

Wykład 3

Funkcje wyższych rzędów

Funkcje jako dane

Rozwijanie i zwijanie funkcji

Składanie funkcji

Funkcjonały dla list

Optymalizacja wyrażeń

Mechanizm konstrukcji list w Haskellu

Funkcje wyższego rzędu jako struktury sterowania

Rekonstrukcja typu

Typy kwalifikowane w Haskellu

Dodatek: *Domknięcia funkcyjne w językach Java i C++*

Funkcje wyższych rzędów a funkcje jako wartości pierwszej kategorii

Funkcja operująca na innych funkcjach jest nazywana funkcją wyższego rzędu lub funkcjonałem (ang. functional, stąd functional programming, ale po polsku programowanie funkcyjne).

Pojęcie funkcji wyższych rzędów jest ściśle związane z funkcjami jako wartościami pierwszej kategorii. W obu przypadkach funkcje mogą być argumentami i wynikami innych funkcji. „Funkcja wyższego rzędu” (funkcjonał) jest pojęciem matematycznym, natomiast „wartość pierwszej kategorii” jest pojęciem informatycznym, odnoszącym się do encji w językach programowania, na które nie są nałożone żadne ograniczenia, dotyczące ich wykorzystanie (z wyjątkiem zgodności typów). Tak więc możliwe są:

- funkcje, produkujące funkcje jako wyniki;
- funkcje jako argumenty innych funkcji;
- struktury danych (np. krotki, listy) z funkcjami jako składowymi.

Funkcje jako dane

Funkcje są często przekazywane jako dane w obliczeniach numerycznych. Poniższy funkcjonal oblicza sumę $\sum_{i=0}^m f(i)$. Dla efektywności wykorzystuje on rekursję ogonową.

```
let sigma f m =  
  let rec suma (i,s)=  
    if i=m then s else suma(i+1, s +. f(i+1))  
  in suma(0, f 0)  
;;  
val sigma : (int -> float) -> int -> float = <fun>
```

W połączeniu z funkcjonalami wygodne okazuje się wykorzystanie funkcji anonimowych. Możemy obliczyć np. $\sum_{k=0}^9 k^2$ bez konieczności oddzielnego definiowania funkcji podnoszenia do kwadratu.

```
sigma (fun k -> float(k*k)) 9;;  
- : float = 285.
```

Funkcje jako dane

Wykorzystując ten funkcjonal łatwo obliczyć np. $\sum_{i=0}^3 \sum_{j=0}^4 i + j$.

```
# sigma (fun i -> sigma (fun j -> float(i+j)) 4) 3;;  
-: float = 70.
```

W przypadku częstego sumowania elementów macierzy można łatwo uogólnić to na $\sum_{i=0}^m \sum_{j=0}^n g(i, j)$, czyli

```
# let sigma2 g m n =  
    sigma (fun i -> sigma (fun j -> g(i,j)) n) m;;  
val sigma2 : (int * int -> float) -> int -> int -> float = <fun>
```

Wartość $\sum_{i=0}^3 \sum_{j=0}^4 i + j$ można teraz wyliczyć prościej:

```
# sigma2 (fun (k,l) -> float(k+l)) 3 4;;  
- : float = 70.
```

Rozwijanie i zwijanie funkcji

Na wykładzie 2. była mowa o postaci rozwiniętej (ang. *curried*) i zwiniętej (ang. *uncurried*) funkcji (na cześć Haskella Curry'ego). Możemy napisać funkcjonal *curry* (odp. *uncurry*), który bierze funkcję w postaci zwiniętej (odp. rozwiniętej) i zwraca tę funkcję w postaci rozwiniętej (odp. zwiniętej).

```
# let curry f x y = f (x, y);;                (* Rozwijanie funkcji *)
(* let curry = function f -> function x -> function y -> f (x,y);; *)
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# let uncurry f (x, y) = f x y;;              (* Zwijanie funkcji *)
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>

# let plus (x,y) = x+y;;
val plus : int * int -> int = <fun>
# curry plus 4 5;;
- : int = 9
# let add x y = x+y;;      (* lub let add = curry plus;; *)
val add : int -> int -> int = <fun>
# uncurry add (4,5);;
-: int = 9
```

W Haskellu funkcje *curry* i *uncurry* należą do standardowego preludium.

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

Składanie funkcji

W matematyce często jest używana operacja złożenia (superpozycji) funkcji.

Niech będą dane funkcje $f:\alpha\rightarrow\beta$ oraz $g:\gamma\rightarrow\alpha$. Złożenie $(f\circ g):\gamma\rightarrow\beta$ jest definiowane następująco:

$$(f\circ g)(x) = f(g\ x).$$

```
# let ($) f g = fun x -> f ( g x);;      (* Składanie funkcji *)
val ( $ . ) : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let sqr x = x*x;;
val sqr : int -> int = <fun>
# (sqr $. sqr) 2;;
-: int = 16
# let third l3 = (List.hd $. List.tl $. List.tl) l3;;
val third : 'a list -> 'a = <fun>
# third [1;2;3;4;5];;
-: int = 3
# let next_char = Char.chr $. (+)1 $. Char.code;;
val next_char : char -> char = <fun>
# next_char 'f';;
-: char = 'g'
```

W Haskellu operator składania funkcji należy do standardowego preludium.

Prelude> :t (.)

(.) :: (b -> c) -> (a -> b) -> a -> c

Aplikacja funkcji do argumentu

Aplikacja w OCamlu (a także w innych językach funkcyjnych) jest najczęściej używaną operacją. Zwykle brak dla niej specjalnego symbolu, ma ona **najwyższy priorytet i wiąże w lewo**. Jednak w wielu językach programowania wprowadzane są symbole dla aplikacji, z nieco inną semantyką.

W bibliotece standardowej OCaml'a (moduł Pervasives) jest zdefiniowany operator aplikacji @@, który **wiąże w prawo**, czyli $f @@ g @@ x$ jest równoważne $f (g x)$.

<pre>val (@@) : ('a -> 'b) -> 'a -> 'b</pre>	Haskell	<pre>(\$) :: (a -> b) -> a -> b</pre>
<pre># abs 1 - 5;; (* wiązanie lewostronne *)</pre>		<pre>Prelude> abs 1 - 5</pre>
<pre>- : int = -4</pre>		<pre>-4</pre>
<pre># abs @@ 1 - 5;; (* wiązanie prawostronne *)</pre>		<pre>Prelude> abs \$ 1 - 5</pre>
<pre>- : int = 4</pre>		<pre>4</pre>

Zdefiniowany jest też operator odwróconej aplikacji (ang. reverse-application operator), który ma odwróconą kolejność argumentów i **wiąże w lewo**:

```
val (|>) : 'a -> ('a -> 'b) -> 'b
```

Tutaj $x |> g |> f$ jest równoważne $f (g x)$, czyli definicja jest taka: $\text{let } (|>) \ x \ f = f \ x$.

W języku F# jest on zwykle nazywany operatorem potokowym (ang. forward pipe operator)

```
# 5-12 |> abs |> succ;;  
- : int = 8
```

Obciążenia (zawężenia) funkcji w Haskellu

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

$$(\odot) \equiv \lambda x \rightarrow \lambda y \rightarrow x \odot y$$

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

$$(x \odot) \equiv \lambda y \rightarrow x \odot y$$

$$(\odot y) \equiv \lambda x \rightarrow x \odot y$$

Dotyczy to również infiksowych i miksfixsowych wbudowanych konstruktorów wartości.

W języku OCaml łatwo można ustalić lewy argument:

$$(\odot) x \equiv \text{fun } y \rightarrow x \odot y$$

Ustalenie prawego argumentu wymaga napisania wyrażenia funkcyjnego:

$$\text{fun } x \rightarrow x \odot y$$

Funkcjonały dla list - map

Łącząc funkcje wyższych rzędów i parametryczny polimorfizm można pisać bardzo ogólne funkcjonały.

Funkcjonał *map* aplikuje funkcję do każdego elementu listy:

$$\text{map } f \ [x_1; \dots ; x_n] \rightarrow [f \ x_1; \dots ; f \ x_n]$$

```
# let rec map f xs =  
  match xs with  
  [] -> []  
  | x::xs -> (f x) :: map f xs ;;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>  
# map (fun x -> x*x) [1;2;3;4];;  
-: int list = [1; 4; 9; 16]
```

Ten funkcjonał jest zdefiniowany w module List:

```
# List.map;;  
-: f:('a -> 'b) -> 'a list -> 'b list = <fun>  
# List.map String.length ["Litwo";"ojczyzna";"moja"];;  
-: int list = [5; 8; 4]
```

W Haskellu funkcja *map* należy do standardowego preludium.

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

Optymalizacja wyrażeń

Przez indukcję można udowodnić wiele użytecznych własności równościowych dla list (i innych struktur danych), które można wykorzystać do optymalizacji programów, np.

$$\text{map } f (\text{map } g \text{ xs}) = \text{map } (f \circ g) \text{ xs}$$

lub krócej

$$(\text{map } f) \circ (\text{map } g) = \text{map } (f \circ g)$$

Funkcja z lewej strony powyższej równości wymaga dwukrotnego przejścia listy (po drodze powstaje lista tymczasowa); natomiast funkcja z prawej strony przechodzi po liście tylko raz i tworzy od razu listę wynikową.

Obie funkcje są poprawne z punktu widzenia matematyka, ale informatyk powinien mieć zawsze na uwadze efektywność programu i oczywiście wybierze funkcję z prawej strony równości.

Funkcjonały dla list - filter

Funkcjonał *filter* aplikuje predykat do każdego elementu listy; jako wynik zwraca listę elementów spełniających ten predykat w oryginalnym porządku.

```
# let rec filter pred = function
  [] -> []
  | x::xs -> if pred x then x::filter pred xs
              else filter pred xs
;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Efektywniejszą implementację (z wykorzystaniem rekursji ogonowej) można znaleźć w module List.

```
# List.filter (fun s -> String.length s <= 6)
  ["Litwo"; "ojczyzna"; "moja"];;
-: string list = ["Litwo"; "moja"]
```

W Haskellu funkcja *filter* należy do standardowego preludium.

```
filter :: (a -> Bool) -> [a] -> [a]
```

Filter z rekursją ogonową

```
# let filter_bad p =  
  let rec find acc = function      (* zamiast match *)  
    [] -> acc  
  | x :: xs -> if p x then find (acc@[x]) xs else find acc xs  
  in  
    find [];;  
val filter_bad : ('a -> bool) -> 'a list -> 'a list = <fun>
```

(* Złożoność $O(n^2)$. Nigdy w ten sposób! *)

```
let filter p =  
  let rec find acc = function  
    [] -> List.rev acc  
  | x :: xs -> if p x then find (x :: acc) xs else find acc xs  
  in  
    find [];;
```

(* Złożoność $O(2n)$. Tylko tak! *)

Generowanie liczb pierwszych metodą sita Eratostenesa

Utwórz ciąg skończony $[2, 3, 4, 5, 6, \dots, n]$. Dwa jest liczbą pierwszą. Usuń z ciągu wszystkie wielokrotności dwójki, ponieważ nie mogą być liczbami pierwszymi. Pozostaje ciąg $[3, 5, 7, 9, 11, \dots]$. Trzy jest liczbą pierwszą. Usuń z ciągu wszystkie wielokrotności trójki, ponieważ nie mogą być liczbami pierwszymi. Pozostaje ciąg $[5, 7, 11, 13, 17, \dots]$. Pięć jest liczbą pierwszą itd.

Na każdym etapie ciąg zawiera liczby mniejsze lub równe n , które nie są podzielne przez żadną z wygenerowanych do tej pory liczb pierwszych, więc pierwsza liczba tego ciągu jest liczbą pierwszą. Powtarzając opisane kroki otrzymamy wszystkie liczby pierwsze z zakresu $[2..n]$.

```
# let primes to_n =  
  let rec sieve n =  
    if n <= to_n then n::sieve(n+1) else []  
  and find_primes = function  
    h::t -> h:: find_primes (List.filter (fun x -> x mod h <> 0) t)  
    | [] -> []  
  in find_primes(sieve 2);;  
val primes : int -> int list = <fun>  
  
# primes 30;;  
- : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29]
```

Funkcjonały dla list - insert

Funkcjonał *insert* bierze funkcję *poprzedza* (w postaci rozwiniętej) zadającą porządek elementów w liście, wstawiany element *elem* oraz listę uporządkowaną i wstawia *elem* do listy zgodnie z zadany porządkiem.

```
# let rec insert poprzedza elem xs =  
  match xs with  
    [] -> [elem]  
  | h::t as l -> if poprzedza elem h then elem::l  
                  else h::(insert poprzedza elem t);;  
val insert : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list = <fun>
```

Funkcję *insert* łatwo wyspecjalizować dla zadanego porządku, np.

```
# let insert_le elem = insert (<=) elem;;  
val insert_le : 'a -> 'a list -> 'a list = <fun>  
# insert_le 4 [1;2;3;4;5;6;7;8];;  
- : int list = [1; 2; 3; 4; 4; 5; 6; 7; 8]
```

Funkcjonały dla list – uogólnienie typowych funkcji

```
# let rec sumlist xs =  
  match xs with  
    h::t -> h + sumlist t  
        (* funkcja f, której argumentami są głowa listy h i wynik wywołania  
          rekurencyjnego definiowanej funkcji na ogonie t *)  
    | [] -> 0;; (* wynik acc definiowanej funkcji dla listy pustej [] *)  
val sumlist : int list -> int = <fun>
```

Analogiczną strukturę miało wiele rozważanych przez nas funkcji na listach. Uogólniając tę funkcję i umieszczając `f` i `acc` jako dodatkowe argumenty otrzymujemy poniższy funkcjonal:

```
# let rec fold_right f xs acc =  
  match xs with  
    h::t -> f h (fold_right f t acc)  
    | [] -> acc;;  
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Funkcję `sumlist` możemy teraz zdefiniować wykorzystując funkcjonal `fold_right`:

```
# let sumlist xs = fold_right (+) xs 0;;  
val sumlist : int list -> int = <fun>
```

Funkcjonały dla list – fold_left – fold_right

Funkcjonał *fold_left* aplikuje dwuargumentową funkcję *f* do każdego elementu listy (z lewa na prawo) akumulując wynik w *acc* (efektywnie – rekursja ogonowa):

$$\text{fold_left } f \text{ acc } [x_1; \dots ; x_n] \rightarrow f(\dots (f(f \text{ acc } x_1) x_2) \dots) x_n$$

```
# let rec fold_left f acc = function
  h::t -> fold_left f (f acc h) t
  | [] -> acc;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Dualny funkcyjonał *fold_right* aplikuje dwuargumentową funkcję *f* do każdego elementu listy (z prawa na lewo) akumulując wynik w *acc* (zwykła rekursja):

$$\text{fold_right } f [x_1; \dots ; x_n] \text{ acc} \rightarrow f x_1 (f x_2 (\dots (f x_n \text{ acc}) \dots))$$

```
# let rec fold_right f xs acc =
  match xs with
  h::t -> f h (fold_right f t acc)
  | [] -> acc;;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Moduł `List` w OCamlu zawiera wiele użytecznych funkcyjonałów dla list.

W Haskellu funkcje *foldl* i *foldr* należą do standardowego preludium.

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

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

fold_left - wykorzystanie

```
# let sumlist = List.fold_left (+) 0;;          (* sumowanie elementów listy *)
val sumlist : int list -> int = <fun>
# sumlist [4;3;2;1];;
- : int = 10

# let prodlist = List.fold_left ( * ) 1;;       (* mnożenie elementów listy *)
val prodlist : int list -> int = <fun>
# prodlist [4;3;2;1];;
-: int = 24

# let flatten xs = List.fold_left (@) [] xs;;  (* "spłaszczanie" list *)
val flatten : 'a list list -> 'a list = <fun>
# flatten [[5;6];[1;2;3]];
-: int list = [5; 6; 1; 2; 3]
(* W Haskellu do tego celu służy funkcja concat ze standardowego preludium.
   concat :: [[a]] -> [a]  *)

(* konkatenowanie elementów listy napisów *)
# let implode = List.fold_left (^) "";;
val implode : string list -> string = <fun>
# implode ["Ala "; "ma "; "kota"];;
- : string = "Ala ma kota"
```

Abstrakcja pomaga zauważyć i wyrazić analogie

Bez wykorzystania odpowiednio wysokiego poziomu abstrakcji funkcyjnej trudno byłoby zauważyć analogie między funkcjami z poprzedniego slajdu. Struktura funkcji jest identyczna, ponieważ w każdym wypadku mamy do czynienia z monoidem:

$\langle \text{liczby całkowite}, +, 0 \rangle$

$\langle \text{liczby całkowite}, *, 1 \rangle$

$\langle \text{listy}, @, [] \rangle$

$\langle \text{napisy}, ^, "" \rangle$

Operacja monoidu jest łączna, więc w omawianych wyżej funkcjach moglibyśmy użyć funkcjonału `fold_right`, np.

```
# let implodeR xs = List.fold_right (^) xs "";;  
val implodeR : string list -> string = <fun>  
# implodeR ["Ala "; "ma "; "kota"];;  
- : string = "Ala ma kota"
```

Przykłady algebr abstrakcyjnych (homogenicznych)

- *Półgrupa* $\langle S, \bullet \rangle$ $\bullet : S \times S \rightarrow S$

$$a \bullet (b \bullet c) = (a \bullet b) \bullet c \quad (\text{łączność})$$

- *Monoid* $\langle S, \bullet, 1 \rangle$ jest półgrupą z obustronną jednością (elementem neutralnym) $1 : S$

$$a \bullet 1 = a \quad 1 \bullet a = a$$

- *Grupa* $\langle S, \bullet, \bar{}, 1 \rangle$ jest monoidem, w którym każdy element posiada element odwrotny względem

binarnej operacji monoidu $\bar{} : S \rightarrow S$

$$a \bullet a = 1 \quad a \bullet \bar{a} = 1$$

Motto

Matematykiem jest, kto umie znajdować analogie między twierdzeniami; lepszym, kto widzi analogie dowodów, i jeszcze lepszym, kto dostrzega analogie teorii, a można wyobrazić sobie i takiego, co między analogiami widzi analogie.

Stefan Banach

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

$[a, b \dots] \equiv a : a+d : a+2d : a+3d : \dots$ gdzie $d = b-a$

$[a, b \dots c] \equiv a : a+d : a+2d : a+3d : \dots : c : []$ gdzie $d = b-a$

$[a \dots] \equiv a : a+1 : a+2 : a+3 : \dots$

$[a \dots c] \equiv a : a+1 : a+2 : a+3 : \dots : c : []$

Przykład.

$[-1.5, 0 \dots 7] \equiv [-1.5, 0.0, 1.5, 3.0, 4.5, 6.0, 7.5]$

`factorial :: Int -> Int`

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

`factorial 5 => 120`

Funkcje wyższego rzędu jako struktury sterowania

```
int silnia (int n)
{ int i,s;
  i=0; s=1;           // utworzenie pary <0,1>
  while (i < n)       // n-krotna iteracja
  { i = i+1;          // operacji f takiej,
    s = i*s;           // że f<i,s> = <i+1,(i+1)*s>
  }                   // niezmiennik pętli i! = s
  return s;           // wydzielenie drugiego elementu pary
}
```

Wykorzystując nieformalne komentarze, możemy formalnie zapisać tę funkcję w języku OCaml (z rekursją ogonową, co jest naturalne zważywszy, że komentarze dotyczą wersji iteracyjnej). Ten przykład jest ilustracją faktu, że siła wyrazu języków funkcyjnych jest równa sile wyrazu języków imperatywnych.

```
let silnia' n =
  let rec f(i,s) = if i<n then f(i+1,(i+1)*s) else (i,s)
  in snd (f(0,1));;
# silnia' 5;;
- : int = 120
```

Rekonstrukcja typu

```
# let f z y x = y (y z x) ;;  
val f : 'a -> ('a -> 'b -> 'a) -> 'b -> 'b -> 'a = <fun>
```

Skąd kompilator wziął ten typ? Formalnie typ powstał w wyniku rozwiązania układu równań w pewnej algebrze typów. Na podstawie liczby argumentów widzimy, że musi być

$f : \alpha \rightarrow \beta \rightarrow \gamma \rightarrow \delta$.

Z kontekstu użycia argumentu $y : \beta$ widzimy, że y jest funkcją i otrzymujemy dwa równania:

$\beta = \alpha \rightarrow \gamma \rightarrow \varphi$ oraz $\beta = \varphi \rightarrow \delta$

w których przez φ oznaczyliśmy typ wyniku funkcji y .

Oczywiście $\beta = \alpha \rightarrow (\gamma \rightarrow \varphi) = \varphi \rightarrow \delta$, skąd otrzymujemy:

$\alpha = \varphi$ oraz $\delta = \gamma \rightarrow \varphi = \gamma \rightarrow \alpha$

czyli $\beta = \alpha \rightarrow \gamma \rightarrow \alpha$.

Na argumenty x i z kontekst nie nakłada żadnych ograniczeń, więc ostatecznie

$f : \alpha \rightarrow (\alpha \rightarrow \gamma \rightarrow \alpha) \rightarrow \gamma \rightarrow \gamma \rightarrow \alpha$.

W praktyce do rekonstrukcji typu kompilator wykorzystuje *algorytm unifikacji*.

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.

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

Typy kwalifikowane w Haskellu będą przedstawione później.

Domknięcia funkcyjne w języku Java

Najważniejsze nowości w wersji Java 8.

- Wyrażenia funkcyjne, lambda wyrażenia (ang. lambda expressions)
- Referencje do metod (ang. method references)
- Metody domyślne (ang. default methods)
- Interfejsy funkcyjne (ang. functional interfaces)
- Strumienie (ang. streams)

Strumienie pozwalają przetwarzać kolekcje danych w stylu funkcyjnym. Rekursywne definicje strumieni nie są dopuszczalne. Można jednak zdefiniować własne listy leniwe, o zachowaniu analogicznym do list leniwych w OCamlu (wykład 5).

Przetwarzanie strumieni zwykle wymaga podania:

- źródła danych (np. kolekcji)
- ciągu operacji pośrednich, tworzących potok (ang. pipeline)
- operacji końcowej, wymuszającej potokowe wykonanie operacji pośrednich i produkującej wynik.

Następujący program ilustruje wykorzystanie strumieni (porównaj go z przykładami z tego wykładu).

Wykorzystanie strumieni w wersji Java 8 - przykłady

```
import java.util.*;           // Arrays, List<E>
import static java.util.stream.Collectors.toList;

public class StreamExamples{
    public static void main(String...args){
        List<String> words = Arrays.asList("Litwo", "ojczyzna", "moja");
        List<Integer> wordLengths = words.stream().map(String::length).collect(toList());
        System.out.println(wordLengths);    // [5, 8, 4]
        List<String> wordsFiltered = words.stream().filter(s -> s.length() <= 6).collect(toList());
        System.out.println(wordsFiltered);  // [Litwo, moja]
        List<Integer> numbers = Arrays.asList(1,-2,3,4);
        System.out.println(numbers.stream().map(n -> n*n).collect(toList())); // [1, 4, 9, 16]
        int sum1 = numbers.stream().reduce(0, (a, b) -> a + b);
        System.out.println(sum1);           // 6
        int sum2 = numbers.stream().reduce(0, Integer::sum);
        System.out.println(sum2);           // 6
        int max = numbers.stream().reduce(0, (a, b) -> Integer.max(a, b));
        System.out.println(max);            // 4
        Optional<Integer> min = numbers.stream().reduce(Integer::min);
        System.out.println(min);            // Optional[-2]
    }
}
```

Domknięcia funkcyjne w języku C++

W standardowej bibliotece C++ istnieją odpowiedniki funkcjonalów z języków funkcyjnych:

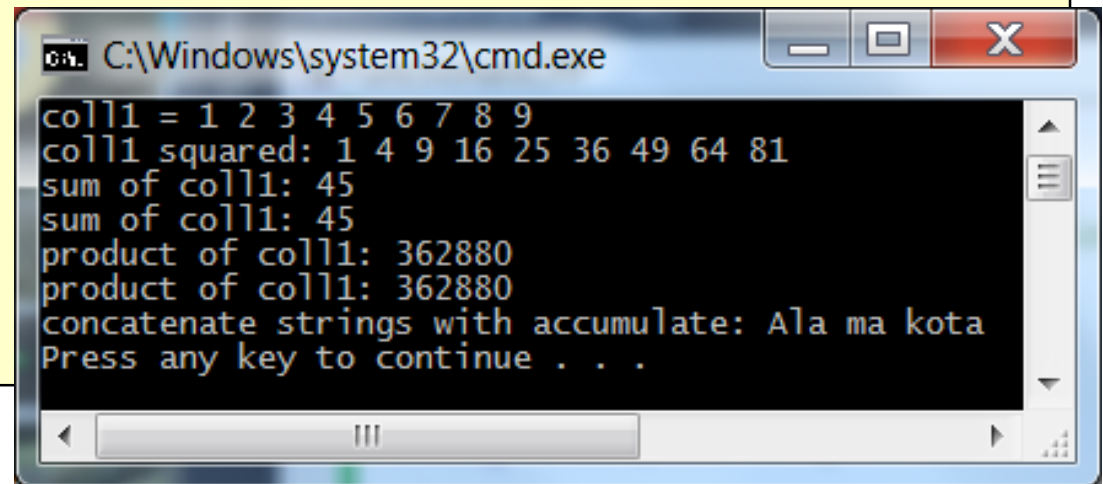
find	std::find_if
map	std::transform
filter	std::copy_if (std::remove_if razem z std::erase)
fold_left	std::accumulate
fold_right	W bibliotece C++ nie ma odpowiednika funkcjonału fold_right, ale można zasymulować jego działanie, przekazując do std::accumulate iteratory odwrotne rbegin, rend (lub crbegin, crend).

Funkcje std::find_if , std::transform oraz std::copy_if wymagają dołączenia pliku nagłówkowego <algorithm>, a funkcja std::accumulate - pliku nagłówkowego <numeric>.

Funkcjonały w C++

```
#include <vector>
#include <algorithm>
#include <functional> // multiplies
#include <numeric>
#include <iostream>
#include <string>
using namespace std;

template <typename T>
void printElements(const T& coll) {
    for (const auto& elem : coll)
        std::cout << elem << ' ';
    std::cout << std::endl;
}
```



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays the output of a C++ program. The output is as follows:

```
coll1 = 1 2 3 4 5 6 7 8 9
coll1 squared: 1 4 9 16 25 36 49 64 81
sum of coll1: 45
sum of coll1: 45
product of coll1: 362880
product of coll1: 362880
concatenate strings with accumulate: Ala ma kota
Press any key to continue . . .
```

Funkcjonały w C++

```
int main() {  
    vector<int> coll1{ 1, 2, 3, 4, 5, 6, 7, 8, 9 }, coll2;  
    cout << "coll1 = "; printElements(coll1);  
  
    transform(coll1.cbegin(), coll1.cend(), std::back_inserter(coll2), [](int i){return i*i;});  
    cout << "coll1 squared: "; printElements(coll2);  
  
    cout << "sum of coll1: " << accumulate(coll1.cbegin(), coll1.cend(), 0, [](int acc, int b){return acc + b; })  
        << endl;          // lub  
    cout << "sum of coll1: " << accumulate(coll1.cbegin(), coll1.cend(), 0) << endl;  
  
    cout << "product of coll1: " << accumulate(coll1.cbegin(), coll1.cend(), 1, [](int acc, int b){return acc*b; })  
        << endl;          // lub  
    cout << "product of coll1: " << accumulate(coll1.cbegin(), coll1.cend(), 1, multiplies<int>()) << endl;  
  
    vector<string> strings{ "Ala ", "ma ", "kota" };  
    cout << "concatenate strings with accumulate: "  
        << accumulate(strings.cbegin(), strings.cend(), string(), [](string acc, string el) {return acc + el; }) << endl;  
}
```

Zadania kontrolne (1)

1. Podaj typy poniższych funkcji:

a) **let** f1 x = x 2 2;;

b) **let** f2 x y z = x (y ^ z);;

c) **let** f3 x y z = x y z;;

d) **let** f4 x y = **function** z -> x::y;;

2. Zdefiniuj funkcje *curry3* i *uncurry3*, przeprowadzające konwersję między zwiniętymi i rozwiniętymi postaciami funkcji od trzech argumentów. Podaj ich typy.

4. Przekształć poniższą rekurencyjną definicję funkcji *sumprod*, która oblicza jednocześnie sumę i iloczyn listy liczb całkowitych na równoważną definicję nierekurencyjną z jednokrotnym użyciem funkcji *fold_left*.

```
let rec sumprod = function
  h::t -> let (s,p)= sumprod t
          in (h+s,h*p)
| []    -> (0,1) ;;
```

Zadania kontrolne (2)

5. Poniższe dwie wersje funkcji *quicksort* działają niepoprawnie. Dlaczego?

```
let rec quicksort = function
  [] -> []
| [x] -> [x]
| xs -> let small = List.filter (fun y -> y < List.hd xs ) xs
        and large = List.filter (fun y -> y >= List.hd xs ) xs
        in quicksort small @ quicksort large;;
```

```
let rec quicksort' = function
  [] -> []
| x::xs -> let small = List.filter (fun y -> y < x ) xs
          and large = List.filter (fun y -> y > x ) xs
          in quicksort' small @ (x :: quicksort' large);;
```

Zdefiniuj poprawną i efektywniejszą wersję (w której podział listy jest wykonywany w czasie liniowym w jednym przejściu) funkcji *quicksort*.

6. Zdefiniuj funkcje sortowania

a) przez wstawianie z zachowaniem stabilności i złożoności $O(n^2)$

`insort : ('a->'a->bool) -> 'a list -> 'a list.`

b) przez łączenie (scalanie) z zachowaniem stabilności i złożoności $O(n \lg n)$

`mergesort : ('a->'a->bool) -> 'a list -> 'a list.`

Pierwszy parametr jest funkcją, sprawdzającą porządek. Zamieść testy sprawdzające stabilność.