

***„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”***

# ***Wykład 3***

## ***Funkcje wyższych rzędów***

# *Funkcje wyższych rzędów*

Funkcje jako dane

Rozwijanie i zwijanie funkcji

Metody jako operatory w języku Scala

Operatory infiksowe w języku OCaml

Składanie funkcji

Funkcjonały dla list

Wyrażenie for w języku Scala

Funkcje wyższego rzędu jako struktury sterowania

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

Dodatek: *Języki formalne – przypomnienie*

Gramatyki formalne

Kompilator jednoprzebiegowy

# *Funkcje wyższych rzędów a funkcje jako wartości pierwszej kategorii*

Funkcja operująca na innych funkcjach jest nazywana funkcją wyższego rzędu lub funkcjonalem (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 (i,j) -> float(i+j)) 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 funkcjonał *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 *)
(* let uncurry = function f -> function (x,y) -> f x y *)
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
```

# *Rozwijanie i zwijanie funkcji - Scala*

```
val plus = (x:Int,y:Int) => x+y
```

```
plus: (Int, Int) => Int = <function2>
```

```
// curried jest metodą, zdefiniowaną w cesze Function2
```

```
(plus curried 4) (5)
```

```
res0: Int = 9
```

```
val add = (x:Int) => (y:Int) => x+y
```

```
add: Int => (Int => Int) = <function1>
```

```
// uncurried jest przeciążoną metodą, zdefiniowaną w obiekcie Function
```

```
Function.uncurried (add)
```

```
res1: (Int, Int) => Int = <function2>
```

```
Function.uncurried (add) (4,5)
```

```
res2: Int = 9
```

# *Operatory postfiksowe w języku Scala*

W OCamlu można zdefiniować:

```
# let add = curry plus;;  
val add : int -> int -> int = <fun>
```

Analogiczna definicja w Scali powoduje błąd (od wersji 2.13):

```
scala> plus curried
```

```
^
```

error: postfix operator curried needs to be enabled

by making the implicit value scala.language.postfixOps visible.

This can be achieved by adding the import clause 'import scala.language.postfixOps'

or by setting the compiler option -language:postfixOps.

See the Scaladoc for value scala.language.postfixOps for a discussion why the feature needs to be explicitly enabled.



# *Operatory postfiksowe w języku Scala*

Należy stosować zalecaną klauzulę import (przynajmniej w trakcie używania Scali jako języka funkcyjnego).

```
scala> import scala.language.postfixOps
import scala.language.postfixOps
scala> plus curried
res3: Int => (Int => Int) = <function1>
```

Innym sposobem uniknięcia powyższego błędu kompilacji jest stosowanie notacji kropkowej, co jawnie pokazuje, że `curried` jest metodą, wywołaną na rzecz obiektu `plus` (w Scali funkcje są obiektami).

```
scala> plus.curried
res4: Int => (Int => Int) = <function1>
```

# Metody jako operatory w języku Scala

W języku Scala nie ma operatorów, ale metody mogą być wykorzystywane w notacji operatorowej. Metoda wykorzystywana w notacji operatorowej, np. `a*b`, jest wywoływana dla *lewego* argumentu, czyli tak `a.*(b)` – chyba że nazwa metody kończy się dwukropkiem. W takim wypadku metoda jest wywoływana dla *prawego* argumentu. Np. w `1::Nil` metoda jest wywoływana dla prawego argumentu, czyli tak `Nil.:(1)`.

## Pierwszeństwo operatorów (w porządku malejącym)

Pierwszeństwo operatorów w Scali jest oparte na *pierwszym* znaku nazwy metody wykorzystywanej w notacji operatorowej (z wyjątkiem operatorów przypisania).

(wszystkie inne znaki specjalne)

\* / %

+ -

:

= !

< >

&

^

(wszystkie litery)

(wszystkie operatory przypisania)

## Łączność operatorów

Łączność operatorów w Scali jest oparta na *ostatnim* znaku nazwy metody. Jeśli nazwa metody kończy się dwukropkiem `'.'`, to mamy łączność prawostronną; w przeciwnym razie - lewostronną.

# Operatory infiksowe w języku OCaml

Każdy operator infiksowy może być zamieniony na funkcję dwuargumentową w postaci rozwiniętej przez umieszczenie go w nawiasach: *( op )*

```
# ( + );;  
- : int -> int -> int = <fun>  
# let succ = ( + ) 1;;  
val succ : int -> int = <fun>  
# succ 3;;  
- : int = 4
```

## *Można definiować swoje operatory infiksowe i prefiksowe*

```
# let (++) c1 c2 = ((fst c1) + (fst c2), (snd c1) + (snd c2));;  
val ( ++ ) : int * int -> int * int -> int * int = <fun>  
# let c = (2,3) in c ++ c;;  
- : int * int = (4, 6)
```

*ident ::= ( litera | \_ ) { litera | 0 ... 9 | \_ | ' }*

*operator ::= ! | \$ | % | & | \* | + | - | . | / | : | < | = | > | ? | @ | ^ | | | ~*

*op-infiksowy ::= ( = | < | > | @ | ^ | | | & | + | - | \* | / | \$ | % ) { operator }*

*op-prefiksowy ::= ( ! | ? | ~ ) { operator }*

Połączenie zbiorów identyfikatorów alfanumerycznych i symbolicznych spowodowałoby konieczność oddzielania identyfikatorów spacjami, np. wyrażenie `x+1` byłoby traktowane jako identyfikator, a nie jako `x + 1`.

# Operatory infiksowe w języku OCaml

(\* Przykład: implikacja \*)

let (=>) b1 b2 =

(\* postać rozwinięta, operator infiksowy \*)

match (b1, b2) with

(true, false) -> false

| \_ -> true;;

(\* użycie

# false => true;;

- : bool = true

\*)

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

# (+);;

- : int -> int -> int = <fun>

# (+.);;

- : float -> float -> float = <fun>

# (=);;

- : 'a -> 'a -> bool = <fun>

Informacje o łączności i względnych priorytetach operatorów OCaml można znaleźć [w dokumentacji OCaml](#).

# *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 n = n*n;;
val sqr : int -> int = <fun>
# ((fun n -> 2*n) $. sqr) 3;;
-: int = 18

# 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'
```

# *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** (wśród operatorów infiksowych) 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  $f (g x)$ .

```
val (@@) : ('a -> 'b) -> 'a -> 'b
# abs 1 - 5;;          (* wiązanie lewostronne *)
- : int = -4
# abs @@ 1 - 5;;      (* wiązanie prawostronne *)
- : int = 4
```

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
```

# *Składanie funkcji - Scala*

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

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

```
val sqr = (n:Int) => n*n
sqr: Int => Int = <function1>
// compose jest metodą, zdefiniowaną w cesze Function1
((n:Int) => 2*n) compose sqr
res7: Int => Int = <function1>
(((n:Int) => 2*n) compose sqr) (3)
res8: Int = 18
```

Stosowana jest też notacja, uwzględniająca kolejność obliczania składanych funkcji.

$$f ; g = g \circ f$$

Cecha Function1 definiuje odpowiednią metodę andThen.

```
(((n:Int) => 2*n) andThen sqr) (3)
res9: Int = 36
```

## *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 funkcjonal 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]
```

**Scala: `xs map f` // map jest metodą w klasie List**



# 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 xs =  
  match xs with  
    [] -> []  
  | 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"; "ojczyzno"; "moja"];;  
-: string list = ["Litwo"; "moja"]
```

**Scala:** `xs filter pred`    `// filter jest metodą w klasie List`

## *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 - OCaml*

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 xs =
    match xs with
      | 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]
```

# *Generowanie liczb pierwszych metodą sita Eratostenesa - Scala*

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

```
val primes = (toN: Int) => {  
  def findPrimes(sieve:List[Int]):List[Int] =  
    sieve match {  
      case h::t => h::findPrimes (t filter (x=>x%h != 0))  
      case Nil  => Nil  
    }  
  findPrimes(List.range(2, toN + 1))  
}  
primes: Int => List[Int] = <function1>  
  
primes(30)  
res7: List[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 zadaniem 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 – insert (Scala)*

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.

```
def insert[A](poprzedza: A=>A=>Boolean)(elem: A)(xs: List[A]): List[A] = xs match {  
  case Nil => elem::Nil  
  case y::ys => if (poprzedza (elem) (y)) elem::xs else y::insert(poprzedza)(elem)(ys)  
}  
insert: [A](poprzedza: A => (A => Boolean))(elem: A)(xs: List[A])List[A]
```

Funkcję *insert* łatwo wyspecjalizować dla zadanego porządku (wyjaśnienie będzie później), np.

```
def insertLe[A] (elem: A) (xs: List[A]) (implicit comparator: A => Ordered[A]) =  
  insert ((x: A) => (y: A) => x<= y) (elem) (xs)  
insertLe: [A](elem: A)(xs: List[A])(implicit comparator: A => Ordered[A])List[A]  
  
insertLe (4) (List(1,2,3,4,5,6,7,8))  
res8: List[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* w postaci rozwiniętej 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 xs =  
  match xs with  
  | 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.

# Przykłady w języku Scala

**List(1,2,3,4) map (x=>x\*x)**

res9: List[Int] = List(1, 4, 9, 16)

**List(1,2,3,4) filter (x=>x%2 == 0)**

res10: List[Int] = List(2, 4)

**(List(1,2,3,4) foldLeft 0) ((sum,x)=>sum+x)**

res11: Int = 10

**(0 /: List(1,2,3,4)) ((sum,x)=>sum+x)**      // /: to jest alias dla foldLeft

res12: Int = 10

**(List(1,2,3,4) foldRight 0) ((x,sum)=>sum+x)**

res13: Int = 10

**(List(1,2,3,4) :\ 0) ((x,sum)=>sum+x)**      // :\ to jest alias dla foldRight

res14: Int = 10

## *Przykłady w języku Scala*

**List(List(5,6), List(1,2,3)) flatten**

res15: List[Int] = List(5, 6, 1, 2, 3)

Złożenie funkcji map z konkatenacją podlist, otrzymanych jako wynik z funkcji map jest tak częste, że istnieje specjalna metoda w klasie List o nazwie flatMap. Zamiast pisać np.

**List(List(5,6),List(1,2,3)) map (x=> x tail) flatten**

można użyć flatMap:

**List(List(5,6),List(1,2,3)) flatMap (x=> x tail)**

res16: List[Int] = List(6, 2, 3)

## *fold\_left* - wykorzystanie

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

(\* sumowanie elementów listy \*)

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

(\* mnożenie elementów listy \*)

```
# let flatten xss = List.fold_left (@) [] xss;;  
val flatten : 'a list list -> 'a list = <fun>  
# flatten [[5;6];[1;2;3]];;  
-: int list = [5; 6; 1; 2; 3]
```

(\* "spłaszczanie" list \*)

```
(* konkatowanie 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*  $\langle S, \bullet, \bar{\phantom{x}}, 1 \rangle$  jest monoidem, w którym każdy element posiada element odwrotny względem

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

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

## *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

## Wyrażenie *for* w języku Scala

Wyrażenie *for* (ang. for comprehension, for expression) w języku Scala jest użyteczną abstrakcją lingwistyczną dla sekwencji wywołań funkcji `map`, `flatMap` i `filter`. Idea i nazwa pochodzą z matematyki (ang. set comprehension), gdzie zbiór można zdefiniować opisując własności elementów zbioru, np.  $\{x^2 \mid x \in \{1..5\}\} = \{1, 4, 9, 16, 25\}$ .

```
for (x <- List.range(1, 6)) yield x*x  
res0: List[Int] = List(1, 4, 9, 16, 25)
```

Haskell udostępnia jeszcze zwięźlejszą notację dla list (ang. list comprehension): `[x^2 | x <- [1..5]]`

Składnia wyrażenia *for* w Scali jest następująca:

```
for (seq) yield expr
```

*seq* jest tutaj sekwencją *generatorów*, *definicji* i *filtrów*, oddzielonych średnikami.

```
def factors (n:Int) = for (k <- List.range(1, n+1); if n % k == 0) yield k    // lista dzielników n  
factors: (n: Int)List[Int]
```

```
factors(15)  
res1: List[Int] = List(1, 3, 5, 15)
```



# *Wyrażenie for w języku Scala*

*Generator* jest postaci:

*pattern* <- *expr*

*Definicja* jest postaci:

*pattern* = *expr*

*Filtr* jest postaci:

if *expr*

*expr* jest tutaj wyrażeniem typu Boolean. Filtr usuwa z iteracji wszystkie elementy, dla których wartość *expr* jest równa false.

Każde wyrażenie for zaczyna się od generatora. Jeśli w wyrażeniu for występuje kilka generatorów, to późniejsze generatory zmieniają się szybciej niż wcześniejsze, np.

```
for (x <- List("one", "two"); y<-List(1,2)) yield (x,y)
```

```
res2: List[(String, Int)] = List((one,1), (one,2), (two,1), (two,2))
```

## *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 języka OCaml 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:

$$\beta = \alpha \rightarrow \gamma \rightarrow \varphi \quad \text{oraz} \quad \beta = \varphi \rightarrow \delta$$

w których przez  $\varphi$  oznaczyliśmy typ wyniku funkcji  $y$ .

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

$$\alpha = \varphi \quad \text{oraz} \quad \delta = \gamma \rightarrow \varphi = \gamma \rightarrow \alpha$$

$$\text{czyli } \beta = \alpha \rightarrow \gamma \rightarrow \alpha.$$

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*.

W językach z typizacją statyczną nie są potrzebne predykaty, sprawdzające typy.

# *Definicja języka programowania (wymagania)*

Opisy języków programowania powinny z jednej strony ułatwiać programiście pisanie programów (generowanie słów należących do języka), z drugiej zaś umożliwiać automatyczne sprawdzanie (rozpoznawanie), czy program istotnie należy do języka.

## **Definicja języka programowania zawiera:**

- opis *składni* (syntaktyki), czyli definicję zbioru napisów, będących poprawnymi programami;
- opis *semantyki*, czyli znaczenia programów;
- ewentualnie *system wnioskowania*, służący do dowodzenia poprawności programów, tj. ich zgodności ze specyfikacją.

# *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 deklaratywnym. W odróżnieniu od Scali 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 w językach OCaml i Scala z tego wykładu).

# Wykorzystanie strumieni w języku Java - 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 funkcjonałów z języków funkcyjnych:

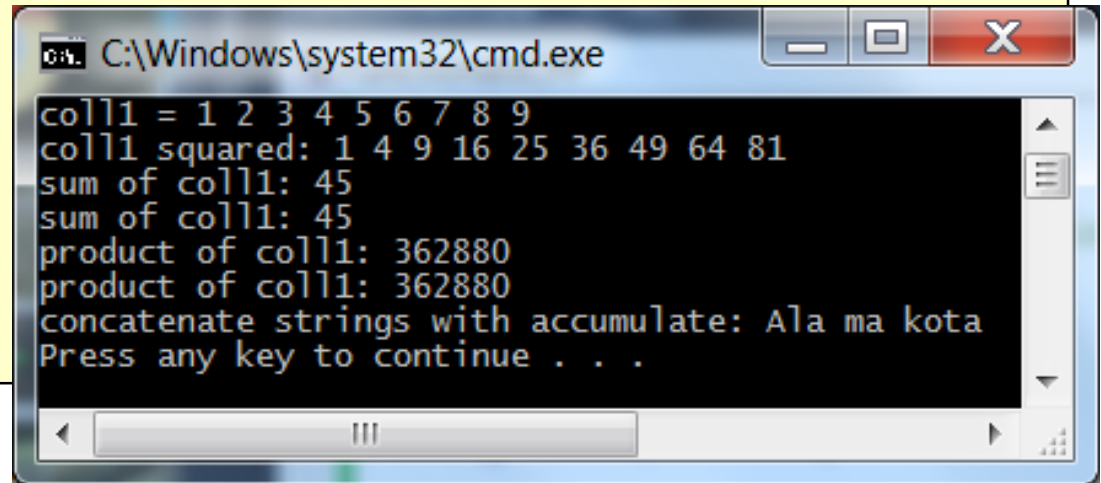
find	std::find_if
map	std::transform
filter	std::copy_if (std::remove_if razem z std::erase)
fold_left	std::accumulate
fold_right	W bibliotece C++ nie ma odpowiednika funkcjonału fold_right, ale można zasymulować jego działanie, przekazując do std::accumulate iteratory odwrotne rbegin, rend (lub crbegin,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

# *Dodatek*

## *Języki formalne - definicje*

*Alfabet*  $A$  jest skończonym zbiorem *symboli*. *Słowem* lub *napisem* (*łańcuchem*) nad alfabetem  $A$  nazywamy skończony ciąg symboli z tego alfabetu. Słowem jest również ciąg pusty oznaczany przez  $\varepsilon$ . Długością  $|w|$  słowa  $w$  nazywamy liczbę symboli w słowie. Przez  $A^*$  oznaczamy zbiór wszystkich słów nad alfabetem  $A$ , zaś przez  $A^+$  zbiór wszystkich niepustych słów nad tym alfabetem.

W zbiorze  $A^*$  definiujemy dwuargumentową relację *konkatenacji* (*złożenia*) słów: jeśli  $x=a_1...a_m$  oraz  $y=b_1...b_n$ , gdzie  $a_i, b_j \in A$  to ich konkatenacją jest słowo  $xy = a_1...a_m b_1...b_n$ . Operacja konkatenacji jest łączna  $((w_1 w_2) w_3 = w_1 (w_2 w_3))$  i posiada element neutralny  $\varepsilon$  ( $w\varepsilon = \varepsilon w = w$ ).

*Językiem* (*formalnym*) nad alfabetem  $A$  nazywamy dowolny zbiór słów nad tym alfabetem, czyli dowolny podzbiór zbioru  $A^*$ .

# Gramatyki formalne

Gramatyką nazywamy czwórkę  $G = (N, T, P, S)$ , gdzie:

- $N$  - skończony zbiór *symboli nieterminalnych* (zmiennych syntaktycznych, kategorii syntaktycznych);
- $T$  - skończony zbiór *symboli terminalnych* (alfabet), rozłączny z  $N$ ;
- $P$  - skończony zbiór *produkcji* postaci  $a \rightarrow b$ , gdzie  $a \in (N \cup T)^+$ ,  $b \in (N \cup T)^*$  ;
- $S$  - wyróżniony symbol nieterminalny, nazywany *symbolem początkowym*.

Nieformalnie: Język  $L(G)$ , generowany przez gramatykę  $G$  jest zbiorem wszystkich słów zbudowanych z symboli terminalnych, które da się otrzymać z symbolu początkowego za pomocą przekształceń, określonych przez reguły produkcji. Zasada stosowania produkcji  $a \rightarrow b$  : jeśli  $a$  jest pod słowem już wygenerowanego słowa, to  $a$  można zastąpić w tym słowie przez  $b$ .

J.E.Hopcroft, R.Motwani, J.D.Ullman. *Wprowadzenie do teorii automatów, języków i obliczeń*. PWN, Warszawa, 2005

# Hierarchia Chomsky'ego

Niech  $G = (N, T, P, S)$  i  $V = N \cup T$ . Gramatyka  $G$  jest gramatyką

- *typu 0 (bez ograniczeń)*, jeśli każda produkcja jest postaci  $u \rightarrow w$ ,  $u \in V^+$ ,  $w \in V^*$  ;
- *typu 1 (kontekstowa)*, jeśli każda produkcja jest postaci  $uAw \rightarrow uzv$ ,  
 $u, w \in V^*$ ,  $A \in N$ ,  $z \in V^+$  ;
- *typu 2 (bezkontekstowa - GBK)*, jeśli każda produkcja jest postaci  $A \rightarrow z$ ,  
 $A \in N$ ,  $z \in V^+$  ;
- *typu 3 (regularna)*, jeśli każda produkcja jest postaci  
 $A \rightarrow bB$  (gramatyka prawostronnie regularna),  
lub każda produkcja jest postaci  
 $A \rightarrow Bb$  (gramatyka lewostronnie regularna),  
 $A \in N$ ,  $B \in N \cup \{\varepsilon\}$ ,  $b \in T^+$  ;

Gramatyki typu 1,2,3 są *gramatykami nieskracającymi*, z czego wynika ich *rozstrzygalność*. Każda gramatyka typu  $i$  jest jednocześnie gramatyka typu  $j$ , dla  $0 \leq j \leq i$ , ale nie odwrotnie. Jeśli gramatyka  $G$  generująca język  $L(G)$  jest kontekstowa (bezkontekstowa, regularna) to język  $L(G)$  też jest nazywany kontekstowym (bezkontekstowym, regularnym) .

# Przykłady gramatyk

Im wyższy typ gramatyki, tym mniej precyzyjnie opisuje zbiór wywodliwych słów.

- Przykład 1. Gramatyka regularna (typu 3)

Niech  $L_3 = \{a^k b^l c^m \mid k, l, m \geq 1\}$ .

$G_3 = (\{S, V, U\}, \{a, b, c\}, P, S)$ , gdzie

$P = \{ S \rightarrow aS \mid aV, V \rightarrow bV \mid bU, U \rightarrow cU \mid c \}$ .

- Przykład 2. Gramatyka bezkontekstowa (typu 2)

Niech  $L_2 = \{a^k b^l c^m \mid k, l, m \geq 1 \text{ i } k = m\}$ .

$G_2 = (\{S, V\}, \{a, b, c\}, P, S)$ , gdzie

$P = \{ S \rightarrow aSc \mid aVc, V \rightarrow Vb \mid b \}$ .

- Przykład 3. Gramatyka kontekstowa (typu 1)

Niech  $L_1 = \{a^k b^l c^m \mid k, l, m \geq 1 \text{ i } k = l = m\}$ .

$G_1 = (\{S, V\}, \{a, b, c\}, P, S)$ , gdzie

$P = \{ S \rightarrow abc \mid aSVc, cV \rightarrow Vc, bV \rightarrow bb \}$ .

# *Wyrażenia regularne - zastosowania*

Wyrażenia regularne mają siłę wyrazu gramatyk regularnych.

- **Znajdowanie wzorców w tekście.**

Użytkownicy systemów UNIX-owych znają komendę `grep` (Global search for Regular Expressions and Print), pozwalająca na wyszukiwanie w plikach tekstowych fragmentów opisanych za pomocą wyrażen regularnych.

Biblioteki wszystkich współczesnych języków programowania również umożliwiają wykorzystanie wyrażen regularnych przy przetwarzanie tekstu. Na przykład w języku Java odpowiednie klasy zawarte są w pakiecie `java.util.regex`.

- **Leksery i generatory lekserów (LEX).**

# Języki bezkontekstowe - zastosowania

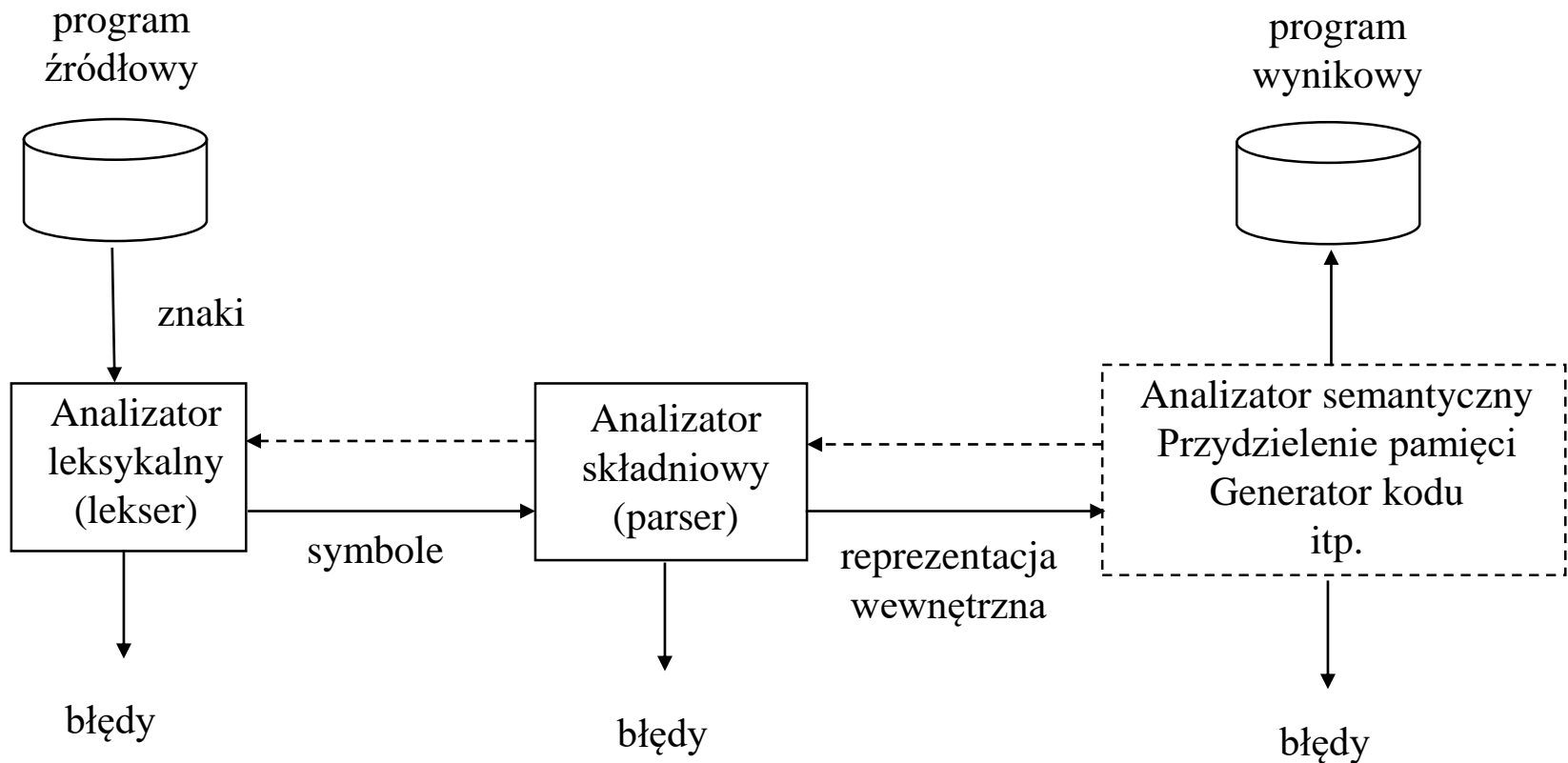
- Opis składni języków programowania.
- Parsery i generatory parserów (YACC).
- XML i definicja typu dokumentu.

Składnia języków programowania przedstawiona jest zwykle w *notacji BNF* (Backus-Naur Form), na przykład:

- *Java Language and Virtual Machine Specification*, <http://docs.oracle.com/javase/specs/>
- B.W.Kernighan, D.M.Ritchie, *Język ANSI C*, WNT, Warszawa 1997, Dodatek A: Przewodnik języka C.
- *Język C++* <https://www.iso.org/obp/ui/#iso:std:iso-iec:14882:ed-4:v1:en>
- *Ocaml. Documentation and user's manual*, Chapter 6: The OCaml Language <http://caml.inria.fr/pub/docs/manual-ocaml/index.html>
- P.Van Roy, S.Haridi. *Programowanie. Koncepcje, techniki i modele*. Helion 2005, Dodatek C: Składnia języka
- Scala Language Specification, <http://www.scala-lang.org/files/archive/spec/>

Istnieje wiele wariantów i rozszerzeń notacji BNF. Do opisu składni języków bezkontekstowych używane są także *diagramy syntaktyczne* (diagramy składniowe).

# Kompilator jednoprzebiegowy



A.V.Aho, M.S.Lam, R.Sethi, J.D.Ullman. Kompilatory. Reguły, metody, narzędzia. PWN, Warszawa.2019

N.Wirth. *Algorytmy + struktury danych = programy*. WNT, Warszawa, 1980, 1989, 1999,

Rozdział 5, Struktury językowe i kompilatory

W.M.Waite, G.Goos. Konstrukcja kompilatorów. WNT, Warszawa, 1989



# *Analizator leksykalny (lekser)*

Zadaniem *analizatora leksykalnego* jest określenie i pobranie kolejnego *symbolu* (leksemu, tokena), reprezentowanego w ciągu wejściowym przez grupę *znaków*, zgodnie z regułami reprezentacji języka. Struktura leksemów opisywana jest za pomocą wyrażeń regularnych.

Celem analizy leksykalnej jest więc przekształcenie programu źródłowego, przedstawionego jako *ciąg znaków* w *ciąg symboli*.

Analizator leksykalny wykonuje m.in. następujące czynności:

- opuszcza białe znaki (odstęp, znaki tabulacji, znaki końca wiersza itp.)
- rozpoznaje słowa kluczowe (if, then, while itp.)
- rozpoznaje i zapamiętuje identyfikatory
- rozpoznaje ciągi cyfr jako liczby i oblicza ich wartość

Biblioteki wszystkich współczesnych języków programowania zawierają generator lekserów (LEX).

Z każdym tokenem można łączyć akcję semantyczną, co umożliwia łatwe obliczenie np. wartości literałów numerycznych.

# *Analizator składniowy (parser)*

Głównym zadaniem *analizatora składniowego (syntaktycznego)* jest sprawdzenie, za pomocą pewnego *algorytmu rozbioru*, czy ciąg symboli, utworzony przez analizator leksykalny, jest poprawnym słowem zadanego języka. Jako wynik tworzona jest pewna struktura, która może być kodem wynikowym programu, gotowym do wykonania lub interpretacji, częściej jednak jest strukturalną reprezentacją wewnętrzną, będącą podstawą generacji kodu wynikowego. Często do tego celu służy drzewo rozbioru.

Najefektywniejsza jest analiza składniowa bez powrotów z wyprzedzeniem o jeden symbol, tzn. kolejny krok analizy zależy tylko od stanu obliczeń i bieżącego symbolu i żadnego kroku analizy nie można cofnąć.

We wszystkich systemach UNIX-owych oraz w bibliotekach wszystkich współczesnych języków programowania można znaleźć generatory parserów (YACC - Yet Another Compiler Compiler), które na podstawie wejściowej gramatyki bezkontekstowej tworzą drzewo rozbioru.

Z każdą produkcją gramatyki można łączyć akcję semantyczną, która jest fragmentem kodu w odpowiednim języku (C, C++, Java, OCaml, itp), która jest wykonywana za każdym razem, kiedy tworzony jest wierzchołek drzewa rozbioru odpowiadający tej produkcji.