T` zapisujemy jako `v::T`. Interpreter sam jest w stanie określić typ dowolnego wyrażenia. Możliwe jest również jawne określenie typu funkcji (choć nie jest to konieczne). Deklaracja taka pełni jednak m.in. funkcję komentarza pozwalającego łatwiej zrozumieć kod.
Podczas pracy z interpreterem można sprawdzić typ wyrażenia wywołując dla niego polecenie `:type` (skrót `:t`).

```
*Functions> :t 5
5 :: Num a => a                    -- wartość dowolnego typu klasy Num
*Functions> :t "Ala ma kota."
"Ala ma kota." :: [Char]
*Functions> :t f1
f1 :: Floating a => a -> a         -- funkcja a -> a, gdzie a należy
                                   -- do klasy Floating
```

Polimorfizm w języku Haskell jest realizowany poprzez zastosowanie tzw. klas typów, które są zbiorem warunków, które musi spełnić typ danych, aby mógł przynależeć do danej klasy. Funkcje można definiować tak, aby były określone nie dla pojedynczych typów, ale dla klas typów. Funkcja taka może zostać wówczas wywołana dla argumentów dowolnego typu przynależnego do danej klasy.

Podstawowe typy danych i klasy

- Typy całkowite: Int, Int8, Int16, Int32, Int64, Integer (typ nieograniczony), Word, Word8, Word16, Word32, Word64 (typy unsigned);
- Typy zmiennoprzecinkowe: Float, Double;
- Typ stałoprzecinkowy: Rational (ułamki zwykłe);
- Typ logiczny: Bool;
- Typ znakowy: Char;
- Typ tekstowy: String – lista znaków `[Char]`;
- Listy (nawiasy kwadratowe);
- Krotki (nawiasy okrągłe);
- Funkcje (->).

```
g :: Int -> Int                 -- argument typu Int, wynik typu Int
g x = x^3

h :: Float -> Int -> Float      -- argumenty typu Float i Int,
h x n = x^n                     -- wynik typu Float
```

Uwaga: Biorąc pod uwagę częściową aplikację, na funkcję `h` można patrzeć jak na funkcję, która przyjmuje argument typu `Float` i zwraca funkcję typu `Int -> Float`.

Podstawowe klasy typów

- Show – klasa typów zawierających funkcje konwertujące ich wartości na typ String (lista znaków);
- Read – klasa dualna do Show, zawiera typy, których wartości mogą być konwertowana z typu String.
- Eq – klasa typów, których elementy można porównywać;
- Num – klasa typów numerycznych;
- Integral – klasa typów całkowitoliczbowych;
- Floating – klasa typów zmiennoprzecinkowych;
- Fractional – klasa typów rzeczywistych (Float, Double, Rational);
- Ord – klasa typów porządkowych;

```
f :: Num a => a -> a      -- funkcja polimorficzna
f x = x^2 -3*x + 1
```

Konwersje

Konwersje między wartościami różnych typów **muszą** być wykonywane **jawnie**.

- fromIntegral – konwersja dowolnego typu klasy Integral na dowolny typ numeryczny;
- toRational – konwersja typów liczbowych na ułamki zwykłe;
- fromRational – konwersja ułamków zwykłych na liczby zmiennoprzecinkowe;
- truncate, round, ceiling, floor – konwersja typów liczbowych na typy całkowitoliczbowe.

```
Prelude> :m +Data.Ratio          -- moduł do obsługi ułamków zwykłych
Prelude Data.Ratio> fromRational (2 % 7)
0.2857142857142857
Prelude Data.Ratio> truncate 34.6
34
Prelude Data.Ratio> round 34.6
35
Prelude Data.Ratio> toRational 34.5
69 % 2
```

Krotki i listy

Krotki są elementami zbiorów będących iloczynami kartezjańskimi zbiorów. Krotki są zapisywane z użyciem nawiasów okrągłych. Poszczególne zbiory będące czynnikami w iloczynie kartezjańskim mogą być różnych typów.

Listy można traktować jako ciąg wartości określonego typu, które umieszczone są wewnątrz nawiasów kwadratowych, np. `[]`, `[1,2,4,17]`, `[True, False, True]`, `[sin, cos, tan]`, `[[1,2,3], [1,2]]`. W Haskellu można pracować również z listami nieskończonymi.

```
d2 :: (Double, Double) -> Double  -- argumentem jest para liczb
d2 (x,y) = sqrt (x^2 + y^2)       -- typu Double

d3 :: (Double, Double, Double) -> Double
d3 (x,y,z) = sqrt (x^2 + y^2 + z^2)

d :: [Double] -> Double           -- argumentem jest lista liczb
d [x,y] = sqrt (x^2 + y^2)        -- typu Double
d [x,y,z] = sqrt (x^2 + y^2 + z^2)
```

Instrukcje warunkowe

```
sgn x  = if x < 0 then -1
         else if x > 0 then 1
         else 0

-- sgn (-9)
```

W Haskellu instrukcja **`if`** **musi** zawierać klauzulę **`else`**.

```
f x =
  case x of
    0 -> 1
    1 -> 5
    2 -> 2
    _ -> -1
```

Znak podkreślenia pełni w powyższej definicji rolę **dżokera** – zastępuje dowolną inną wartość. Do wcinania kodu należy używać spacji, a nie znaków tabulacji. Zastosowane wcięcia mają znaczenie dla interpretacji kodu.

Haskell – organizacja kodu

Układ kodu w Haskellu ma istotne znaczenie dla jego interpretacji. Obowiązuje tutaj następująca zasada:

Otwierająca klamra jest domyślnie wstawiana po słowach kluczowych: where, let, do oraz of i zapamiętywana jest pozycja od której rozpoczyna się kolejne polecenie. Od tego momentu każda linia o takim samym wcięciu kończy się domyślnie średnikiem. Jeżeli wcięcie kolejnej linii jest mniejsze, to wcześniej automatycznie jest wstawiana klamra zamykająca.

Zasadę tę można pominąć, jeśli jawnie będziemy wpisywać średniki i klamry. W takim przypadku układ kodu nie ma już znaczenia.

```
f x = case x of { 0 -> 1; 1 -> 5; 2 -> 2; _ -> -1}

-- lub

f x = case x of { 0 -> 1;
      1 -> 5; 2 -> 2
  ; _ -> -1}
```

Definiowanie funkcji przez określanie przypadków

- Zastosowanie poszczególnych przypadków jest określane przez **warunki** umieszczane bezpośrednio po znaku |.
- Przypadki są przeglądane od góry do dołu. Stosowany jest pierwszy napotkany przypadek, dla którego warunek ma wartość True.
- Jako warunek dla ostatniego przypadku można użyć wartości True lub stałej otherwise.

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

f x | x < 0      = x^2
    | otherwise = x

g x | x <= 0 = 0
    | x > 0  = 2 * x
    | x > 3  = x^3     -- ten przypadek nigdy nie zostanie użyty

silnia :: Integer -> Integer
silnia n | n == 0 = 1
         | n > 0  = n * silnia (n - 1)
```

Definiowanie funkcji przez części

Funkcję można definiować podając poszczególne **części** jej definicji dla różnych wartości argumentu. Podobnie jak przy określaniu przypadków, również tutaj istotna jest kolejność w jakiej ułożone zostaną poszczególne definicje.

```
f 0 = 1
f 1 = 5
f 2 = 2
f _ = -1
```

UWAGA: Jeżeli nie zostanie podana definicja z linii (4), to kompilator będzie zgłaszał błąd dla argumentów innych niż 0, 1 i 2.

```
power :: Float -> Int -> Float
power a 1 = a
power a b = a * power a (b - 1)
```

Rekurencja

```
silnia :: Integer -> Integer
silnia n | n == 0 = 1
         | n > 0  = n * silnia (n - 1)

fib :: Integer -> Integer
fib n | n == 0 = 0
      | n == 1 = 1
      | n > 1  = fib (n - 1) + fib (n - 2)
-- bardzo kosztowna rekurencja

power :: Float -> Int -> Float
power a 1 = a
power a b = a * power a (b - 1)

f x = f x
```

UWAGA: Ostatnia definicja jest poprawna z punktu widzenia składni języka, ale wywołanie funkcji `f` rozpoczyna nieskończone obliczenia.

Parametry formalne

Podobnie jak w innych językach programowania, również w Haskellu wyróżnia się parametry formalne – z którymi funkcja jest definiowana i parametry aktualne – z którymi jest wywoływana. W przeciwieństwie do innych języków programowania, parametry formalne w Haskellu nie muszą być wyłącznie nazwami, dopuszczalne jest również używanie wzorców.

Jako wzorzec można użyć:

- liczby,
- stałe True lub False,
- nazwy (np. `x`),
- listy, których elementy są wzorcami,
- krotki, których elementy są wzorcami,
- operator : (dwukropek) ze wzorcami po swojej prawej i lewej stronie.

Wzorce – przykłady

```
nie True  = False
nie False = True

i True True = True
i _ _       = False

lub True _ = True
lub _ True = True
lub _ _    = False

fst :: (a,b) -> a      -- prelude
fst (x,_) = x

snd :: (a,b) -> b      -- prelude
snd (_,y) = y

head :: [a] -> a       -- prelude
head (x:_) = x

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

Podwyrażenia

Definiując funkcje można stosować podwyrażenia, definiowane po słowie kluczowym where. Zastosowanie podwyrażeń pozwala m.in. na uniknięcie wielokrotnego wyznaczania wartości tych samych formuł.

Zamiast konstrukcji where można użyć również let ... in ...

```
trojmian (a, b, c) = if d > 0 then [ (-b + m)/n, (-b - m)/n ]
                     else if d == 0 then [ (-b + m)/n ]
                     else []
                     where d = b ^ 2 - 4.0 * a * c
                           m = sqrt d
                           n = 2.0 * a

trojmian' (a, b, c) = let
                        d = b ^ 2 - 4.0 * a * c
                        m = sqrt d
                        n = 2.0 * a
                      in if d > 0 then [ (-b + m)/n, (-b - m)/n ]
                         else if d == 0 then [ (-b + m)/n ]
                         else []
```

Składanie funkcji

Do składania funkcji można stosować **operator złożenia** `.` (kropka). Jest on odpowiednikiem matematycznego operatora składania funkcji $f \circ g$.

```
(sqrt.abs)(-4)              -- równoważne sqrt (abs (-4))
                            -- Uwaga: w zapisie sqrt (abs (-4))
                            -- konieczne obie pary nawiasów!

(sqrt.fromIntegral.silnia) 5   -- równoważne
                               -- sqrt (fromIntegral( silnia 5))

-- w x = x^3 - 2*x + 1

-- (w.sqrt) 3      -- 2.7320508075688767
-- (sqrt.w) 3      -- 4.690415759823 43
-- (sin.sqrt) pi   -- 0.979735932475 8174
-- (sqrt.sin) pi   -- 1.106619335554 1812e-8
```

Łączność operatorów

Dla operatora określa się rodzaj wiązania, które decyduje o kolejności obliczeń, jeśli operator jest użyty w wyrażeniu więcej niż raz, np. `8/4/2`. Możliwe są trzy przypadki:

- Operator jest lewostronnie łączny, jeśli $a \oplus b \oplus c \equiv (a \oplus b) \oplus c$,
  np. operatory: `+`, `-`, `*`, `/`, `` `quot` ``, `` `rem` ``, `` `div` ``, `` `mod` ``, `!!`, `>>`, `>>=`.
- Operator jest prawostronnie łączny, jeśli $a \oplus b \oplus c \equiv a \oplus (b \oplus c)$,
  np. operatory: `.`, `^`, `^^`, `**`, `:`, `++`, `||`, `&&`, `$`, `$!`, `` `seq` ``.
- Operator nie jest łączny, jeśli $a \oplus b \oplus c$ jest niepoprawne składniowo i wymagane jest użycie nawiasów okrągłych wskazujących kolejność obliczeń,
  np. operatory: `==`, `/=`, `<`, `>`, `<=`, `>=`, `` `elem` ``, `` `notElem` ``.

W algebrze rozważa się również operatory łączne:

- Operator jest łączny, jeśli $a \oplus b \oplus c \equiv (a \oplus b) \oplus c \equiv a \oplus (b \oplus c)$,
  uwaga: w języku Haskell nie można definiować operatorów łącznych.

Definiowanie operatorów – przykłady

```
infix 5 !^!
(!^!) :: Int -> Int -> Int
n !^! k = silnia n `div` (silnia k * silnia (n - k))

infix 1 ==>
(==>) :: Bool -> Bool -> Bool
x ==> y = (not x) || y
-- lub
(==>) :: Bool -> Bool -> Bool
True  ==> x = x
False ==> x = True

infix 1 <=>
(<=>) :: Bool -> Bool -> Bool
x <=> y = x == y

infixr 2 <+>
(<+>) :: Bool -> Bool -> Bool
x <+> y = x /= y
```

Linie typu `infix 5 !^!` określają regułę łączności i priorytet dla operatora:

- infixr – łączność prawostronna,
- infixl – łączność lewostronna
- infix – brak łączności.

Funkcje wyższych rzędów

W funkcyjnych językach programowania funkcje są często traktowane tak samo jak inne wartości np. liczby, listy itp, np.:

- Funkcja ma określony swój typ.
- Funkcja może być wynikiem innej funkcji.
- Funkcja może być argumentem innej funkcji.

Ostatnia z przedstawionych możliwości oznacza, że można definiować **funkcje wyższego rzędu**, których zachowanie jest zdeterminowane przez funkcje będące ich argumentami.

```
until :: (a->Bool) -> (a->a) -> a -> a          -- prelude
until p f x
  | p x = x
  | otherwise = until p f (f x)
```

Funkcja `until` implementuje iterację. Jej parametry to: funkcja definiująca własność jaką powinien spełniać końcowy wynik, funkcja stosowana w każdej iteracji do korekty wyniku i wartość początkowa. Wywołanie `until p f x` należy rozumieć jako: *dopóki nie jest spełnione* `p` *stosuj funkcję* `f` *do wartości* `x`.

Obliczanie wartości funkcji $\sqrt{x}$

Wartość pierwiastka kwadratowego z liczby rzeczywistej można wyznaczać iteracyjnie według następującego algorytmu: *Jeżeli y jest przybliżeniem $\sqrt{x}$, to kolejne (lepsze) przybliżenie uzyskujemy ze wzoru $\frac{1}{2}(y + \frac{x}{y})$. Wynik obliczeń jest satysfakcjonujący jeśli y^2 różni się od x nie więcej niż założona dokładność.*

```
module Iteration ( (~=)
                 )
where

-- operator "w przybliżeniu równe"
infix 5 ~=
a ~= b = (a - b < h) && (b - a < h)
       where h = 0.000001

-- nasza wersja funkcji sqrt
root x = until goodEnough improve 1.0
       where improve y    = 0.5 * (y + x/y)
             goodEnough y = y * y ~= x
```

Obliczanie rozwiązania równania $f(x) = 0$ metodą bisekcji

```
module Bisection
where

import Iteration

zero :: (Float -> Float) -> Float -> Float -> Float
zero f a b | (f a) * (f b) > 0 = error "wrong interval"
           | (a - b) ~= 0      = a
           | (f x) ~= 0        = x
           | (f a) * (f x) < 0 = zero f a x
           | otherwise         = zero f x b
          where x = (a + b) / 2
```

```
Prelude> :l bisection iteration
[1 of 2] Compiling Iteration        ( iteration.hs, interpreted )
[2 of 2] Compiling Bisection        ( bisection.hs, interpreted )
Ok, modules loaded: Iteration, Bisection.
*Bisection> zero sin 1 5
3.141592
*Bisection> zero (sin.sqrt) 1 20
9.869602
```

Listy

Listy można traktować jako ciąg wartości określonego typu, które umieszczone są wewnątrz nawiasów kwadratowych, np. `[]`, `[1,2,4,17]`, `[True, False, True]`, `[sin, cos, tan]`, `[[1,2,3], [1,2]]`.

```
Prelude> :t []
[] :: [a]
Prelude> :t [1,2,4,17]
[1,2,4,17] :: (Num t) => [t]
Prelude> :t [True, False, True]
[True, False, True] :: [Bool]
Prelude> :t [sin, cos, tan]
[sin, cos, tan] :: (Floating a) => [a -> a]
Prelude> :t [[1,2,3], [1,2]]
[[1,2,3], [1,2]] :: (Num t) => [[t]]
```

Podstawą konstruowania list jest operator `:` (dwukropek), który dodaje element (argument umieszczony po lewej stronie operatora) na początku listy (argument po prawej stronie operatora), np.:

```
2 : [7,12]          -- [2,7,12]
2 : 4 : 7 : []      -- [2,4,7]
```

Listy

- Zastosowanie notacji przedziałowej (dwie kropki) pozwala łatwo definiować listy zawierające ciągi arytmetyczne o różnicy 1, np.:

```
[1 .. 4]          -- [1,2,3,4]
[1.5 .. 8.0]      -- [1.5,2.5,3.5,4.5,5.5,6.5,7.5,8.5]
[2 ..]            -- [2,3,4,...]
['a' .. 'f']      -- "abcdef"
```

- Dwie listy są równe, jeżeli zawierają dokładnie te same elementy, ułożone w tej samej kolejności. Listy można porównywać stosując operator ==.
- Listy można porównywać stosując operatory <, <= itd. Obowiązuje wówczas porządek leksykograficzny, np.

```
Prelude> [2,3] < [3,1]
True
Prelude> [2,3] < [2,3,4]
True
Prelude> [1,4,3] < [1,3,4]
False
```

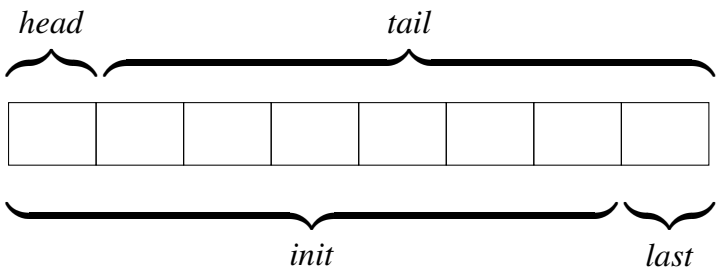
Listy – wzorce

Funkcje wykonujące operacje na listach opierają się na rekurencji.
Bardzo często lista jest zapisywana w postaci wzorca `(x:xs)`, co należy rozumieć jako `'iks' : ['iksy']`, przy czym druga część może być listą pustą. Wzorzec taki może również zawierać jako swój element znak _ jeśli nieistotna jest któraś z tych części listy.

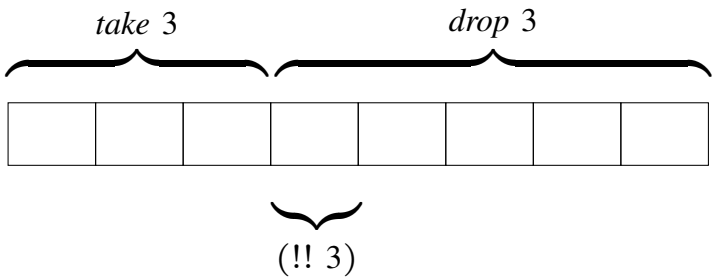
- `(x:xs)` – lista co najmniej jednoelementowa, `x` jest głową, a `xs` ogonem;
- `[]` – lista pusta;
- `[_]` – lista jednoelementowa, nie interesuje nas wartość tego elementu, ale sam fakt jego istnienia;
- `[x,y]` – lista dwuelementowa;
- `[x,_,y]` – lista trójelementowa, nie interesuje nas wartość drugiego elementu;
- `[_,_,_]` – lista trójelementowa, nie interesują nas wartości elementów;
- `(x1:x2:xs)` – lista co najmniej dwuelementowa: `x1` – pierwszy element, `x2` – drugi element, `xs` – ogon.

Pobieranie fragmentów listy (1)

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

tail :: [a] -> [a]              -- prelude
tail (_:xs) = xs

last :: [a] -> a                -- prelude
last [x]    = x
last (_:xs) = last xs

init :: [a] -> [a]              -- prelude
init [_]    = []
init (x:xs) = x : init xs

null :: [a] -> Bool             -- prelude
null []    = True
null (_:_) = False
```



Operacje na listach



Wybrane funkcje dostępne w module prelude:

- `elem x xs` – sprawdza czy element `x` należy do listy `xs`;
- `xs !! n` – pobiera `n`-ty element listy `xs`;
- `take n xs` – pobiera `n` pierwszych elementów listy `xs`;
- `drop n xs` – usuwa `n` pierwszych elementów listy `xs`;
- `length xs` – zwraca długość listy `xs`;
- `sum xs` – zwraca sumę elementów listy `xs` (dla list o wartościach numerycznych);
- `minimum xs` – zwraca najmniejszy element listy `xs` (dla list o wartościach typów porządkowych);
- `maximum xs` – zwraca największy element listy `xs` (dla list o wartościach typów porządkowych);
- `xs ++ ys` – scala listy `xs` i `ys`;
- `concat xss` – scala listę list (wynikiem jest lista elementów).

Operacje na listach – przykłady (1)

```
-- własna funkcja obliczająca długość listy
len :: [a] -> Int
len []     = 0                  -- warunek zakończenia
len (_:xs) = 1 + len xs         -- ważna kolejność części!

-- usuwamy pierwsze wystąpienie elementu y z listy (x:xs)
removeFirst :: (Eq a) => a -> [a] -> [a]
removeFirst y [] = []
removeFirst y (x:xs) | y == x    = xs
                     | otherwise = x : (removeFirst y xs)

-- usuwamy wszystkie wystąpienia elementu y z listy (x:xs)
remove :: (Eq a) => a -> [a] -> [a]
remove y [] = []
remove y (x:xs) | y == x    = remove y xs
                | otherwise = x : remove y xs

-- usuwamy duplikaty z listy (x:xs)
removeDuplicates :: (Eq a) => [a] -> [a]
removeDuplicates []     = []
removeDuplicates (x:xs) = x : removeDuplicates (remove x xs)
-- removeDuplicates "Kasia ma kota i psa"
-- "Kasi mkotp"
```

Operacje na listach – przykłady (2)

```
-- wybieramy z listy co trzeci element
select3 :: [a] -> [a]
select3 []          = []
select3 [_]         = []
select3 [_,_]       = []
select3 (_:_:x:xs) = x : (select3 xs)

-- lista złożona z n kopii elementu e
copy :: Int -> a -> [a]
copy n e | n == 0 = []
         | n > 0  = e : copy (n-1) e

blowup :: [a] -> [a]
blowup [] = []
blowup xs = blowup (init xs) ++ copy len (last xs)
          where len = length xs
-- blowup "Marcin"
-- "Maarrrccccciiiiinnnnnn"

prefix :: Ord a => [a] -> [a] -> Bool
prefix [] _ = True
prefix (_:_) [] = False
prefix (x:xs) (y:ys) = (x == y) && prefix xs ys
```

Funkcja map

Funkcja map stosuje funkcję będącą jej pierwszym argumentem do wszystkich elementów listy będącej jej drugim argumentem.

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

$$
\begin{array}{rcccccccccc}
\text{xs} & = & [& 1 & , & 2 & , & 3 & , & 4 & , & 5 &] \\
 & & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\
\text{map square xs} & = & [& 1 & , & 4 & , & 9 & , & 16 & , & 25 &]
\end{array}
$$

```
map (2 ^) [1 .. 10]
-- [2,4,8,16,32,64,128,256,512,1024]

map (> 3) [1 .. 10]
-- [False,False,False,True,True,True,True,True,True,True]
```

Funkcja filter

Funkcja filter usuwa z listy elementy nie spełniające wskazanego predykatu logicznego (elementy dla których funkcja zwraca wartość False).

```
filter :: (a -> Bool) -> [a] -> [a]              -- prelude
filter _ []     = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs
```

xs	=	[	1	,	2	,	3	,	4	,	5	]
			×		↓		×		↓		×	
filter even xs	=	[			2	,			4			]

```
filter (> 3) [1 .. 10]
-- [4,5,6,7,8,9,10]

filter oneDigit [1 .. 10]
-- [1,2,3,4,5,6,7,8,9]
```

Funkcja foldr

Funkcja foldr wstawia operator pomiędzy wszystkie elementy listy zaczynając od prawej strony.

```
foldr :: (a -> b -> b) -> b -> [a] -> b      -- prelude
foldr _  e []     = e
foldr op e (x:xs) = x `op` foldr op e xs
```

$$
\begin{array}{rcccccccccccc}
\text{xs} & = & [& 1 & , & 2 & , & 3 & , & 4 & , & 5 &] \\
 & & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\
\text{foldr } (+)\ 0\ \text{xs} & = & & (1 & + & (2 & + & (3 & + & (4 & + & (5 & + 0)))))
\end{array}
$$

```
foldr (+) 0 [1 .. 10]
-- 55

foldr (-) 0 [1 .. 10]
-- -5

or = foldr (||) False
```

Funkcja foldl

Funkcja foldl wstawia operator pomiędzy wszystkie elementy listy zaczynając od lewej strony.

```
foldl :: (a -> b -> a) -> a -> [b] -> a      -- prelude
foldl _  e []     = e
foldl op e (x:xs) = foldl op (e `op` x) xs
```

$$
\begin{array}{rcccccccccccc}
\text{xs} & = & [& 1 & , & 2 & , & 3 & , & 4 & , & 5 &] \\
 & & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\
\text{foldl } (+)\ 0\ \text{xs} & = & (((((0 + & 1) & + & 2) & + & 3) & + & 4) & + & 5) &
\end{array}
$$

```
foldl (+) 0 [1 .. 10]
-- 55

foldl (-) 0 [1 .. 10]
-- -55
```

Notacja lambda

Zastosowanie notacji lambda pozwala na definiowanie funkcji anonimowych.

```
\ wzorzec -> wyrażenie
```

```
-- \ x -> x^2
-- \ y -> 2*y - 2
-- \ x y -> sin x + 2 * (cos y)
-- \ (x,y) -> sin x + 2 * (cos y)
-- \ x -> x `mod` 3 == 0

map (\ x -> x^2) [1..9]
-- [1,4,9,16,25,36,49,64,81]

map (\ y -> 2*y - 2) [1..9]
-- [0,2,4,6,8,10,12,14,16]

map (\ (x,y) -> sin x + 2 * (cos y)) [(0,0),(pi/2,pi/4),(pi,0)]
-- [2.0,2.414213562373095,2.0]

filter (\ x -> x `mod` 3 == 0) [1..30]
-- [3,6,9,12,15,18,21,24,27,30]
```

Funkcje takeWhile i dropWhile

Funkcja takeWhile działa podobnie jak filter, ale przegląda listę tylko dopóki nie napotka elementu, który nie spełnia predykatu.

```
takeWhile :: (a -> Bool) -> [a] -> [a]              -- prelude
takeWhile _ []      = []
takeWhile p (x:xs) | p x        = x : takeWhile p xs
                   | otherwise = []
```

Funkcja dropWhile usuwa początkowe elementy listy spełniające podany predykat.

```
dropWhile :: (a -> Bool) -> [a] -> [a]              -- prelude
dropWhile _ []      = []
dropWhile p (x:xs) | p x        = dropWhile p xs
                   | otherwise = x:xs
```

```
takeWhile oneDigit [5 .. 25]
-- [5,6,7,8,9]

dropWhile even [2,4,6,7,8,9]
-- [7,8,9]
```

Definiowanie list – notacja matematyczna

Definiując listy można posłużyć się notacją stosowaną w matematyce do opisywania zbiorów. Podajemy wówczas:

- wyrażenie na podstawie którego wyliczane są elementy listy,
- znak |,
- warunki jakie muszą spełniać zmienne występujące w podanym wyrażeniu.

```
[x^3 | x <- [1..8]]
-- [1,8,27,64,125,216,343,512]

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

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

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

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

Sortowanie list (1)

```
insert :: Ord a => a -> [a] -> [a]
insert e [] = [e]
insert e (x:xs) | e <= x    = e : x : xs
                | otherwise = x : insert e xs

insertionSort :: Ord a => [a] -> [a]
insertionSort = foldr insert []

insertionSort [4,3,6,9,2]
-- [2,3,4,6,9]
-- (4 `insert` (3 `insert` (6 `insert` (9 `insert` (2 `insert` [])))))

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

qsort [23,12,7,16,3,21,21,8]
-- qsort [12,7,16,3,21,21,8] ++ [23] ++ qsort []
--   qsort [7,3,8] ++ [12] ++ qsort [16,21,21] ++ [23] ++ []
--     qsort [3] ++ [7] ++ qsort [8] ++ [12]
--     ++ qsort [] ++ [16] ++ qsort [21,21] ++ [23] ++ []
-- [3,7,8,12,16,21,21,23]
```

Sortowanie list (2)

```
merge :: Ord a => [a] -> [a] -> [a]
merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) | x <= y    = x : merge xs (y:ys)
                    | otherwise = y : merge (x:xs) ys

mergeSort :: Ord a => [a] -> [a]
mergeSort xs | len <= 1  = xs
             | otherwise = merge (mergeSort ys) (mergeSort zs)
             where len  = length xs
                   half = len `div` 2
                   ys   = take half xs
                   zs   = drop half xs
-- mergeSort [18,3,13,7,8,1]
-- merge (mergeSort [18,3,13]) (mergeSort [7,8,1])
--   merge (mergeSort [18]) (mergeSort [3,13])
--   merge (mergeSort [7]) (mergeSort [8,1])
--     merge (mergeSort [18]) (merge (mergeSort [3]) (mergeSort [13]))
--     merge (mergeSort [7]) (merge (mergeSort [8]) (mergeSort [1]))
--     merge [18] (merge [3] [13])
--     merge [7] (merge [8] [1])
--   merge [18] [3,13]
--   merge [7] [1,8]
-- merge [3,13,18] [1,7,8]
```

Listy znaków (1)

- W Haskellu znaki zapisujemy w pojedynczych apostrofach, a napisy w podwójnych. Napisy są traktowane jako listy znaków.
  - `"f"` – lista znaków,
  - `'f'` – znak,
  - `` `f` `` – funkcja traktowana jako operator.
- W Haskellu znaki specjalne zapisujemy podobnie jak w C, np. `\n`, `\t` itd.
- W Haskellu znaki są numerowane zgodnie z kodem ASCII. Dostępne są dwie funkcje ord i chr do konwersji między typami Char i Int:

```
ord :: Char -> Int
chr :: Int -> Char

:module +Data.Char  -- załadowanie modułu Data.Char
```

- Dostępny jest szereg funkcji typu `Char -> Bool`:

```
isSpace     = c == ' ' || c == '\t' || c == '\n'
isUpper     = c >= 'A' && c <= 'Z'
isLower     = c >= 'a' && c <= 'z'
isAlpha     = isUpper c || isLower c
isDigit     = c >= '0' && c <= '9'
isAlphaNum = isAlpha c || isDigit c
```

Listy znaków (2)

- Wszystkie operacje omówione do tej pory dla list są dostępne również dla list znaków.

```
-- Prelude Data.Char>
map toUpper "Hello!"
-- "HELLO!"
map toLower "Hello!"
-- "hello!"
```

- Dodatkowo dostępne są funkcje przeznaczone wyłącznie do pracy z listami znaków:
  - words – podział tekstu na listę słów,
  - lines – podział tekstu na listę linii,
  - unwords – łączenie listy słów w jeden tekst,
  - unlines – łączenie listy linii w jeden tekst.

```
words "Ala ma kota Mruczka"
-- ["Ala","ma","kota","Mruczka"]
lines "Ala ma kota Mruczka\nMruczek lubi mleko"
-- ["Ala ma kota Mruczka","Mruczek lubi mleko"]
unwords ["Ala","ma","kota","Mruczka"]
-- "Ala ma kota Mruczka"
unlines ["Ala ma kota Mruczka","Mruczek lubi mleko"]
-- "Ala ma kota Mruczka\nMruczek lubi mleko\n"
```

Listy nieskończone

- W Haskellu możliwa jest praca z listami nieskończonymi. Jeżeli wynikiem obliczeń jest lista nieskończona, to obliczenia można przerwać w dowolnym momencie za pomocą kombinacji Ctrl + C. Możliwa jest również sytuacja, gdy lista nieskończona jest argumentem obliczeń, a wynik jest listą skończoną.
- Praca z listami nieskończonymi jest możliwa dzięki temu, że Haskell jest językiem leniwym, tzn. dopóki konkretna wartość nie jest wymagana, to nie jest ona wyznaczana.
- Funkcje, które do wyznaczenia swojej wartości potrzebują wszystkich elementów listy, nie mogą być używane z listami nieskończonymi.

```
repeat :: a -> [a]                          -- prelude
repeat x =  x : repeat x

copy' :: Int -> a -> [a]
copy' n x = take n (repeat x)

-- takeWhile (< 10000) (map (2 ^) [1 .. ])
-- [2,4,8,16,32,64,128,256,512,1024,2048,4096,8192]
-- copy' 6 'A'
-- "AAAAAA"
```

Funkcja iterate

```haskell
iterate :: (a->a) -> a -> [a]          -- prelude
iterate f x = x : iterate f (f x)

take 10 (iterate (`div` 10) 5678)
-- [5678,567,56,5,0,0,0,0,0,0]

take 10 (iterate (* 2) 1)
-- [1,2,4,8,16,32,64,128,256,512]

take 5 (iterate ('x':) [])
-- ["","x","xx","xxx","xxxx"]

take 5 (iterate (2:) [])
-- [[],[2],[2,2],[2,2,2],[2,2,2,2]]


infBlowup :: a -> [[a]]
infBlowup x = iterate (x:) []

take 5 (infBlowup 2)
-- [[],[2],[2,2],[2,2,2],[2,2,2,2]]
```

Krotki

Krotki są elementami zbiorów będących iloczynami kartezjańskimi zbiorów. Krotki są zapisywane z użyciem nawiasów okrągłych. Poszczególne zbiory będące czynnikami w iloczynie kartezjańskim mogą być różnych typów.

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

snd :: (a,b) -> b      -- prelude
snd (_,y) = y

splitAt n xs = (take n xs, drop n xs)

search :: Eq a => [(a,b)] -> a -> b
search ((x,y):ts) s | x == s    = y
                    | otherwise = search ts s

splitAt 5 [2^x | x <- [1 .. 10]]
-- ([2,4,8,16,32],[64,128,256,512,1024])
splitAt 3 "Ala ma kota"
-- ("Ala"," ma kota")

search [(2^i, i) | i <- [1 ..]] 1024
-- 10
```

Funkcje zip i zipWith

```
zip :: [a] -> [b] -> [(a,b)]                          -- prelude
zip (x:xs) (y:ys) =  (x,y) : zip xs ys
zip _ _           = []

zipWith :: (a->b->c) -> [a] -> [b] -> [c]             -- prelude
zipWith f (x:xs) (y:ys) =  f x y : zipWith f xs ys
zipWith _ _ _           = []

zip [1,2,3] "kot"
-- [(1,'k'),(2,'o'),(3,'t')]

zipWith (+) [1 .. 10] [2^x | x <- [1 .. 10]]
-- [3,6,11,20,37,70,135,264,521,1034]

positions :: (Eq a) => a -> [a] -> [Int]
positions x xs = [i | (x',i) <- zip xs [1..n], x == x']
                 where n = length xs

positions 2 [1,5,2,2,7,8,2,4]                 -- [3,4,7]

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

pairs [1..9]    -- [(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(8,9)]
```

Synonimy typów

W Haskellu możliwe jest stosowanie synonimów dla typów definiowanych z użyciem słowa kluczowego type. Synonimy mogą być parametryzowane.

```
type Point3D = (Double,Double,Double)

f :: Point3D -> Double
f (x,y,z) = x + y + z

f (1,2,3)           -- 6.0
:t f                -- f :: Point3D -> Double

type List3D a = [(a,a,a)]

projection :: Int -> List3D a -> [a]
projection _ []             = []
projection 1 ((x,_,_):ts) = x : projection 1 ts
projection 2 ((_,y,_):ts) = y : projection 2 ts
projection 3 ((_,_,z):ts) = z : projection 3 ts

projection 3 [(1,2,3), (4,5,6), (7,8,9)]     -- [3,6,9]
:t projection   -- projection :: Int -> List3D a -> [a]
```

Definiowanie typów

- W Haskellu możliwe jest definiowanie nowych typów danych z użyciem słowa kluczowego data.
- Występujące po prawej stronie znaku = nazwy funkcji są nazwami konstruktorów i dla odróżnienia od zwykłych funkcji ich nazwy są pisane z wielkiej litery.
- Konstruktory mogą być funkcjami bezargumentowymi. Typy definiowane z ich użyciem odpowiadają typom wyliczeniowym znanym z innych języków programowania.
- Nazwa konstruktora może być taka sama jak nazwa typu.

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

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

move West (2,4)
-- (1,4)
```

Definiowanie typów – przykład 1

```
data Color = Red
           | Orange
           | Yellow
           | Green
           | Blue
           | Purple
           | White
           | Black
           | Custom Int Int Int

colorToRGB Red    = (255,0,0)
colorToRGB Orange = (255,128,0)
colorToRGB Yellow = (255,255,0)
colorToRGB Green  = (0,255,0)
colorToRGB Blue   = (0,0,255)
colorToRGB Purple = (255,0,255)
colorToRGB White  = (255,255,255)
colorToRGB Black  = (0,0,0)
colorToRGB (Custom r g b) = (r,g,b)

colorToRGB (Custom 23 3 4)    -- (23,3,4)
colorToRGB Blue               -- (0,0,255)
```

Definiowanie typów – przykład 2

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

treeSize :: Tree a -> Int
treeSize (Leaf _) = 1
treeSize (Node _ left right) = 1 +  treeSize left + treeSize right

bt1 = Node 7 (Node 4 (Leaf 2) (Leaf 5)) (Leaf 10)

bt2 = Node 7 (Node 4 (Leaf 2) (Leaf 5))
             (Node 10 (Leaf 9) (Node 13 (Leaf 11) (Leaf 15)))

elemTree :: Ord a => a -> Tree a -> Bool
elemTree e (Leaf x) = e == x
elemTree e (Node x left right) | e == x = True
                               | e < x  = elemTree e left
                               | e > x  = elemTree e right

treeSize bt1                   -- 5
treeSize bt2                   -- 9
elemTree 9 bt1                 -- False
elemTree 9 bt2                 -- True
```

Definiowanie klas

Haskell umożliwia definiowanie nowych klas typów i określanie, które typy do tak zdefiniowanych klas należą (są instancjami klas).

```
data Rectangle = RectPWH (Double,Double) Double Double
               | Rect2P (Double,Double) (Double,Double)

aRect :: Rectangle -> Double
aRect (RectPWH (x,y) w h) = w * h
aRect (Rect2P (x1,y1) (x2,y2)) = (x2 - x1) * (y2 - y1)

cRect :: Rectangle -> Double
cRect (RectPWH (x,y) w h) = 2*w + 2*h
cRect (Rect2P (x1,y1) (x2,y2)) = 2*(x2 - x1) + 2*(y2 - y1)

class Shape a where
  area :: a -> Double
  circumference :: a -> Double

instance Shape Rectangle where
  area = aRect
  circumference = cRect

a2c :: (Shape a) => a -> Double
a2c x = (area x) / (circumference x)
```

Instancje klas

Haskell automatycznie generuje instacje klas, jeżeli użyjemy klauzulę deriving.

```
data Rectangle = RectPWH (Double,Double) Double Double
               | Rect2P (Double,Double) (Double,Double)
               deriving Eq

(RectPWH (2,3) 1 1) == (RectPWH (2,3) 1 1)
-- True
(RectPWH (2,3) 1 1) == (RectPWH (2,4) 1 1)
-- False
(RectPWH (2,3) 1 1) == (Rect2P (2,3) (3,4))
-- False

-- ALE
instance Eq Rectangle where
  (RectPWH (x1,y1) w1 h1) == (RectPWH (x2,y2) w2 h2) =
    x1 == x2 && y1 == y2 && w1 == w2 && h1 == h2
  (Rect2P (x1,y1) (x2,y2)) == (Rect2P (x3,y3) (x4,y4)) =
    x1 == x3 && y1 == y3 && x2 == x4 && y2 == y4
  (RectPWH (x1,y1) w h) == (Rect2P (x2,y2) (x3,y3)) =
    x1 == x2 && y1 == y2 && w == x3 - x2 && h == y3 - y2

(RectPWH (2,3) 1 1) == (Rect2P (2,3) (3,4))
-- True
```

Kompilacja do pliku wykonywalnego

Moduł w Haskellu można skompilować do pliku wykonywalnego, jeżeli nazwa modułu to Main i zawiera on funkcję o nazwie main. Kompilacja takiego programu jest realizowana za pomocą polecenia:

```
ghc --make plik.hs -o plik
```

```haskell
module Main
where

silnia :: Int -> Int
silnia n | n == 0 = 1
         | n > 0  = n * silnia (n - 1)

main = print(silnia 5)
```

```
$ ghc --make silnia.hs -o silnia
[1 of 1] Compiling Main             ( silnia.hs, silnia.o )
Linking silnia ...
$ ./silnia
120
```

Klasa Monad

Monady służą do dołączania do programu kodu imperatywnego – możliwość używania zmiennych i określania kolejności wykonywania obliczeń. Użycie monad pozwala m.in. na realizowanie operacji, które mają efekty uboczne np. operacji wejścia wyjścia (IO). Jednocześnie obliczenia imperatywne są odizolowane od kodu czysto funkcyjnego.

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b   -- bind
  (>>)   :: m a -> m b -> m b          -- then
  return :: a -> m a                   -- return
  fail   :: String -> m a              -- fail, fail s = error s
```

- Typ m musi być typem z jednym parametrem – typ `a` jest „opakowany” przez `m`.
- `f >>= g` jest sekwencją akcji `f` i `g a`, gdzie `a` jest wartością akcji `f`. Operator bind umożliwia wykonywanie „czystych” funkcji na „opakowanych” typach. Biorąc pod uwagę jego typ można powiedzieć, że operator „odpakowuje” typ `a` i przekazuje go funkcji, której wynik znowu jest typem opakowanym (niekoniecznie tym samym).
- Operator then jest zazwyczaj implementowany jako `m >> n = m >>= _ -> n`
  `f >> g` jest sekwencją akcji `f` i `g`, przy czym `g` nie zależy od wyniku `f`.
- Funkcja return „**opakowuje**” typ `a` w typ `m a`.
- Funkcja fail jest konieczna ze względów technicznych (raczej nieużywana).

Implementacja monad

Dodanie typu do klasy Monad (krótko implementacja monady) wymaga zaimplementowania operatora bind i funkcji return. Muszą one spełniać wymagania:

```
m >>= return      =  m                          -- 1
return x >>= f    =  f x                        -- 2
(m >>= f) >>= g   =  m >>= (\x -> f x >>= g)    -- 3
```

Pierwsze dwa prawa definiują zachowanie funkcji return po obu stronach operatora >>=, zaś trzecie definiuje łączność operatora >>=.

Notacja do (cukier syntaktyczny):

```
do e                        -- e

do e                        -- e >> do e'
   e'

do a <- e                   -- e >>= \ a -> do e'
   e'

do let x = e                -- let x = e
   e'                       -- in  do e'
```

Implementacja monad – przykład

```
data Result a = Result a | Fault String
  deriving Show

divide :: (Eq a, Fractional a) => a -> a -> Result a
divide x y | y /= 0    = Result (x / y)
           | otherwise = Fault "Division by zero!"

doubleDivide :: (Eq a, Fractional a) => a -> a -> a -> Result a
doubleDivide a b c = case (divide a b) of   -- (a/b)/c
  Fault s  -> Fault s                       -- propagacja błędu
  Result m -> divide m c

tripleDivide :: (Eq a, Fractional a) => a -> a -> a -> a -> Result a
tripleDivide a b c d = case (divide a b) of   -- ((a/b)/c)/d
  Fault s  -> Fault s
  Result m -> case (divide m c) of
    Fault s  -> Fault s
    Result n -> divide n d

-- tripleDivide 1 1 1 1
-- Result 1.0
-- tripleDivide 1 0 1 1
-- Fault "Division by zero!"
```

Implementacja monad – przykład (cd)

```
instance Monad Result where
  a >>= f = case a of
    Fault s  -> Fault s
    Result n -> f n
  return = Result
  fail = Fault

doubleDivide' :: (Eq a, Fractional a) => a -> a -> a -> Result a
doubleDivide' a b c = (divide a b) >>= (\n -> divide n c)

tripleDivide' :: (Eq a, Fractional a) => a -> a -> a -> a -> Result a
tripleDivide' a b c d = (divide a b) >>= (\m -> divide m c)
  >>= (\n -> divide n d)

doubleDivide'' :: (Eq a, Fractional a) => a -> a -> a -> Result a
doubleDivide'' a b c = do
  n <- divide a b
  divide n c

tripleDivide'' :: (Eq a, Fractional a) => a -> a -> a -> a -> Result a
tripleDivide'' a b c d = do
  m <- divide a b
  n <- divide m c
  divide n d
```

Monada Maybe

Typ Maybe stanowi obudowę dla innego wskazanego typu, dostarczając dodatkowy konstruktor Nothing, który można wykorzystać do wskazania sytuacji wyjątkowych. Jest to najprostsza anonimowa obsługa wyjątków w Haskellu.

```
data Maybe a = Nothing | Just a          -- prelude

instance Monad Maybe where
  Just x  >>= f = f x
  Nothing >>= f = Nothing
  return        = Just
  fail s        = Nothing

findElement :: (a -> Bool) -> [a] -> Maybe a
findElement p []     = Nothing
findElement p (x:xs) = if p x then Just x
                       else findElement p xs

findElement (>9) [2,4,5,6,7,8]
-- Nothing

findElement (>6) [2,4,5,6,7,8]
-- Just 7
```

Monada IO

- Monada IO potrafi składać ze sobą operacje z efektami ubocznymi. Jest używana do realizacji operacji wejścia-wyjścia.
- W przypadku akcji IO typem wyniku akcji jest IO jakiś_typ, np:

  ```
  print :: (Show a) => a -> IO ()        -- prelude
  ```

  Występowanie IO w typie zwracanej wartości oznacza, że mogą wystąpić *efekty uboczne*, i/lub funkcja zwraca różne wartości gdy jest wywoływana z tymi samymi argumentami. `()` jest odpowiednikiem typu `void` z języka C. W tym przypadku oznacza to, że po wykonaniu akcji **`print`** nie jest zwracana żadna istotna wartość.
- Funkcja **main** jest akcją typu **`IO ()`**.
- Akcje I/O generują efekt uboczny tylko wtedy, gdy są wykonywane wewnątrz innych akcji I/O. Wykonanie akcji typu **`IO t`** oznacza, że wykonana zostanie operacja wejścia-wyjścia i ostatecznie zwrócony zostanie wynik typu `t`.

```
module Main where

main = do putStrLn "Please enter your name:"
          name <- getLine                    -- getLine :: IO String
          putStrLn ("Hello " ++ name)        -- putStrLn :: String -> IO ()
```

I/O – przykład 1

```haskell
module Main where

name2reply :: String -> String
name2reply name =
    "Pleased to meet you, " ++ name ++ ".\n" ++
    "Your name contains " ++ charcount ++ " characters."
    where charcount = show (length name)

main :: IO ()
main = do
       putStrLn "Greetings once again.  What is your name?"
       inpStr <- getLine
       let outStr = name2reply inpStr
       putStrLn outStr
```

```
$ ghc --make name.hs -o name
[1 of 1] Compiling Main             ( name.hs, name.o )
Linking name ...
$ ./name
Greetings once again.  What is your name?
Marcin
Pleased to meet you, Marcin.
Your name contains 6 characters.
```

I/O – przykład 2

W przypadku stosowania instrukcji if then else w odniesieniu do akcji wymagane jest by typ obu gałęzi był taki sam. Typ całej instrukcji jest taki sam jak typ obu gałęzi. Typem sekwencji do jest typ ostatniej operacji.

```
module Main where
import System.Random    -- apt-get install libghc-random-dev

main =                  -- main :: IO ()
  do n <- randomRIO (1::Int, 100)
     putStrLn "I'm thinking of a number berween 1 and 100:"
     doGuessing n

doGuessing n =      -- doGuessing :: (Ord a, Read a) => a -> IO ()
  do putStrLn "Enter your guess:"
     k <- getLine
     if read k < n
       then do putStrLn "Too low!"
               doGuessing n
       else if read k > n
              then do putStrLn "Too high!"
                      doGuessing n
              else do putStrLn "You win!"
```

Prelude – operacje wejścia-wyjścia

```
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()          -- dodaje nową linię
print    :: Show a => a -> IO ()
```

Funkcja print realizuje konwersję wartości dowolnego typu należącego do klasy show na tekst odpowiedni do wyświetlenia i dodaje na końcu znak przejścia do nowej linii.

```
getChar      :: IO Char
getLine      :: IO String
getContents  :: IO String
interact     :: (String -> String) -> IO ()
readIO       :: Read a => String -> IO a
readLn       :: Read a => IO a
```

Funkcja getContents zwraca całą zawartość tekstu wprowadzonego przez użytkownika, który jest przetwarzany *leniwie* w zależności od potrzeb. Funkcja interact przyjmuje jako argument funkcję typu `String -> String`, której przekazuje całą zawartość ze standardowego wejścia, a wynik tej funkcji jest wysyłany na standardowe wyjście., np.

```
main = interact (filter isAscii)
```

Funkcja read z klasy Read służy do konwersji tekstu na wartość. Funkcja readIO działa podobnie, z tym że błędy parsowania są przekazywane do monady I/O, a nie przerywają wykonywania programu. Funkcja readLn jest połączeniem getLine i readIO.

I/O – przykład 3

```
module Main
where

silnia :: Int -> Int
silnia n | n == 0 = 1
         | n > 0  = n * silnia (n - 1)

main =
  do putStrLn "Enter natural number: "
     k <- readLn                                  -- 1
     let result = silnia k                        -- 2
     print result                                 -- 3

-- lub zamiast 2 i 3
--   print (silnia k)

-- lub zamiast 1, 2 i 3
--   k <- getLine
--   let result = silnia (read k)
--   putStrLn (show result)
```

Operacje na plikach tekstowych

```
type FilePath =  String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile   :: FilePath           -> IO String
```

- Funkcje writeFile i appendFile zapisują i dopisują do pliku swój drugi argument typu `String`. Nazwa pliku jest podawana jako pierwszy argument. Zapis wartości dowolnego typu należącego do klasy Show wymaga konwersji z użyciem funkcji show.
- Funkcja readFile odczytuje całą zawartość pliku i zwraca ją jako `String`. Plik jest odczytywany *leniwie*.

```
import Data.Char(toUpper)
-- przykład 1
main = do
       inputString <- readFile "input.txt"
       writeFile "output.txt" (map toUpper inputString)

-------------------------------------------------------
-- przykład 2
main = do
       inputString <- readFile "dane.txt"
       let xs = map read (words inputString)
       print (sum xs)
```