Paradygmaty programowania

WPROWADZENIE

dr inż. Łukasz Bartczuk

Katedra Inteligentnych Systemów Informatycznych

p. 517

lukasz.bartczuk@pcz.pl

Konsultacje:

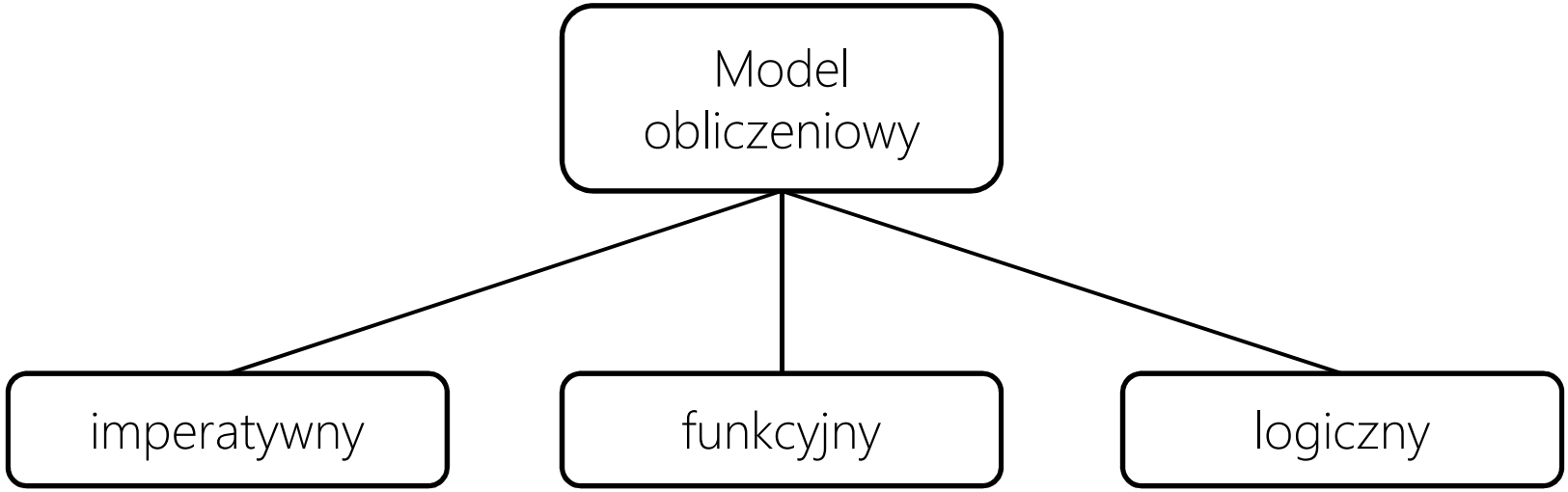
Poniedziałek: 11-13
Czwartek: 9-10

Co to jest komputer?

Co to jest program komputerowy?

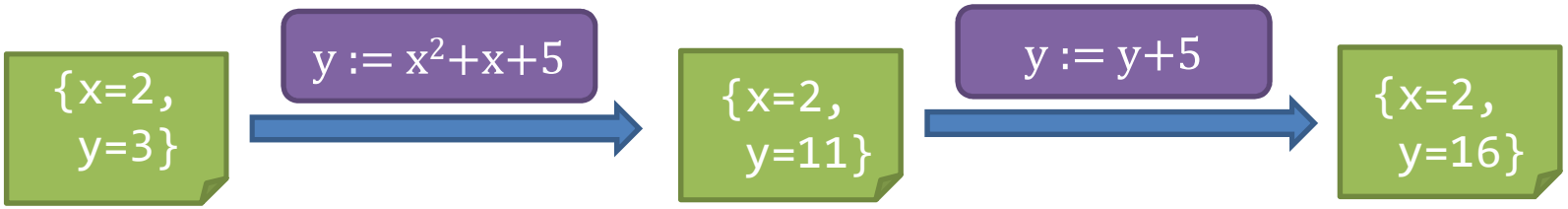
Model obliczeniowy

Model obliczeniowy jest to zbiór wartości (prostych lub złożonych), skojarzonych z nimi operacji oraz operacji wykorzystywanych do zdefiniowania obliczeń.



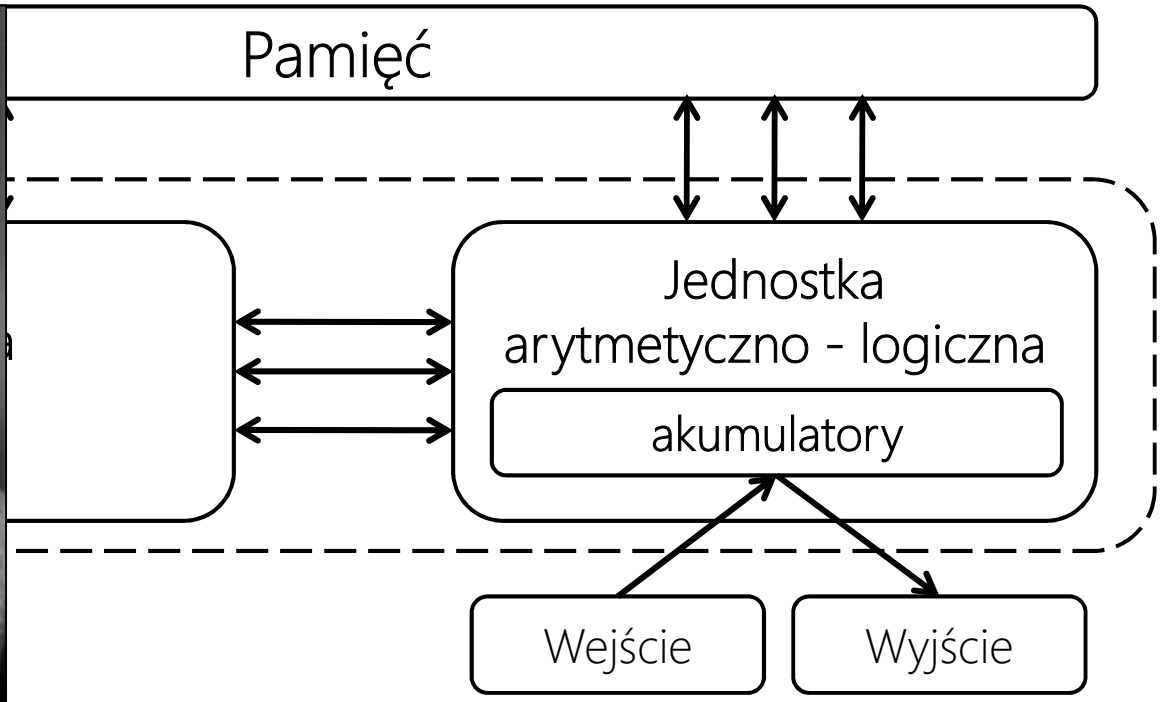
Model imperatywny

Model imperatywny składa się wartości oraz stanu i operacji przypisania.



Maszyna John'a von Neumanna

1903-1957



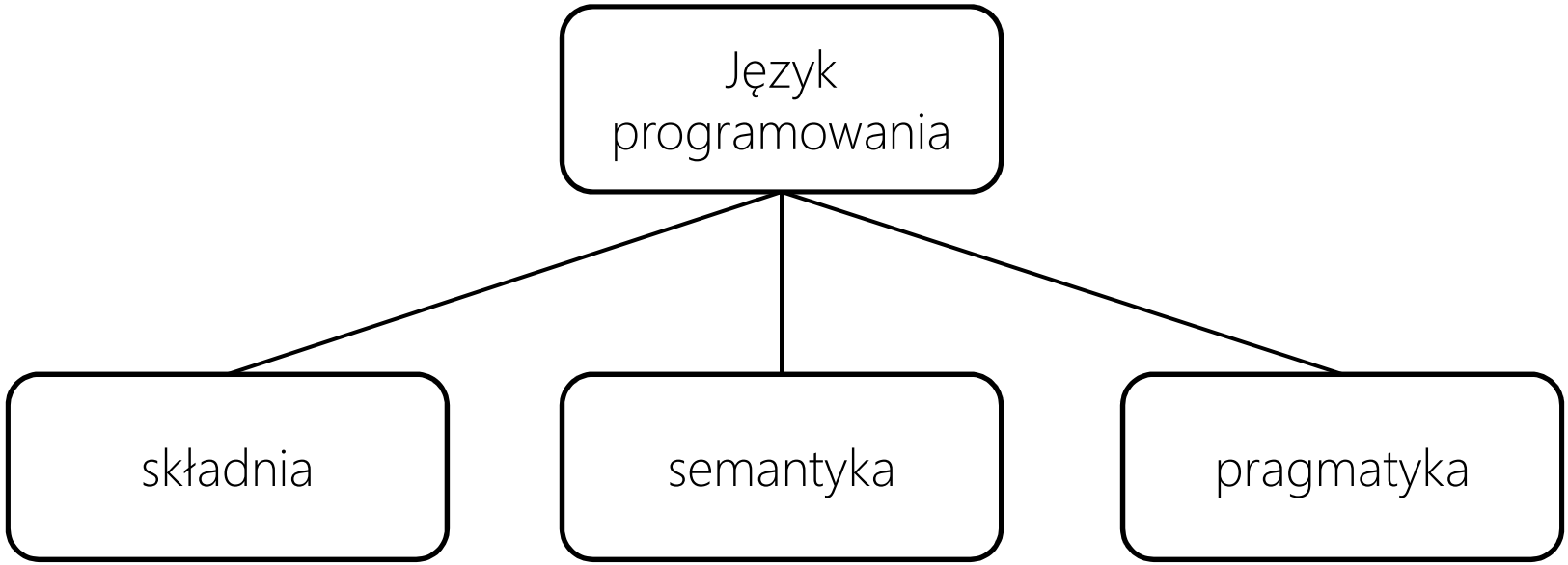
Model funkcyjny

Model ten składa się z wartości i funkcji (przy czym funkcje też są wartością), a podstawową operacją do wykonywania obliczeń jest aplikacja funkcji do wartości.

$f(x) = x^2+x+5$

$g(y) = y+5$

Co to jest język programowania?



Składnia języków programowania

Zbiór zasad określających, który ciąg znaków jest poprawną instrukcją w danym języku programowania.

Do definiowania składni języka programowania najczęściej używa się notacji BNF (Backus-Naur Form)

```
<expression> ::= <term> |
                 <expression> '+' <term> |
                 <expression> '-' <term>
<term>       ::= <factor> |
                 <term> '*' <factor> |
                 <term> '/' <factor>
<factor> ::= <number> | '(' <expression> ')'
```

Semantyka języków programowania

Określa relację pomiędzy elementami składni języka programowania, a modelem obliczeniowym.

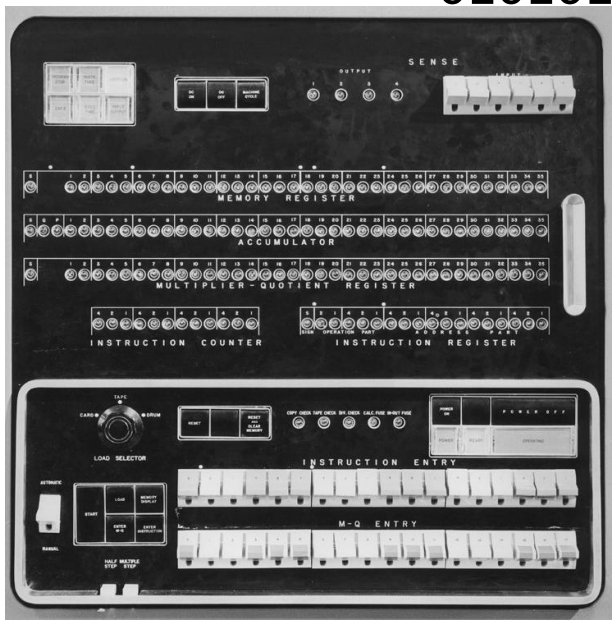
- Semantyka aksjomatyczna
- Semantyka denotacyjna
- Semantyka operacyjna

Pragmatyka języków programowania

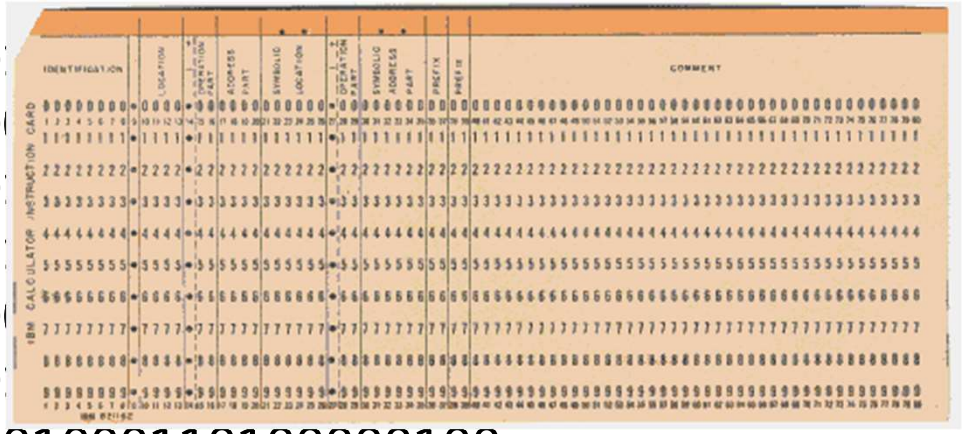
Opisuje praktyczne aspekty implementacji języka i korzystania z niego. Określa m.in. jak konstrukcje i cechy języka mogą być wykorzystane do osiągnięcia określonego celu.

Ewolucja języków programowania

Początkowo programy wyglądały tak:




```
0101010110001000
0000010000
0000000101
0011000111
1000000000
1001000101
01111101001000110100000100
10111101000100010111111100
00000000000000000000000000
00110010011100001110010000
```



Ewolucja języków programowania

Funkcja w języku assembler (nowoczesnym)

1919 - 2001

```
push ebp
mov ebp, esp
sub esp, 0x10
mov DWORD PTR [ebp-0xc], 0x5
mov DWORD PTR [ebp-0x8], 0x6
mov eax, DWORD PTR [ebp-0x8]
mov edx, DWORD PTR [ebp-0xc]
lea eax, [edx+eax*1]
mov DWORD PTR [ebp-0x4], eax
mov eax, 0x0
leave
ret
```

Ewolucja języków programowania

... i w C

```
int main()
{
  int a = 5;
  int b = 6;
  int c = a+b;
  return 0;
}
```

Ewolucja języków programowania

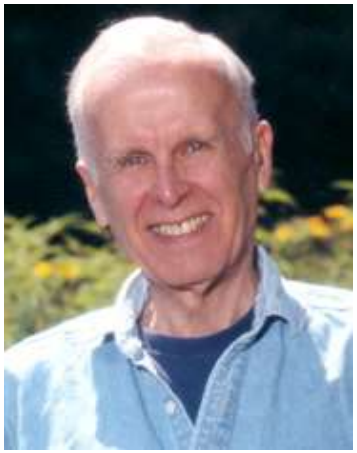
Twórca pierwszego kompilatora

kadm. Grace Hooper

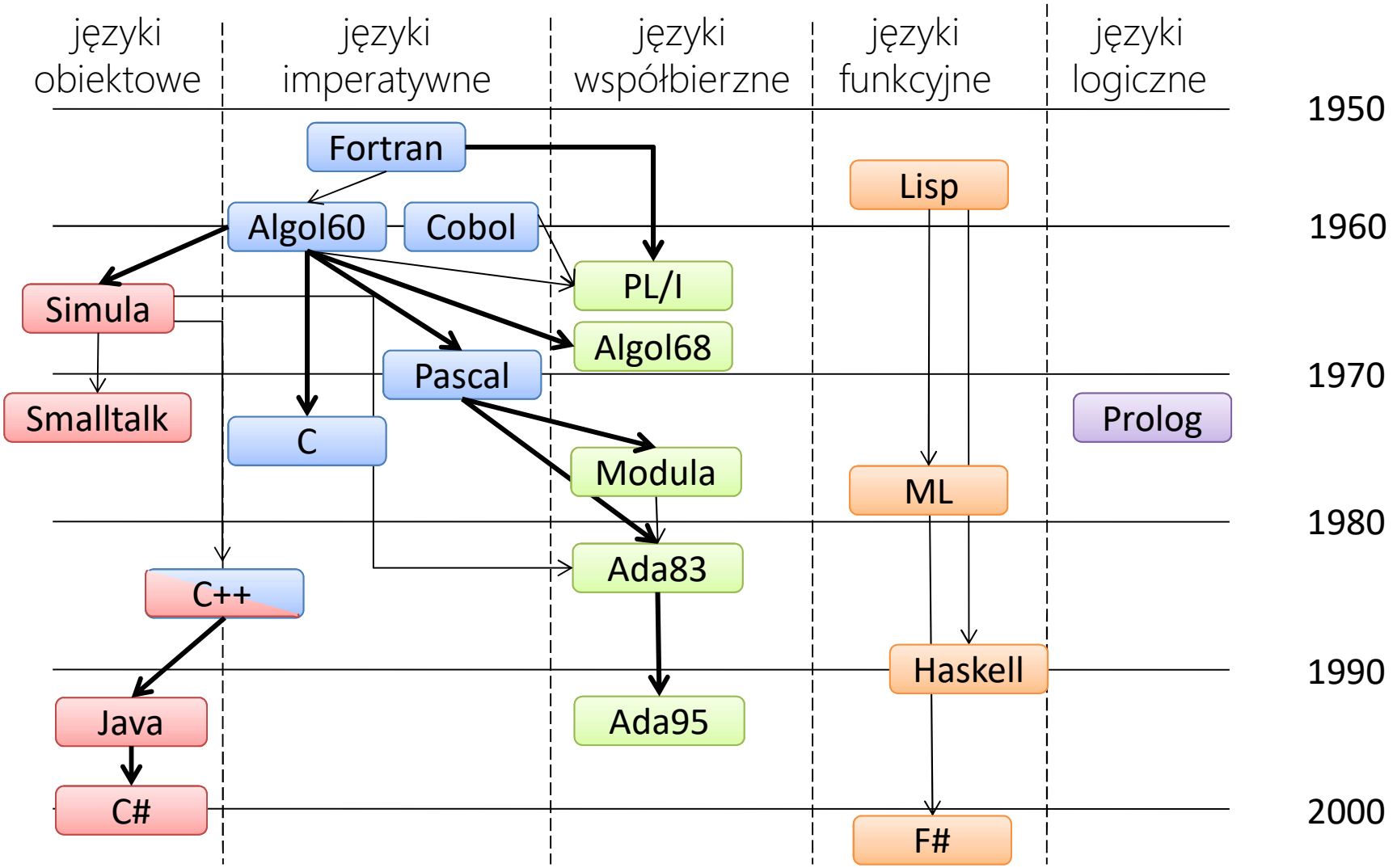
1906 - 1992

Twórca języka Fortran

John Backus



1924 - 2007

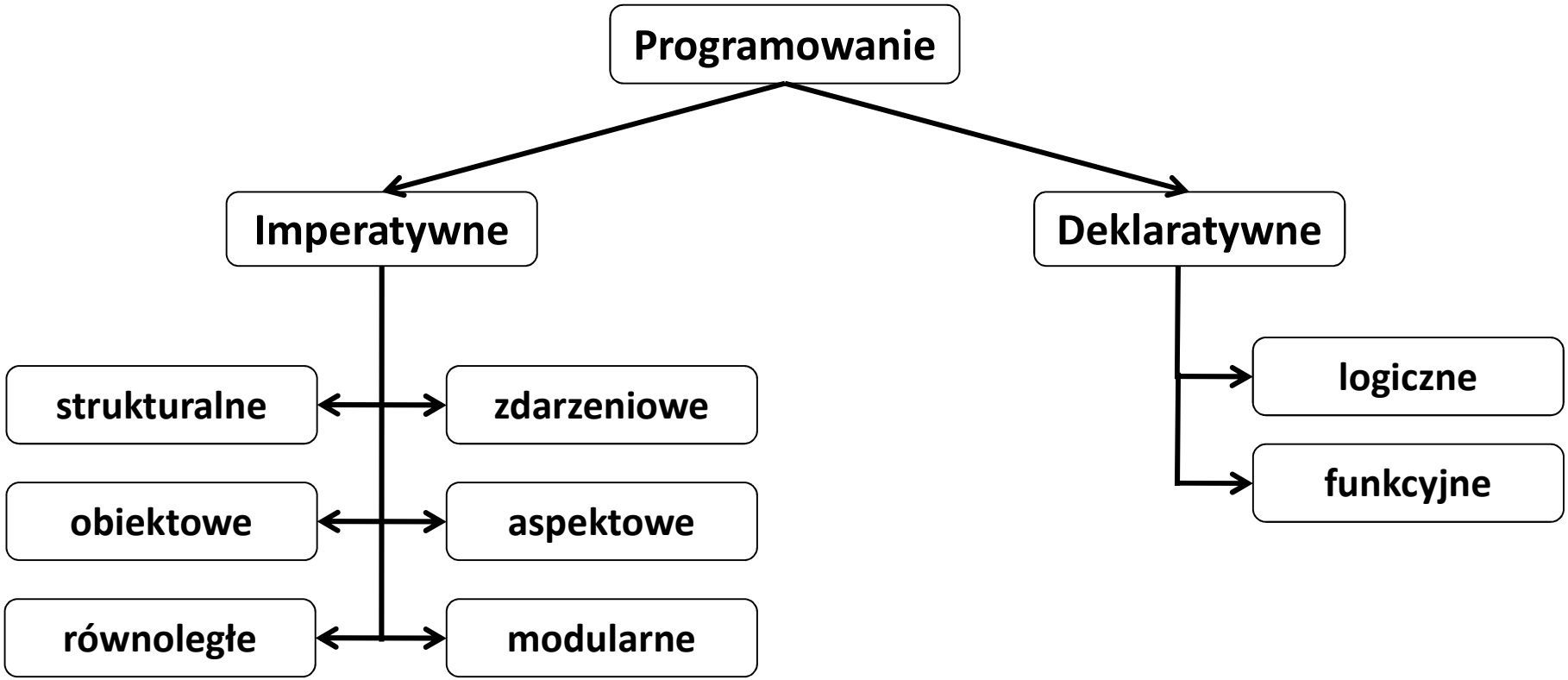

języki obiektowe
języki imperatywne
języki współbierzne
języki funkcyjne
języki logiczne
1950
1960
1970
1980
1990
2000
Fortran
Algol60
Cobol
Lisp
PL/I
Simula
Algol68
Pascal
Smalltalk
C
Prolog
Modula
ML
Ada83
C++
Haskell
Java
Ada95
C#
F#

Paradygmat programowania

Paradygmat programowania jest wyróżniającym się stylem programowania. Każdy paradygmat jest scharakteryzowany przez dominację pewnych kluczowych koncepcji.

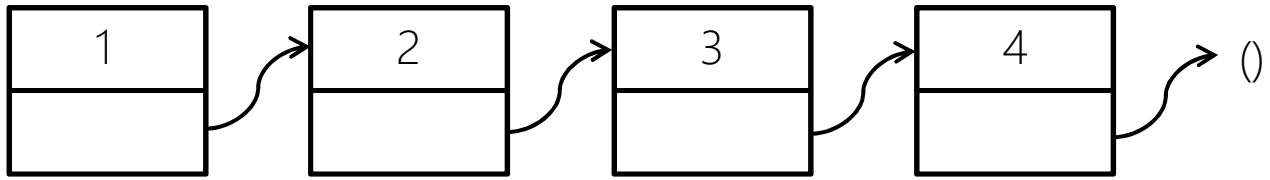
- zbiór koncepcji reprezentujących podejście do implementacji algorytmów
- zbiór mechanizmów używanych przez programistę do pisania programów i określających jak te programy będą następnie wykonywane przez komputer

Paradygmat programowania



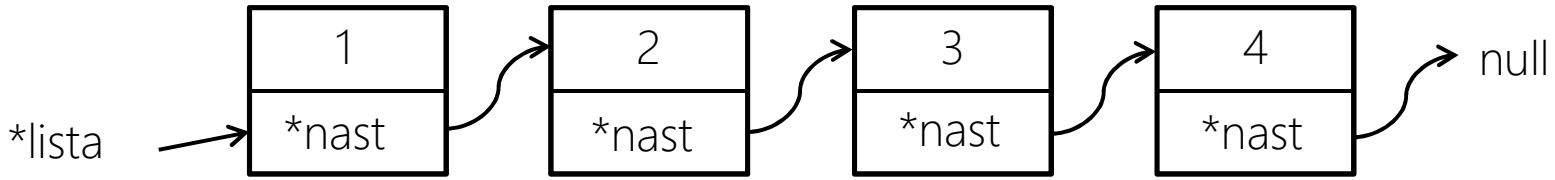
Problem do rozwiązania

Dana jest lista liczb całkowitych:



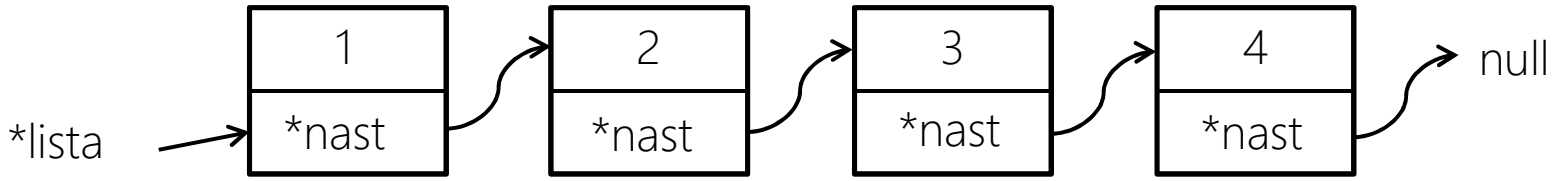
Napisać funkcję, która będzie obliczała sumę elementów na liście

Programowanie imperatywne



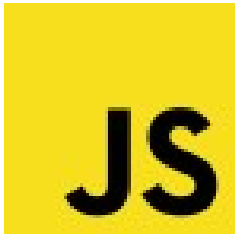
1. Stwórz zmienną **`biezacy`** określającą aktualny element listy i przypisz do niej adres pierwszego elementu na liście (parametr funkcji)
2. Stwórz zmienną **`suma`** przechowującą sumę wartości kolejnych elementów listy
3. W pętli (dopóki zmienna **`biezacy`** ma inną wartość niż **`null`**):
    1. Pobierz wartość składowej **`wartosc`** aktualnego elementu listy, pobierz wartość zmiennej **`suma`** zawierającą sumę wszystkich dotychczasowych elementów. Dodaj te dwie wartości do siebie. Wynik przypisz do zmiennej **`suma`**.
    2. Pobierz wartość wskaźnika **`nast`** aktualnego elementu na liście i przypisz go do zmiennej **`biezacy`**
4. Zwróć wartość zmiennej **`suma`** do programu wywołującego

Programowanie imperatywne

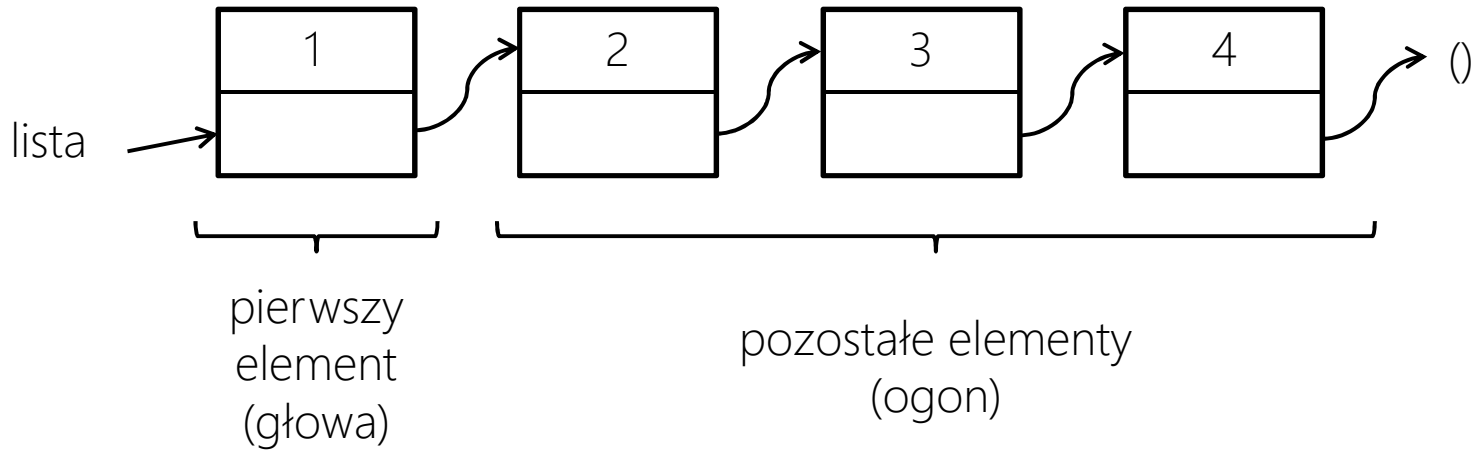



```
int Sumuj(Lista* lista) {
   Lista* biezacy = lista;
   int suma = 0;
   while(biezacy != NULL) {
      suma = suma + biezacy->wartosc;
      biezacy = biezacy->nast;
   }
   return suma;
}
```

Programowanie imperatywne

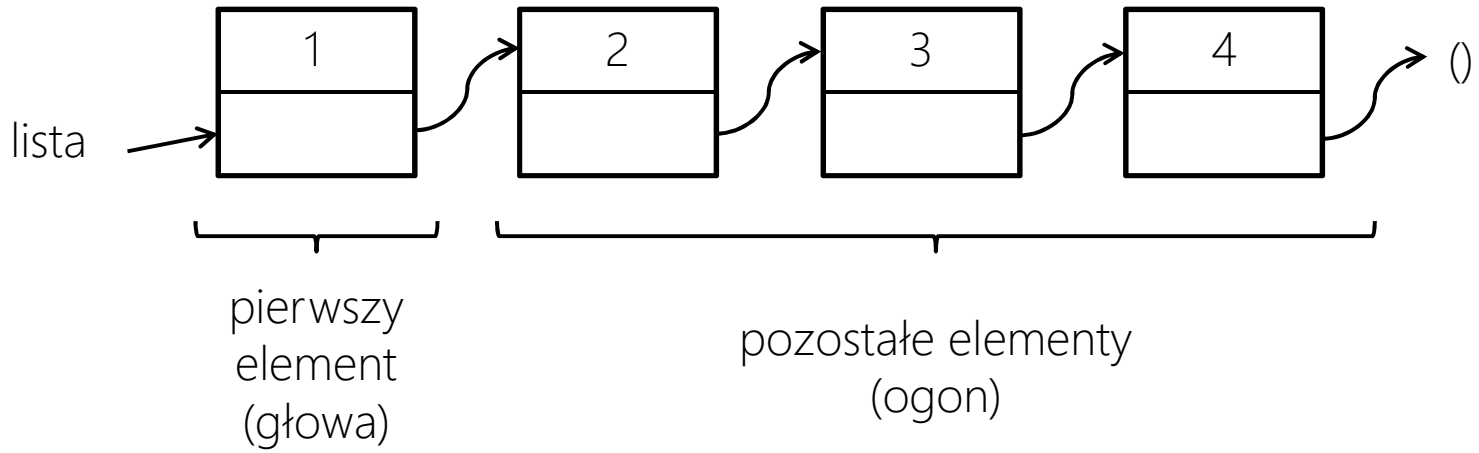


Programowanie funkcyjne



1. Jeżeli lista jest pusta to zwróć 0.
2. Jeżeli lista nie jest pusta to pobierz wartość pierwszego elementu na liście i dodaj go do sumy pozostałych elementów na liście

Programowanie funkcyjne



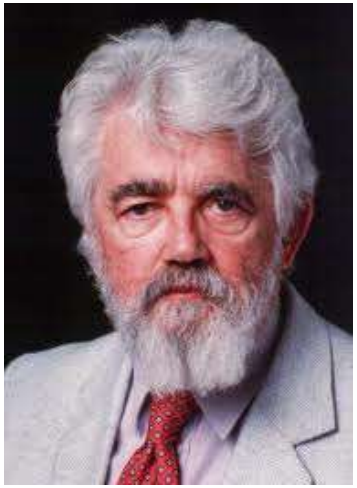

```
let rec sumuj lista =
    if List.isEmpty lista then
       0
    else
       lista.Head + sumuj lista.Tail;;
```

Programowanie funkcyjne

Alonzo Church
(twórca rachunku lambda)

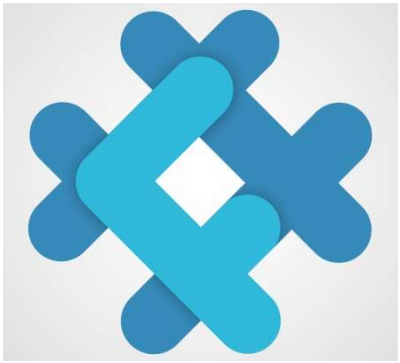
1903 - 1995

John McCarthy
(projektant języka Lisp)

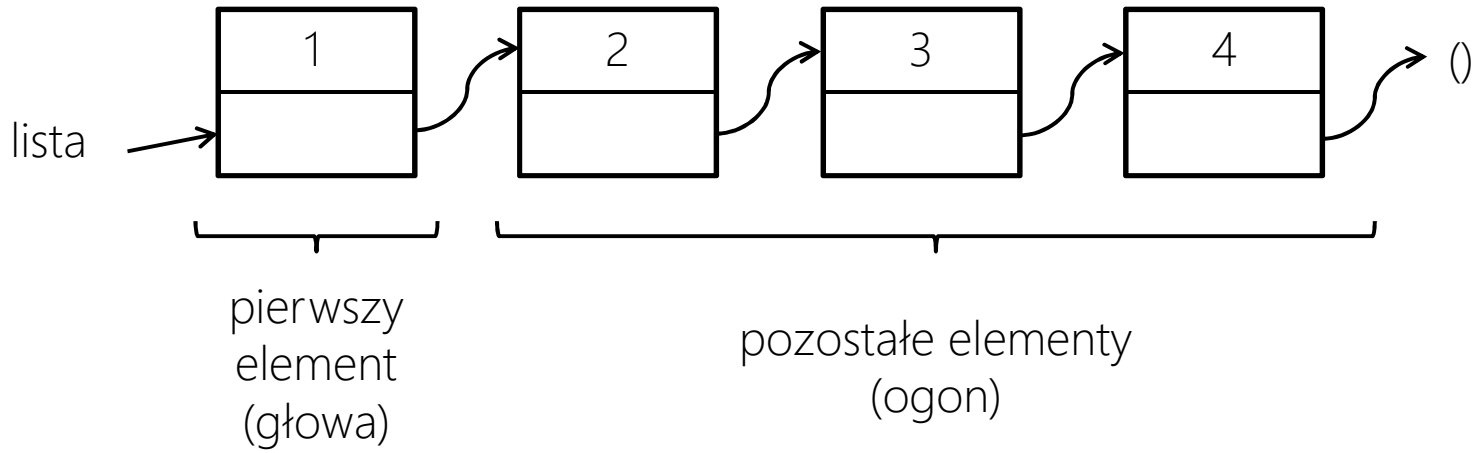


1927 - 2011

Programowanie funkcyjne

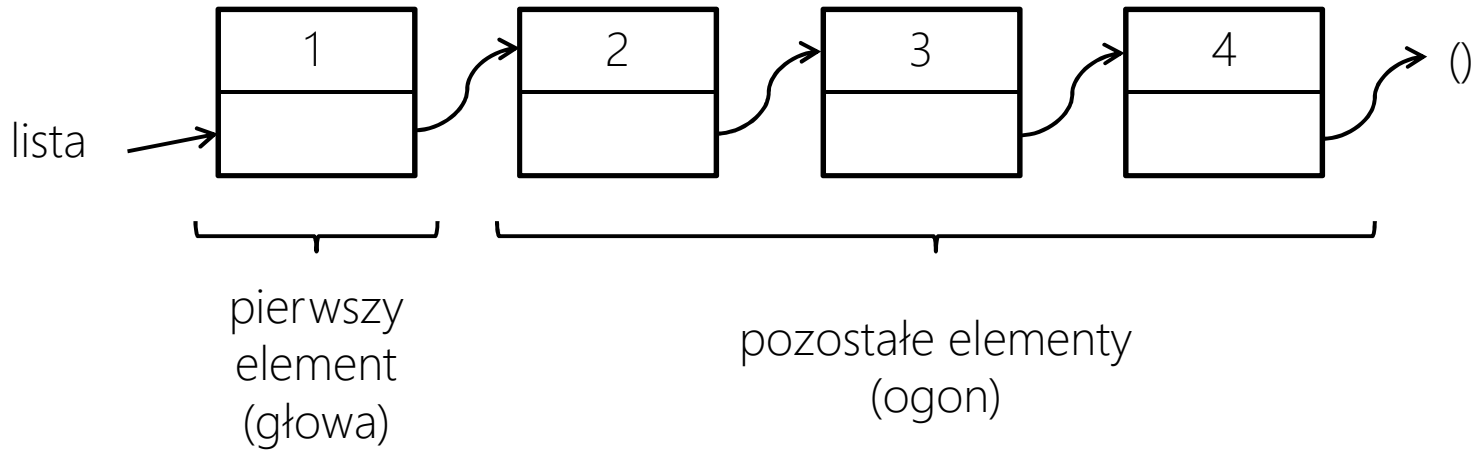


Programowanie logiczne



1. Jeżeli lista jest listą pustą to prawdą jest, że suma jej elementów Y równa się 0.
2. Jeżeli lista ma głowe X i ogon Xs oraz suma elementów w ogonie to Suma, to prawdą jest, że odpowiedzią jest X plus Suma

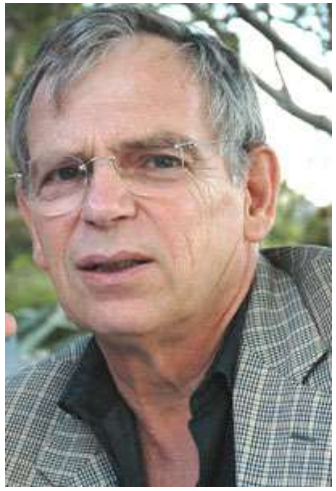
Programowanie logiczne




```
sumuj([], Y) :- Y is 0.
sumuj([X|Xs], Y) :- sumuj(Xs, Suma), Y is Suma+X.
```

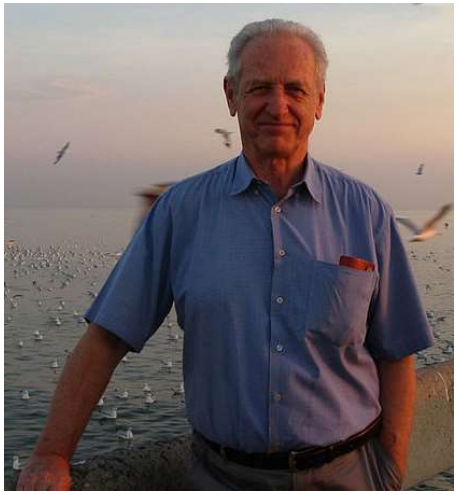
Programowanie logiczne

Alain Colmerauer
Twórca języka Prolog



1941-

Robert "Bob" Anthony Kowalski
Twórca proceduralnej interpretacji klauzul Horna

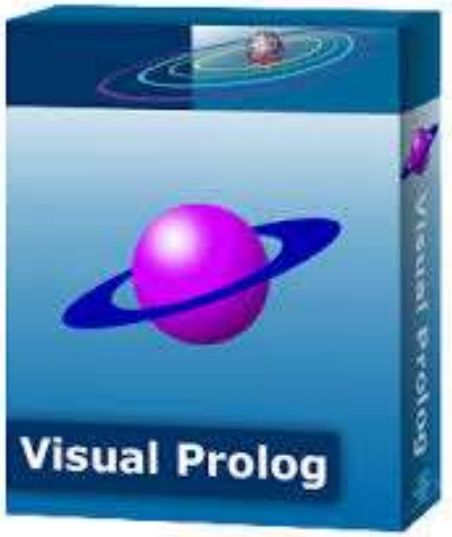


1941-

Programowanie logiczne

ProLog

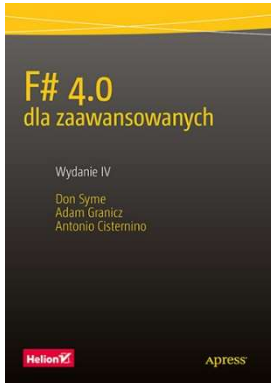
datalog.ai®

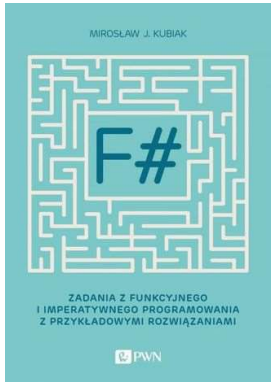


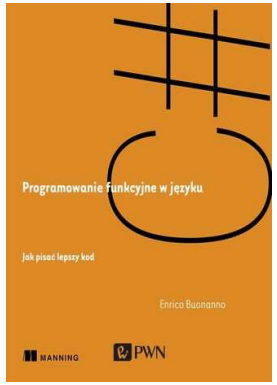
Zaliczenie przedmiotu

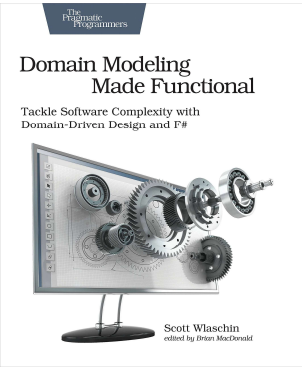
Przedmiot kończy się zaliczeniem

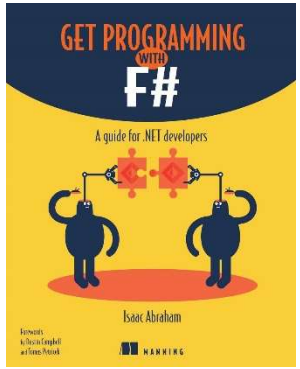
Literatura

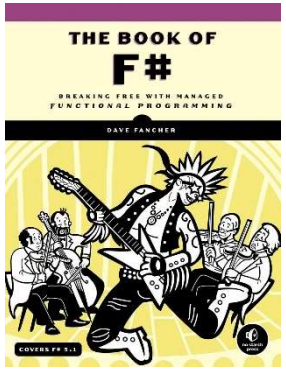

F# 4.0
dla zaawansowanych
Wydanie IV
Don Syme
Adam Granicz
Antonio Cisternino
Helion
Apress

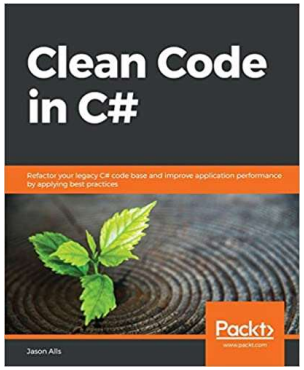

MIROSŁAW J. KUBIAK
F#
ZADANIA Z FUNKCYJNEGO
I IMPERATYWNEGO PROGRAMOWANIA
Z PRZYKŁADOWYMI ROZWIĄZANIAMI
PWN

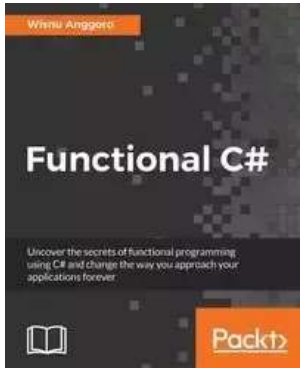

C#
Programowanie funkcyjne w języku
Jak pisać lepszy kod
Enrico Buonanno
MANNING
PWN


The Pragmatic Programmers
Domain Modeling
Made Functional
Tackle Software Complexity with
Domain-Driven Design and F#
Scott Wlaschin
edited by Brian MacDonald


GET PROGRAMMING
WITH
F#
A guide for .NET developers
Isaac Abraham
MANNING

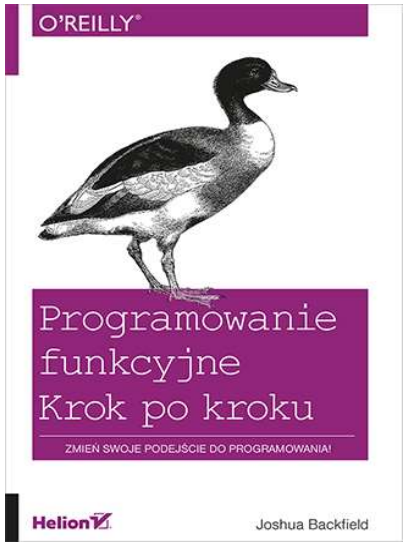

THE BOOK OF
F#
BREAKING FREE WITH MANAGED
FUNCTIONAL PROGRAMMING
DAVE FANCHER

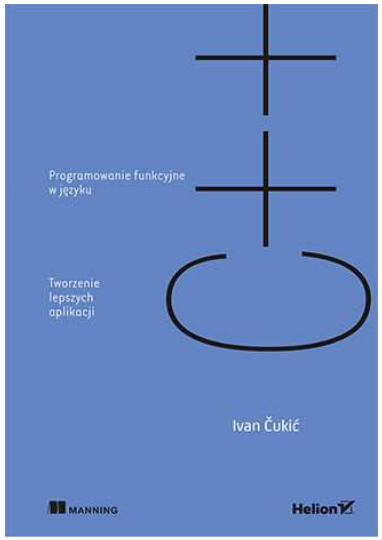

Clean Code
in C#
Refactor your legacy C# code base and improve application performance
by applying best practices
Jason Alls
Packt


Wisnu Anggoro
Functional C#
Uncover the secrets of functional programming
using C# and change the way you approach your
applications forever
Packt

Literatura

Luis Atencio
Programowanie funkcyjne
z JavaScriptem
Sposoby na lepszy kod
Helion


O'REILLY
Programowanie funkcyjne
Krok po kroku
ZMIEŃ SWOJE PODEJŚCIE DO PROGRAMOWANIA!
Helion
Joshua Backfield


Programowanie funkcyjne w języku
Tworzenie lepszych aplikacji
Ivan Čukić
MANNING
Helion

The Pragmatic Programmers
MICHAEL SWAINE
PROGRAMOWANIE FUNKCYJNE
POZNAJ:
CLOJURE
ELIXIR
HASKELL
SCALA
SWIFT
PWN

Materiały w internecie

→ Coursera
→ Pluralsight

Materiały z przedmiotu

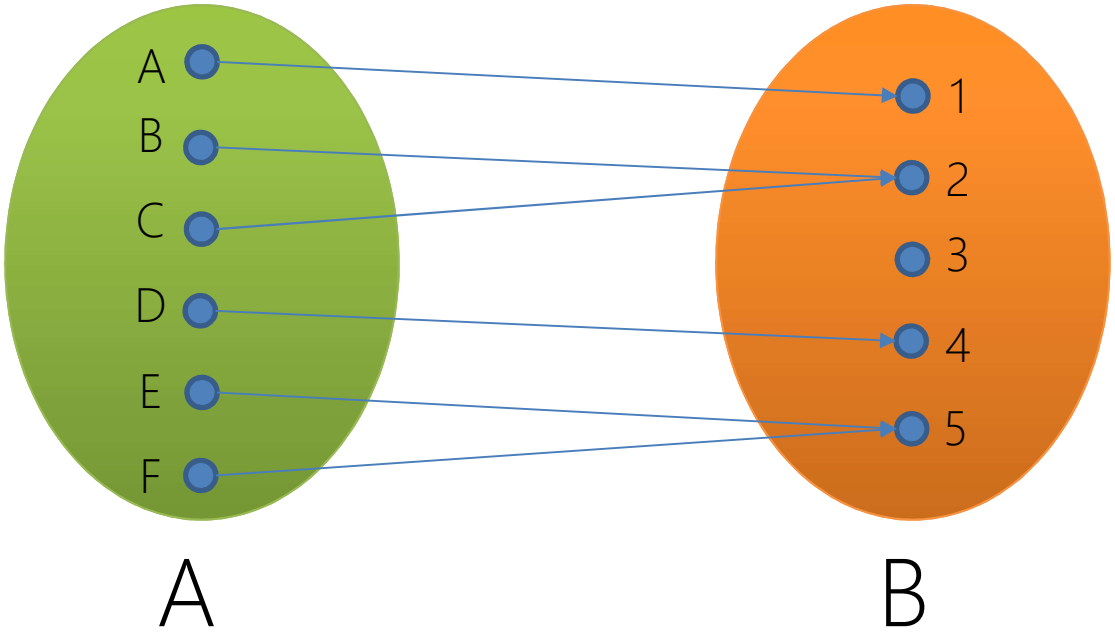
https://1drv.ms/u/s%21AiMGnlQluUxNdrPtpFD4pHWdRak?e=019dO2

http://tiny.cc/pcz-paradygmaty

PROGRAMOWANIE FUNKCYJNE

Programowanie funkcyjne jest
to programowanie
z funkcjami matematycznymi.

Funkcje



$$x \to x^2 + 2x + 3$$

Funkcje

$$f: x \to x^2 + 2x + 3$$

$$f: A \to B$$

$$f: \mathbb{R} \to \mathbb{R}$$

Funkcje

Funkcje matematyczne są funkcjami **czystymi**

Wynik zależy tylko od wartości parametrów funkcji

$$f(3) \rightarrow 3^2 + 2 \cdot 3 + 3 \rightarrow 18$$

Funkcje

Dzięki czystości funkcje możemy zastąpić wartością,
a wartość aplikacją funkcji

Referencyjna trasparentność

$$f(3) \rightarrow 3^2 + 2 \cdot 3 + 3 \rightarrow 18$$

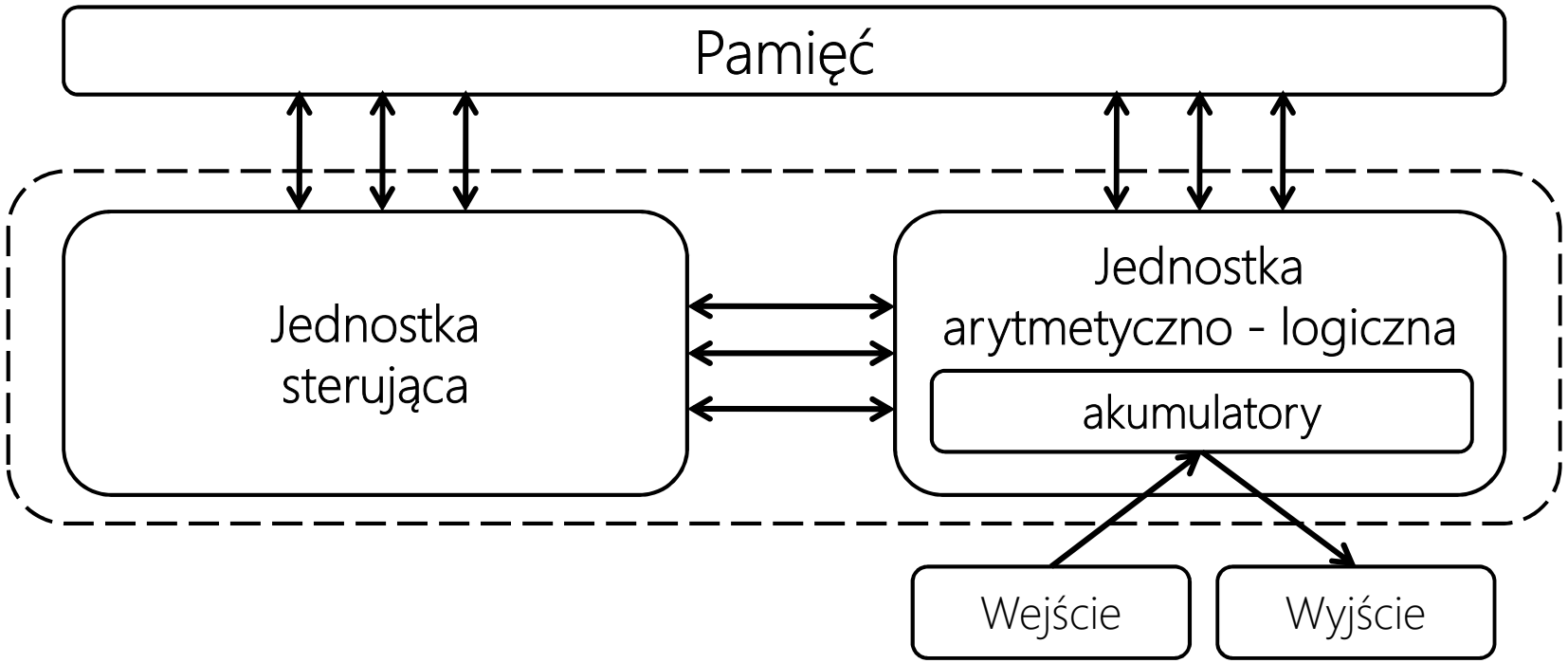
Funkcje

Funkcje mogą mieć więcej niż jeden parametr

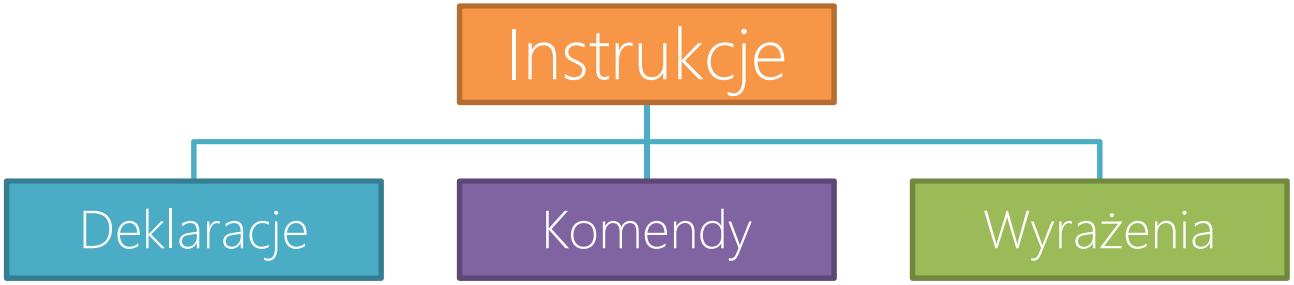
$$f: x, y \rightarrow x^2 + 2y + 3$$

$$f: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

Komputery mają pamięć



Rodzaje instrukcji

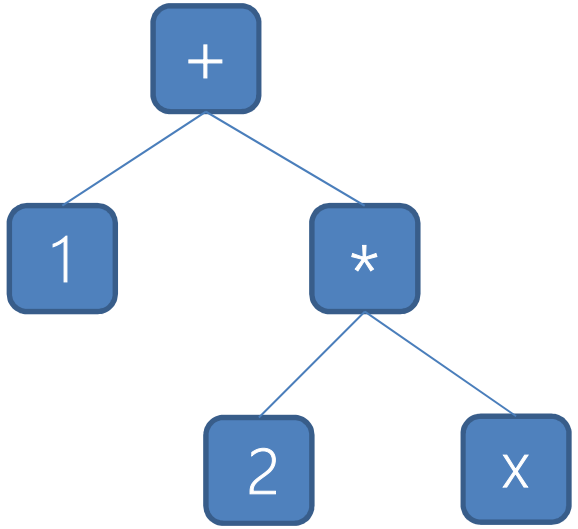



```
var x = 12;
Console.WriteLine(x);
```

Nie wszystkie języki mają wszystkie rodzaje instrukcji

Instrukcje złożone

```
var x = 12;
Console.WriteLine(1+2*x);
Console.ReadKey();
```



Abstrakcje w programowaniu

Procedury

```
void Metoda(int x)
{
   if(x>0)
      Console.WriteLine("Dodatnia");
   else
      Console.WriteLine("Ujemna")
}
```

Funkcje

```
double Oblicz(double x)
{
   return 1+2*x;
}
```

Komendy i wyrażenia

Komendy

- Modyfikują stan programu
- Ich działanie można obserwować tylko poprzez analizę stanu

Wyrażenia

- Tworzą wartości
- Ich rezultat można obserwować poprzez analizę wartości zwracanej

Czyste funkcje

Są to funkcje, które nie mają efektów ubocznych,
a ich wynik zależy tylko od wartości parametrów

```
double Oblicz(double x)
{
    return 1+2*x;
}
```

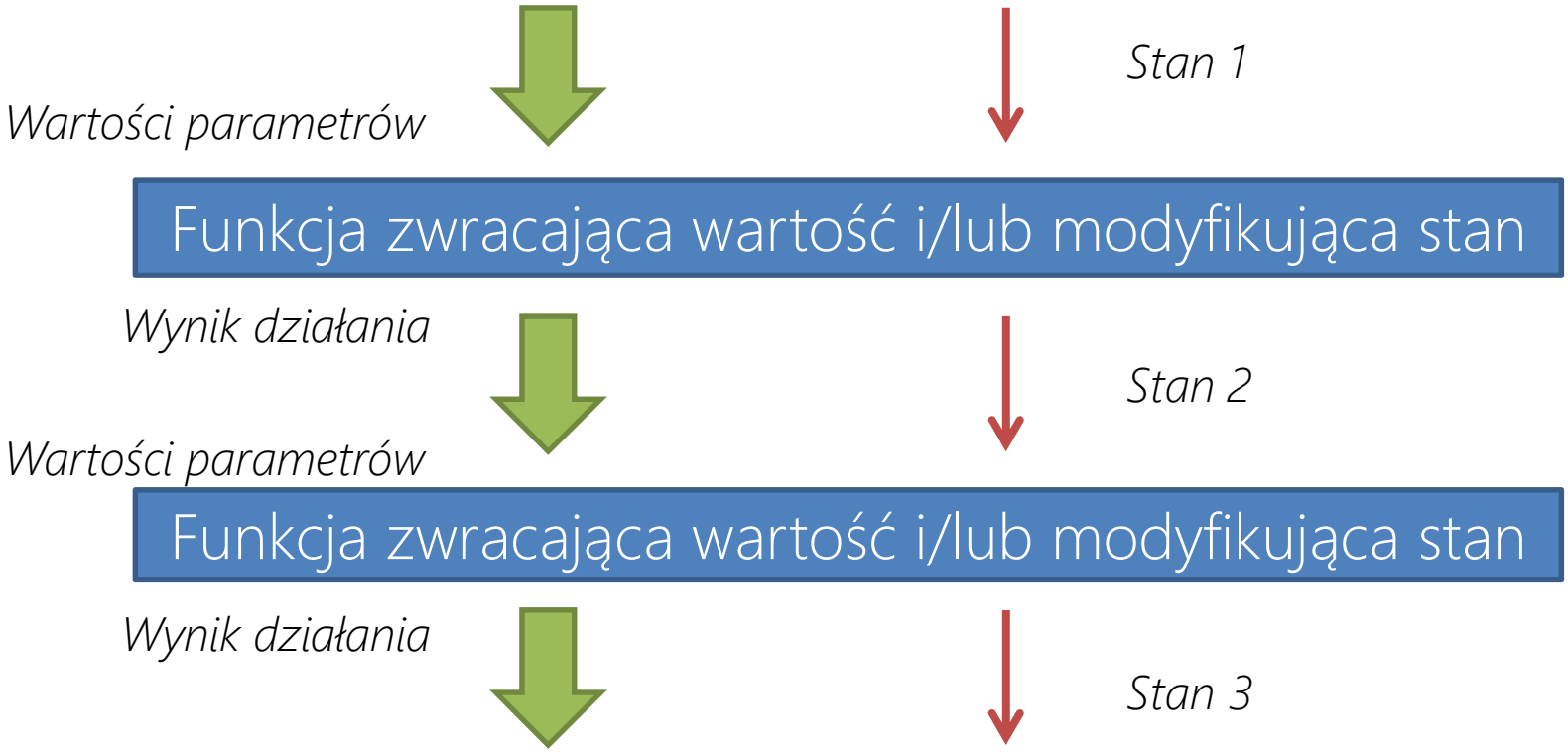
Efekty uboczne

Efekt uboczny – wszelkie działania, które dana instrukcja wykonuje, a które nie są wykonywane przez instrukcje z grupy do której ta instrukcja należy.

```
int x,y;
x = y = 12;
```

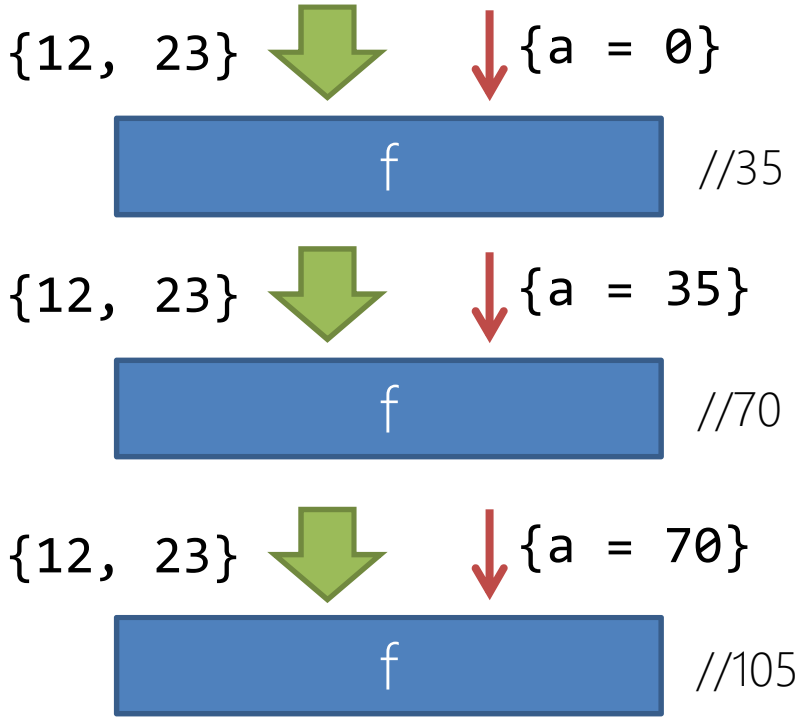
```
class Przyklad {
    private int a = 0;

    public int f(int b, int c) {
        int rezultat = a + b + c;
        a += b + c;
        return rezultat;
    }
}
```

Efekty uboczne



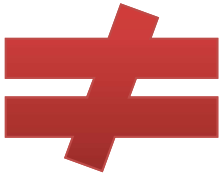
Analiza programu

```
var przyklad = new Przyklad();
Console.WriteLine(przyklad.f(12, 23));
Console.WriteLine(przyklad.f(12, 23));
Console.WriteLine(przyklad.f(12, 23));

class Przykład {
    private int a = 0;

    public int f(int b, int c) {
        int rezultat = a + b + c;
        a += b + c;
        return rezultat;
    }
}
```



Analiza programu

```
f(12, 23) + f(34, 45)
```



```
f(34, 45) + f(12, 23)
```

Wartości niemodyfikowalne

Wartości niemodyfikowalne są to wartości, których składowe mogą być ustalone tylko w konstruktorze.

Wartości niemodyfikowalne

Zwiększają czytelność kodu

W pojedynczym punkcie należy sprawdzać warunki poprawności

Ułatwiają tworzenie aplikacji wielowątkowych

Wymagają większej ilości pamięci

Zwiększają zużycie procesora

Funkcje wyższych rzędów

Funkcje wyższych rzędów, są to funkcje które przyjmują inne funkcje jako swoje parametry, lub tworzą nowe funkcje

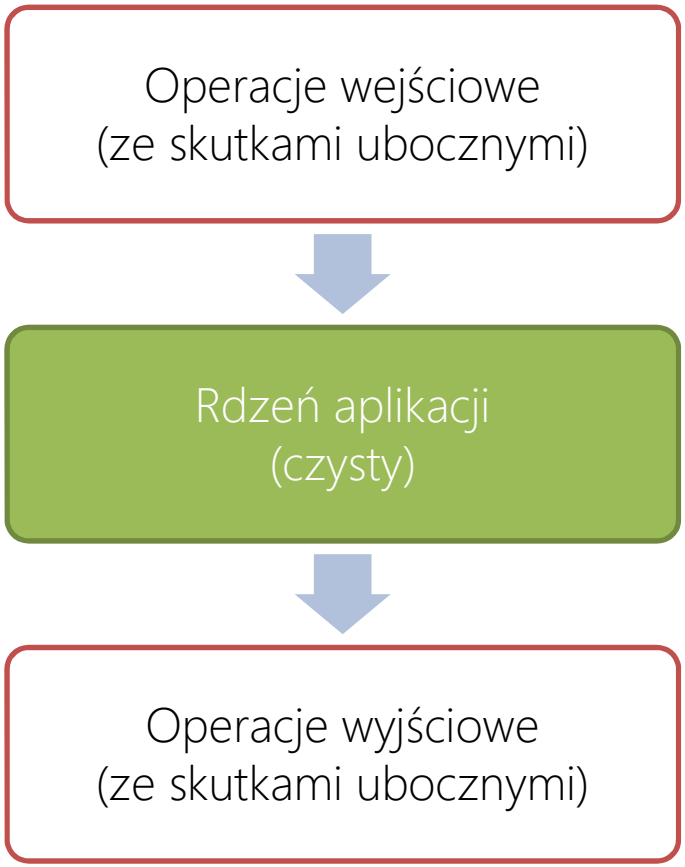
$$\frac{df}{dx} = \frac{d(x^2 + 2x + 3)}{dx} = 2x + 2$$

Tworzenie aplikacji w stylu funkcyjnym

Co mam dane?

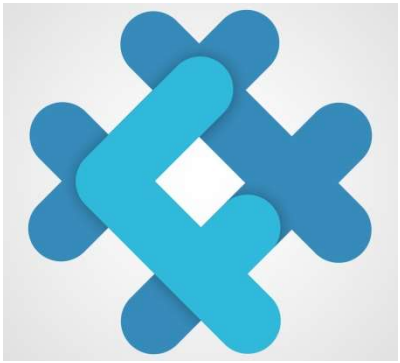
W jaki sposób?

Co chcę uzyskać?


Operacje wejściowe
(ze skutkami ubocznymi)
Rdzeń aplikacji
(czysty)
Operacje wyjściowe
(ze skutkami ubocznymi)

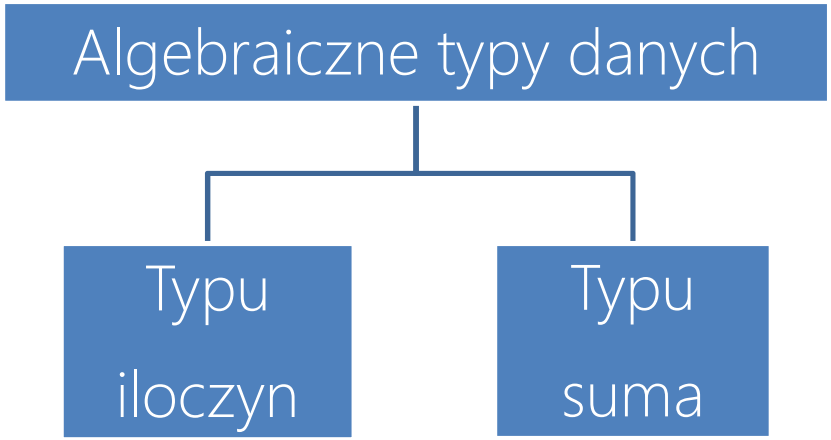
Programowa-
nie funkcyjne
Wartości
niemuto-
walne
Wyrażenia
Funkcje
czyste
Funkcje
wyższych
rzędów
Rekurencja
Ewaluacja
nierygory-
styczna
Dopaso-
wanie do
wzorca

Programowanie funkcyjne



TYPY ALGEBRAICZNE

Typy złożone



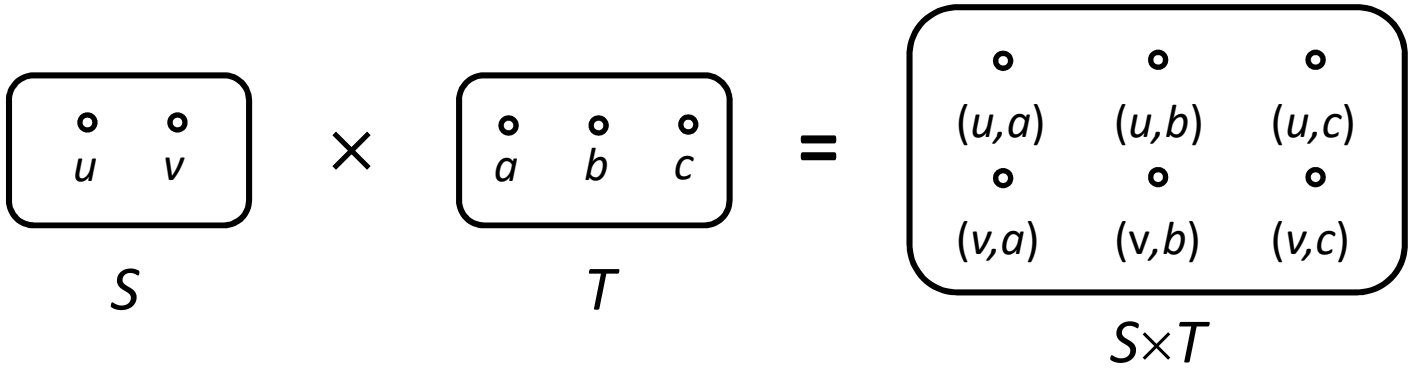
- Krotki (pary)
- Rekordy

- Unie z dyskryminatorem

Iloczyn kartezjański

W typach będących **produktem kartezjańskim** wartości kilku (być może różnych) typów są grupowane w n-tki.

$$S \times T = \{(x,y) \mid x \in S;\ y \in T\}$$



Krotki (n-tki)

Tworzone są jako zestaw wartości (tego samego lub różnych typów danych) podanych w nawiasie i wymienionych po przecinku

```
(12, 23.4, "Ala ma kota", true);;
```

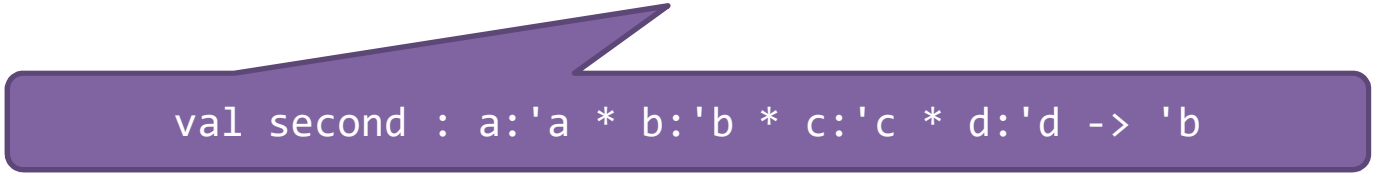
Definicja iloczynu kartezjańskiego następujących typów danych

```
int * float * string * bool
```

Krotki (n-tki)

Aby odzyskać wartość konieczne jest napisanie specjalnych funkcji np.

```
let second (a,b,c,d) = b;;
```




```
second (12, 23.4, true, "Ala ma kota");;
```

Krotki (n-tki)

Krotki mogą być również zwracane z funkcji:

```
let funkcja a b = (a+b, a-b)

let suma,roznica = funkcja 5 4
```

Funkcja jednego czy wielu argumentów?

Załóżmy, że chcemy stworzyć funkcję obliczającą pole prostokąta

```
let pole x1 y1 x2 y2 =
    (abs (x2-x1)) * (abs (y2-y1))

let pole (x1, y1) (x2, y2) =
    (abs (x2-x1)) * (abs (y2-y1))

let pole ((x1, y1), (x2, y2)) =
    (abs (x2-x1)) * (abs (y2-y1))
```

Funkcja jednego czy wielu argumentów?

```
let pole x1 y1 x2 y2 = (abs (x2-x1)) * (abs (y2-y1))
```

```
val pole: x1: int -> y1: int -> x2: int -> y2: int -> int
```

```
let pole (x1, y1) (x2, y2) = (abs (x2-x1)) * (abs (y2-y1))
```

```
val pole: x1: int * y1: int -> x2: int * y2: int -> int
```

```
let pole ((x1, y1), (x2, y2)) = (abs (x2-x1)) * (abs (y2-y1))
```

```
val pole: (int * int) * (int * int) -> int
```

Rekordy

Iloczyn kartezjański, w którym każda składowa ma swoją nazwę

```
type Osoba = {
    Imie : string;
    Nazwisko : string
}
```

Składowe rozdzielamy nową linią lub średnikiem

Rekordy

```
val osoba1 : Osoba = {Imie = "Tomek";
                      Nazwisko = "Kowalski";}
```

```
let osoba1 = {Imie = "Tomek"; Nazwisko = "Kowalski"}
let osoba2 = {Imie = "Tomek"; Nazwisko = "Kowalski"}
let osoba3 = {Imie = "Ewa"; Nazwisko = "Nowak"}
```

```
osoba1 = osoba2 //true
osoba1 = osoba3 //false
```

Rekordy

Rekordy są niemodyfikowalne

```
let osoba3 = {Imie = "Ewa"; Nazwisko = "Nowak"}

let osoba4 = {osoba2 with Nazwisko="Adamski"}
```

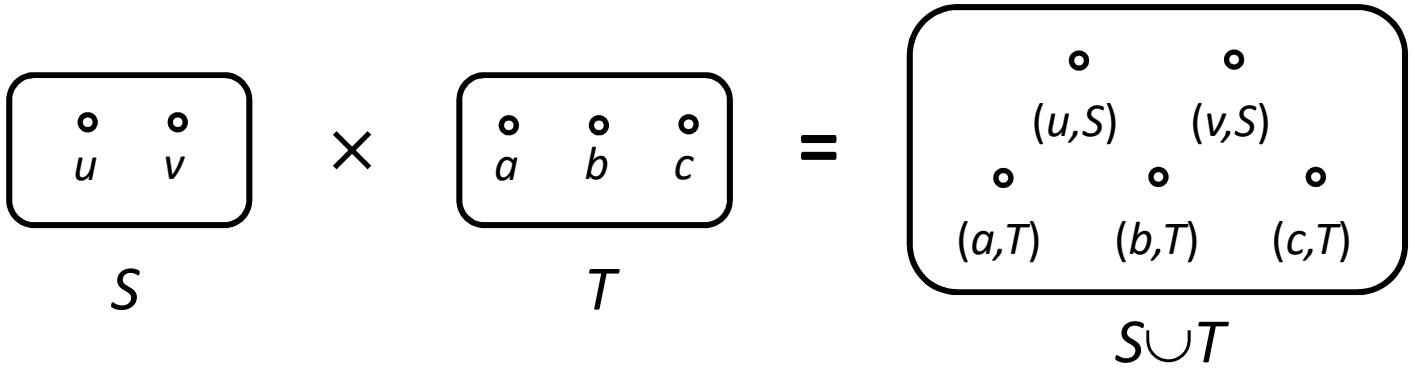
Rekordy

Do rekordów można dołączać metody

```
type Osoba = {
    Imie : string;
    Nazwisko : string
}

type Osoba with
   member this.ZNazwiskiem(nazwisko) =
       { this with Nazwisko = nazwisko }
```

Suma algebraiczna

Jest to typ złożony, który może przyjmować wartości kilku typów, ale w danym momencie może być wykorzystany tylko jeden typ

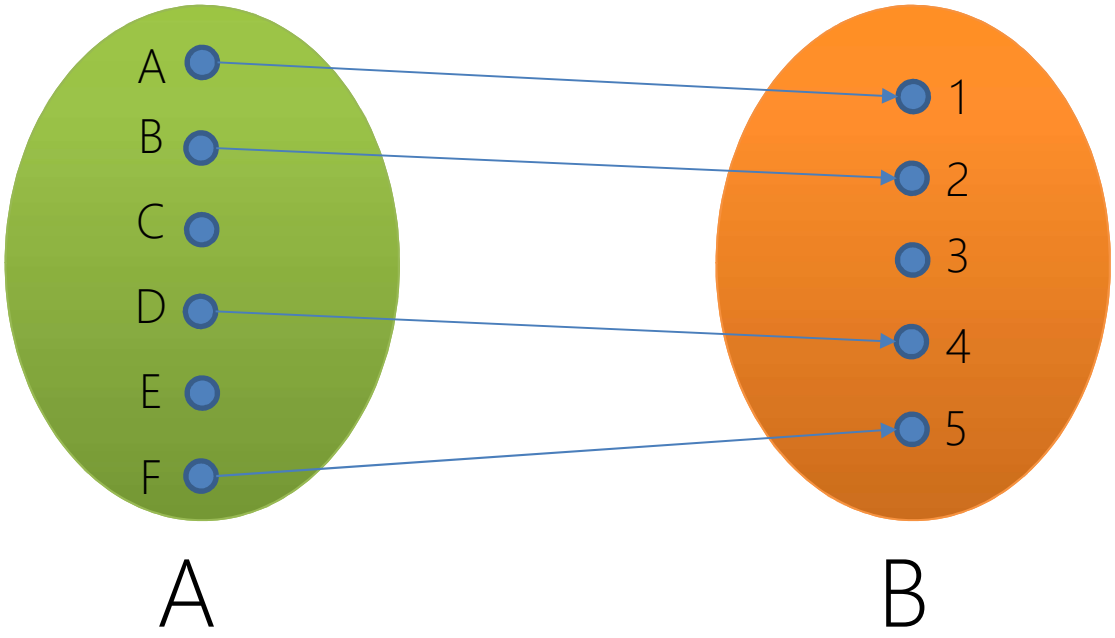


Unie z dyskryminatorem

```
type Znak =
   | Dodatni
   | Zero
   | Ujemny
```

```
let znak x =
   if x>0 then
      Dodatni
   elif x=0 then
      Zero
   else
      Ujemny
```

Funkcje częściowe



$$f\ x\ = \begin{cases} 1 & \text{gdy x=A} \\ 2 & \text{gdy x=B} \\ 4 & \text{gdy x=D} \\ 5 & \text{gdy x=F} \\ \perp & \text{w przeciwnym przypadku} \end{cases}$$

Funkcje częściowe

Załóżmy, że chcemy napisać funkcję, która ma określony rezultat tylko dla wartości dodatnich. Dla zera i wartości ujemnych jest nieokreślona

```
let funkcja x =
   if x>0.0 then
      x
   else
      ….
```

```
let funkcja x =
   if x>0.0 then
      x
   else
      failwith "nie ma wartości"
```

```
val funkcja: x: float -> float
```

Funkcje częściowe

Załóżmy, że chcemy napisać funkcję, która ma określony rezultat tylko dla wartości dodatnich. Dla zera i wartości ujemnych jest nieokreślona

```
type Wynik =
  | Wartosc of float
  | Blad of string
```

```
let funkcja x =
   if x>0.0 then
      Wartosc x
   else
      Blad "Nie ma wartości"
```

```
val funkcja: x: float -> Wynik
```

Funkcje częściowe

Załóżmy, że chcemy napisać funkcję, która ma określony rezultat tylko dla wartości dodatnich. Dla zera i wartości ujemnych jest nieokreślona

```
type Wynik<'a> =
  | Wartosc of 'a
  | Blad of string
```

```
let funkcja x =
   if x>0.0 then
      Wartosc x
   else
      Blad "Nie ma wartości"
```

```
val funkcja: x: float -> Wynik<float>
```

Instrukcja wyboru

C++:

```
switch(x) {
   case 1: instrukcje 1; break;
   case 2:
   case 3: instrukcje 2; break;
   default: inne instrukcje;
}
```

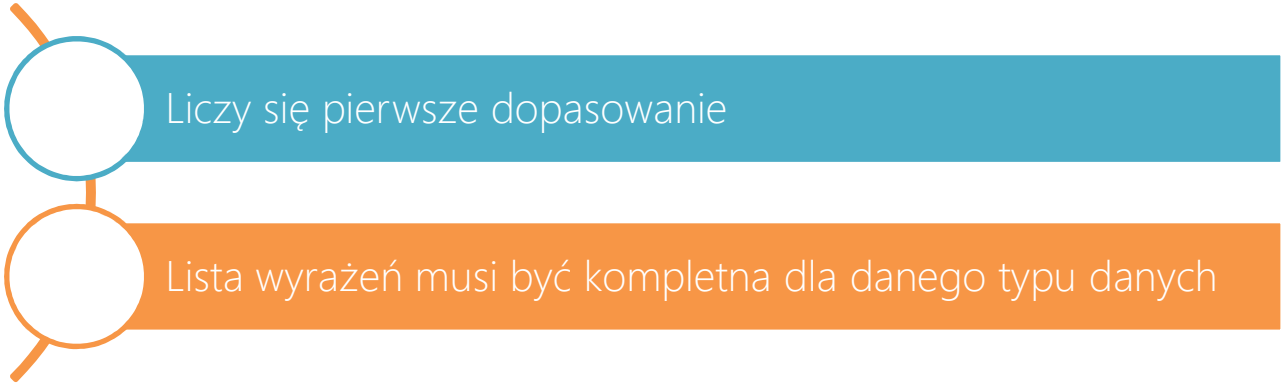
F#:

```
match x with
| 1 -> instrukcje1
| 2 | 3 -> instrukcje2
| _ -> instrukcje3
```

Dopasowywanie wzorców

Wyrażenie **match** pozwala dopasować wzorzec do wartości

```
match wynik with
| Wartosc x -> printfn "Wynik obliczeń %f" x
| Blad wiadomosc -> printfn "%s" wiadomość
```



Funkcje dopasowania wzorców

```
let naString wynik =
    match wynik with
    | Wartosc x -> sprintf "Wynik obliczeń %f" x
    | Blad wiadomosc -> sprintf "%s" wiadomosc
```

```
let naString = function
    | Wartosc x -> sprintf "Wynik obliczeń %f" x
    | Blad wiadomosc -> sprintf "%s" wiadomosc
```

Dopasowanie wzorców

Dopasowanie do rekordów

```
type Pojazd = {
   marka:string
   model:string
   rokProdukcji:int
}

match rekord with
   | {marka="Ford"} -> "To jest Ford"
   | {marka="Opel"; model=model } -> sprintf "To jest Opel %s" model
   | {rokProdukcji=rok} when rok=2021 -> "To jest auto z tego roku"
   | _ -> "Inne auto"
```

Dopasowanie wzorców

```
type Wynik =
  | Wartosc of float
  | Blad of string

let funkcja x =
   if x>0.0 then
      Wartosc x
   else
      Blad "Nie ma wartości"

let wynik = funkcja -1.0

match wynik with
| Wartosc x -> printfn "Wynik obliczen %f" x
| Blad wiadomosc -> printfn "%s" wiadomosc
```

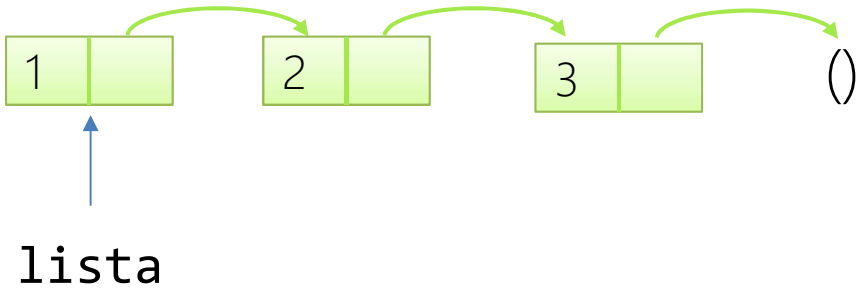
FUNKCYJNE STRUKTURY DANYCH

Listy

$$\text{Lista} = \begin{cases} \text{Pusta} \\ \underbrace{\text{Element}}_{\text{głowa}} :: \underbrace{\text{Lista}}_{\text{ogon}} \end{cases}$$

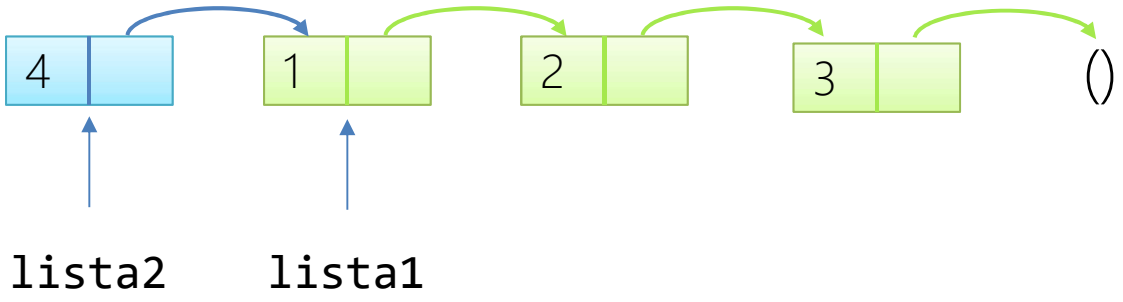
Lista

```
type Lista<'a> =
| Pusta
| Element of 'a*Lista<'a>
```



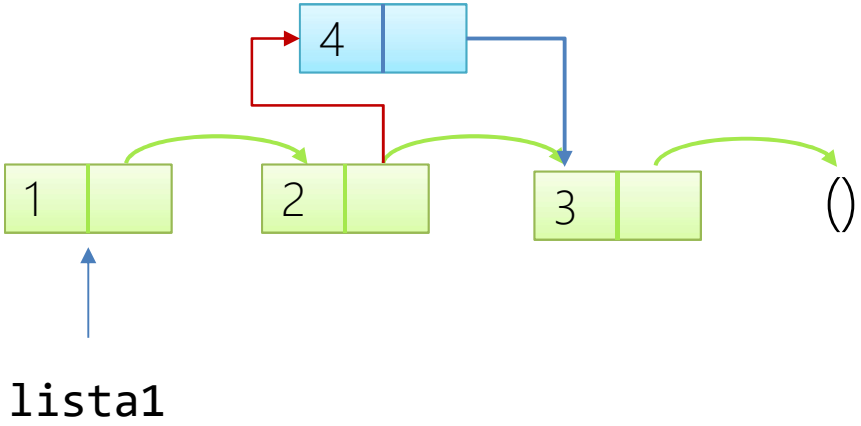

```
let lista = Element(1, Element(2, Element(3, Pusta)))
```

Listy – dodawanie elementu

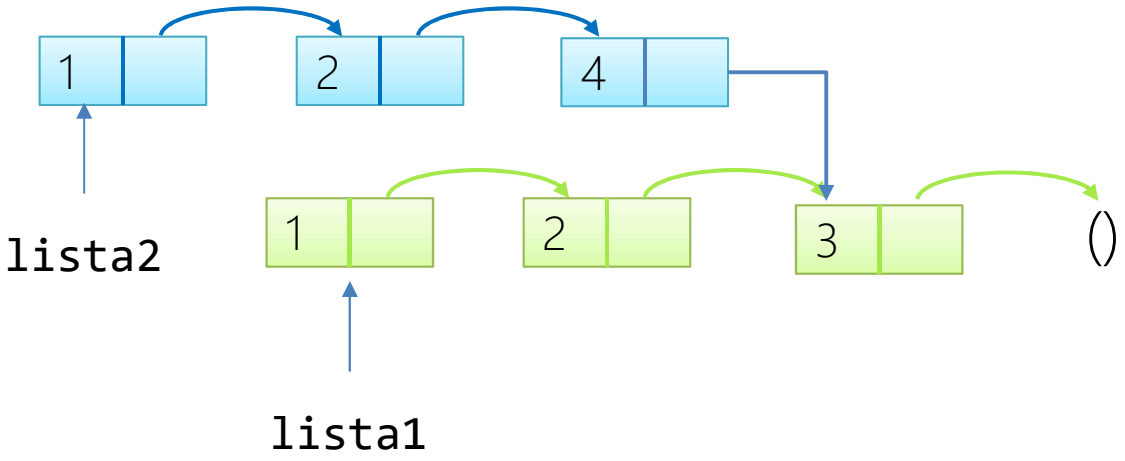



```
let lista1 = Element(1, Element(2, Element(3, Pusta)))

let lista2 = Element(4, lista)
```

Listy – dodawanie elementu



Listy – dodawanie elementu



Listy – dodawanie elementu

```
let rec dodajPo element nowyElement =
  function
  | Pusta -> failwith($"Nie znalazlem elementu {element}")
  | Element (glowa,ogon) ->
      if glowa = element then
        Element(element, Element (nowyElement, ogon))
      else
        Element (glowa, dodajPo element nowyElement ogon)
```

Listy – dodawanie elementu

```
let rec dodajPo element nowyElement =
  function
  | Pusta -> failwith($"Nie znalazlem elementu {element}")
  | Element (glowa,ogon) when glowa = element ->
      Element(element, Element (nowyElement, ogon))
  | Element (glowa,ogon) ->
      Element (glowa, dodajPo element nowyElement ogon)
```

Liczba elementów listy

```
let rec liczbaElementow =
  function
  | Pusta -> 0
  | Element (_,ogon) -> licz ogon + 1
```

Liczba elementów listy

```
let liczbaElementow lista =
  let rec liczbaElementow suma =
    function
    | Pusta -> suma
    | Element(glowa, ogon) -> liczbaElementow (suma+1) ogon
  liczbaElementow 0 lista
```

Listy – dodawanie elementu

```
let rec ogonPoElemencie element =
  function
  | Pusta -> failwith ($"Nie znalazlem elementu {element}")
  | Element (glowa,ogon) ->
      if glowa = element then
        ogon
      else
        ogonPoElemencie element ogon
```

Listy – dodawanie elementu

```
let kopiujDoElementu element lista =
  let rec kopiujDoElementu element nowaLista =
    function
    | Pusta -> nowaLista
    | Element (glowa, ogon) ->
        let nowyElement = Element(glowa, nowaLista)
        if glowa = element then
          nowyElement
        else
          kopiujDoElementu element nowyElement ogon
  kopiujDoElementu element Pusta lista
```

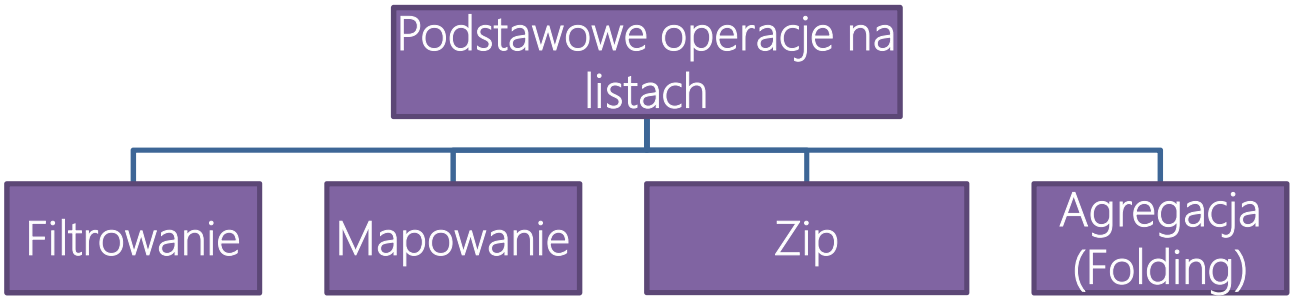
Listy – dodawanie elementu

```
let polaczListy lista1 lista2 =
  let rec polaczListy listaWynikowa =
    function
    | Pusta -> listaWynikowa
    | Element(glowa, ogon) ->
        polaczListy (Element (glowa, listaWynikowa)) ogon
  polaczListy lista2 lista1
```

Listy – dodawanie elementu

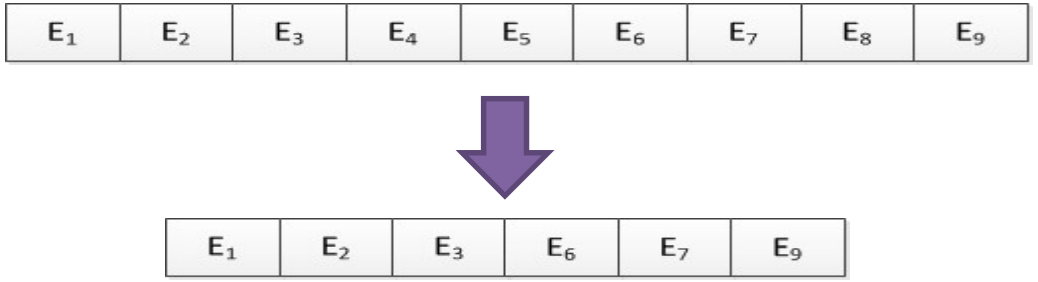
```
let dodajPo element nowyElement lista =
  let ogon = zwrocListePoElemencie element lista
  let poczatek = kopiujDoElementu element lista
  polaczListy  poczatek (Element (nowyElement, ogon))
```

Podstawowe operacje na listach



Filtrowanie

Tworzenie nowej listy na podstawie istniejącej, przy czym wybierane są tylko te elementy, które spełniają określony warunek.

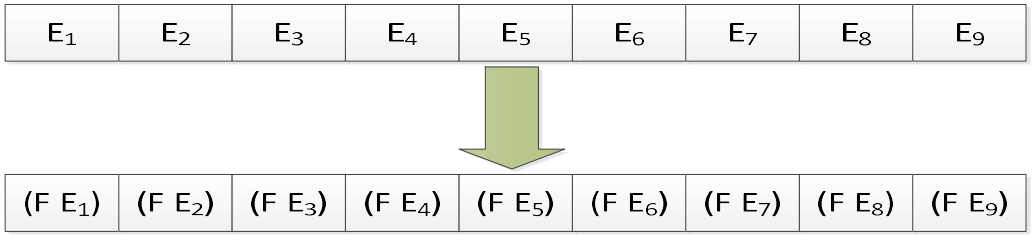


Filtrowanie

```
let rec tylkoParzyste = function
| Pusta -> Pusta
| Element (glowa,ogon) ->
    if glowa%2=0 then
      Element (glowa, tylkoParzyste ogon)
    else
      tylkoParzyste ogon
```

Mapowanie

Polega na utworzeniu nowej listy stosując do każdego elementu pewną funkcję

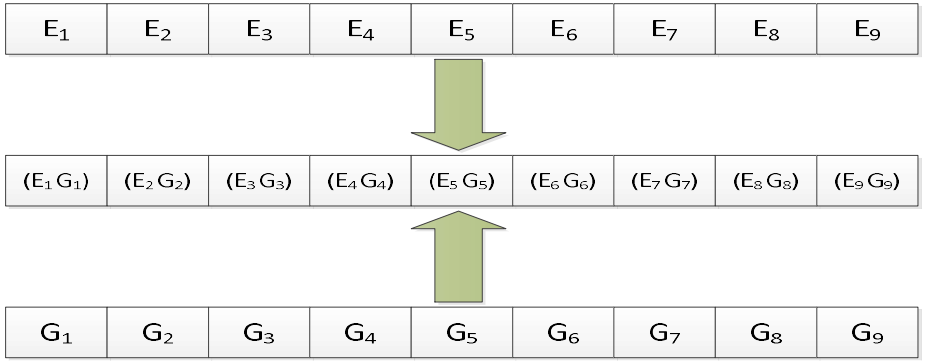


Mapowanie

```
let rec dwaRazyWieksza = function
| Pusta -> Pusta
| Element (glowa, ogon) -> Element (glowa*2, dwaRazyWieksza ogon)
```

zip

Polega na pobraniu dwóch list i stworzeniu trzeciej składającej się z par odpowiadających sobie elementów

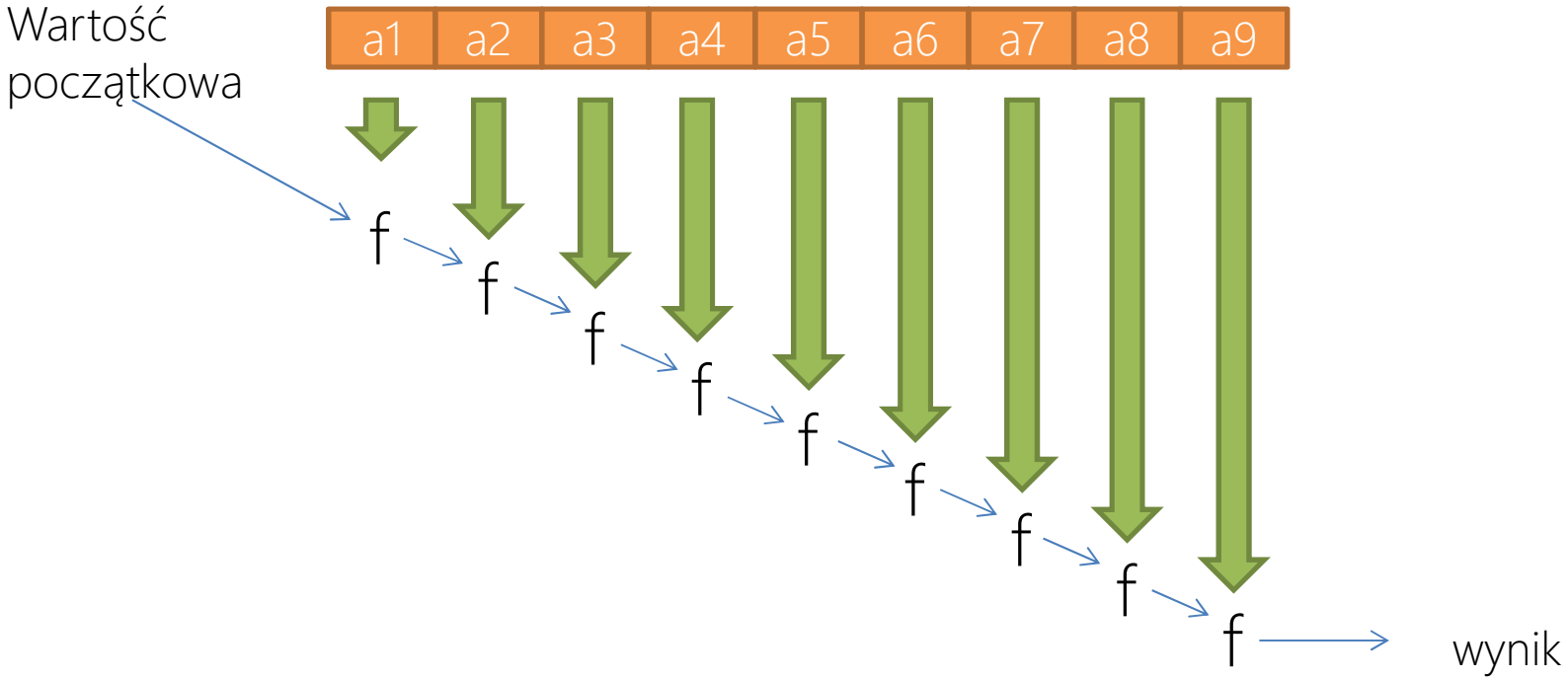


zip

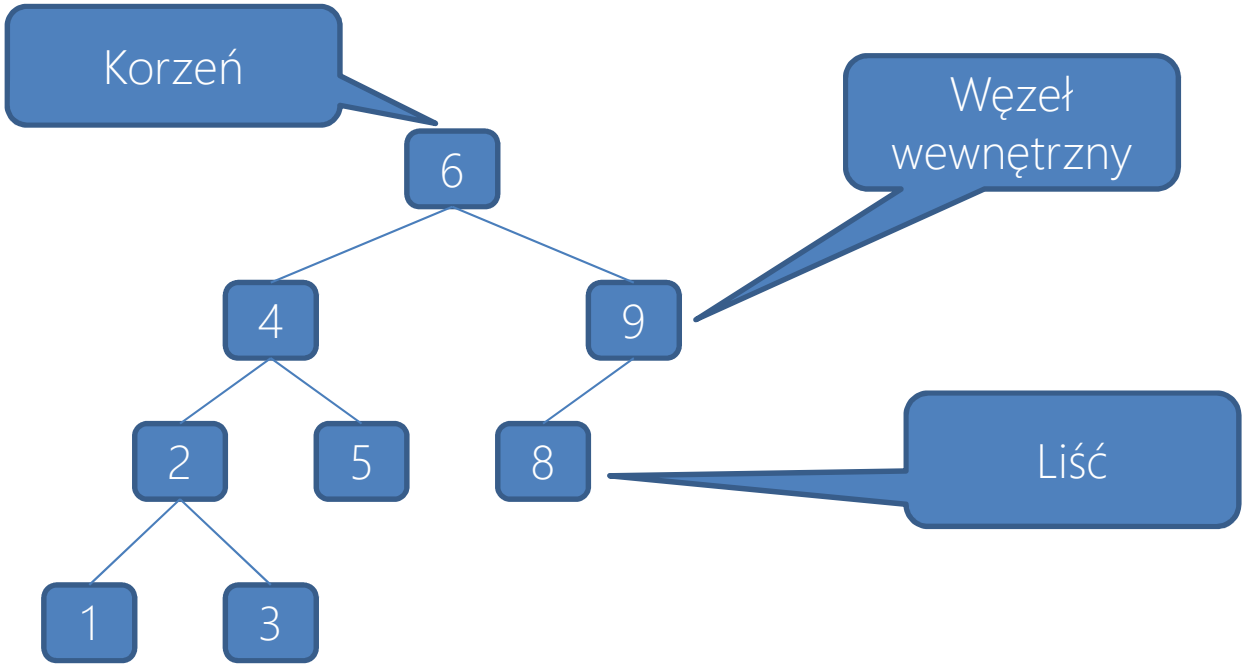
```
let rec zip lista1 lista2 =
   match (lista1, lista2) with
   | (Pusta, Pusta) -> Pusta
   | (Element (glowa1,ogon1), Element (glowa2,ogon2))
     -> Element ((glowa1,glowa2), zip ogon1 ogon2)
   | (Pusta, _) | (_, Pusta)
     -> failwith "Obie listy powinny mieć taką samą liczbę elementów"
```

Agregacja

Jest to operacja zamieniająca listę elementów na wartość skalarną

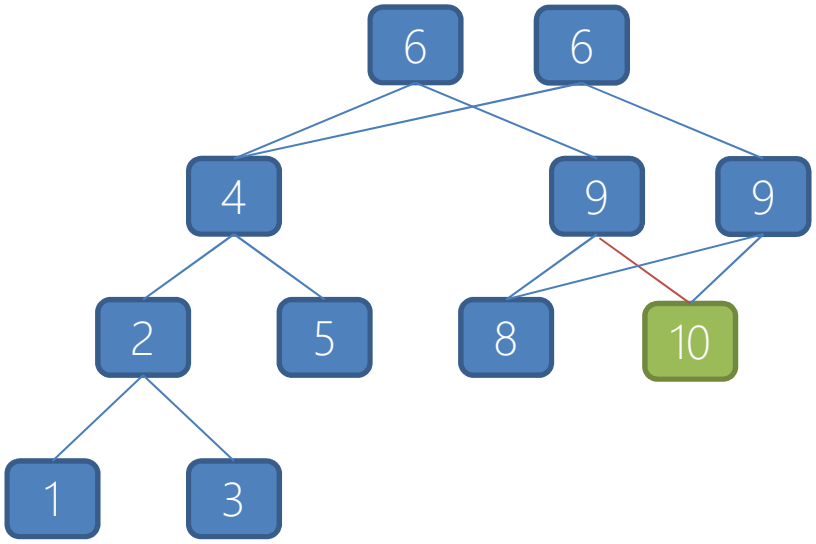


Struktury drzewiaste



$$Drzewo = \begin{cases} n \\ (n * Drzewo * Drzewo) \end{cases}$$

Struktury drzewiaste



Struktury drzewiaste

```
let rec wstaw liczba = function
  | Puste -> Wezel(liczba, Puste, Puste)
  | Wezel(x,l,p) when x<=liczba -> Wezel(x, l, wstaw liczba p)
  | Wezel(x,l,p) -> Wezel(x, wstaw liczba l, p)
```

Struktury drzewiaste

```
let rec wstaw liczba = function
  | Puste -> Wezel(liczba, Puste, Puste)
  | Wezel(x,l,p) when x<=liczba -> Wezel(x, l, wstaw liczba p)
  | Wezel(x,l,p) -> Wezel(x, wstaw liczba l, p)
```

Struktury drzewiaste

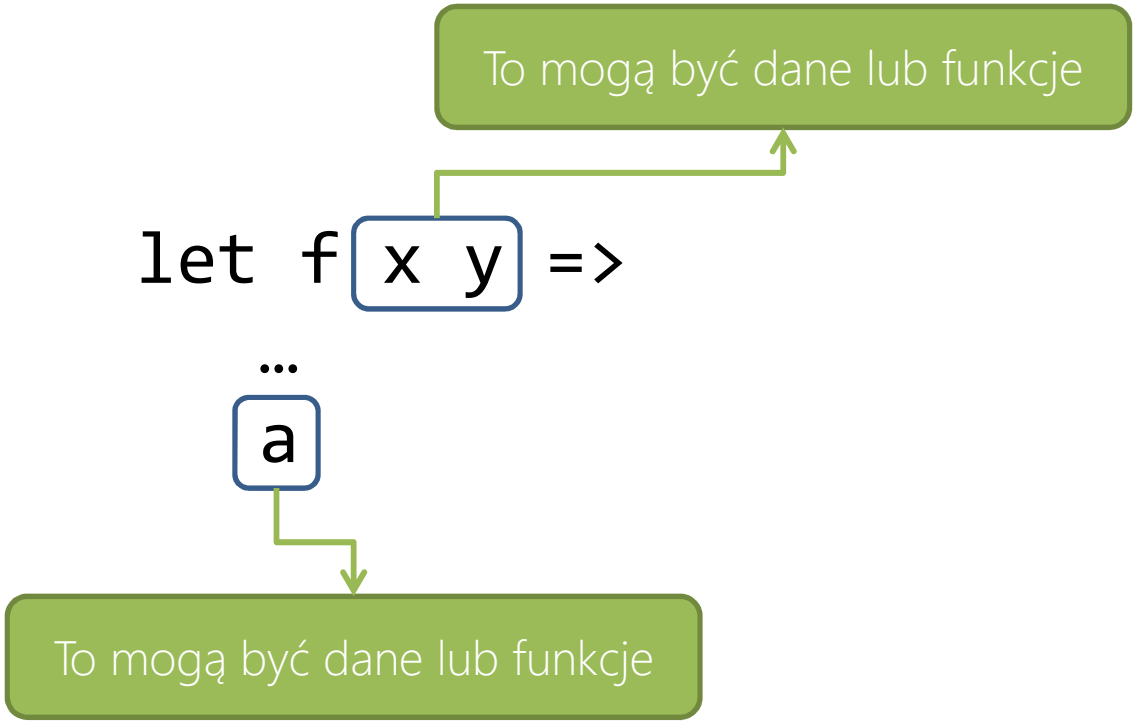
```
let wstaw liczba drzewo =

  let rec znajdzSciezke liczba sciezka = function
    | Puste -> Element(Puste, sciezka)
    | Wezel (x,_,p) as w when x<=liczba -> znajdzSciezke liczba (Wezel (w,sciezka)) p
    | Wezel (_,l,_) as w -> znajdzSciezke liczba (Wezel (w,sciezka)) l

  let rec zbuduj liczba drzewo = function
    | Pusta -> drzewo
    | Element (w,ogon) ->
        let nowyWezel =
          match w with
          | Puste -> Wezel(liczba, Puste, Puste)
          | Wezel(x,l,_) when x<=liczba-> Wezel(x, l, drzewo)
          | Wezel(x,_,p) -> Wezel(x, drzewo, p)
        zbuduj liczba nowyWezel ogon

  let sciezka = znajdzSciezke liczba Pusta drzewo
  zbuduj liczba Puste sciezka
```

Struktury drzewiaste

```
let rec liczbaWezlow = function
| Puste -> 0
| Wezel (_, l, p) -> 1 + liczbaWezlow l + liczbaWezlow p
```

FUNKCJE WYŻSZYCH RZĘDÓW

Funkcje wyższych rzędów



Przykład

```
let rec iloczyn = function
| Pusta -> 1.0
| Element(g,o) -> g*iloczyn o
```

```
let rec suma = function
| Pusta -> 0.0
| Element(g,o) -> g+suma o
```

```
let listaCalk = stworz 10

let il = iloczyn
            listaCalk;;

let su = suma
            listaCalk;;
```

Przykład

```
let rec iloczyn = function
| Pusta -> 1.0
| Element(g,o) -> g * iloczyn o
```

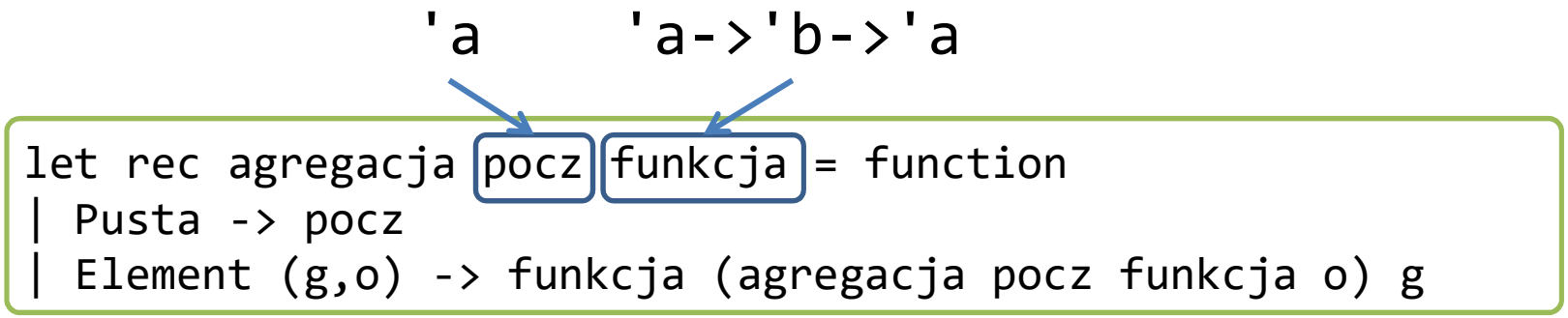
```
let rec suma = function
| Pusta -> 0.0
| Element(g,o) -> g + suma o
```

Wartość początkowa

Działanie

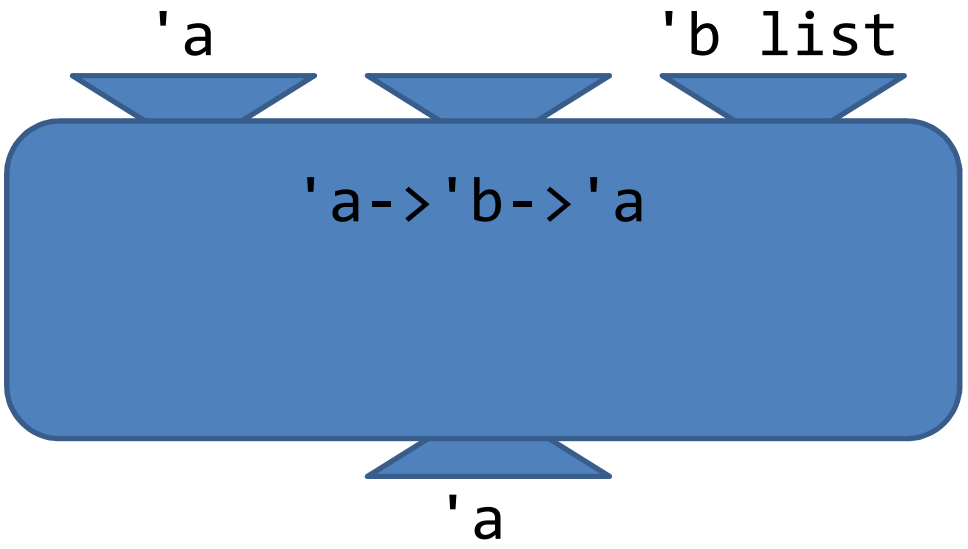
Jakie są elementy wspólne tych dwóch funkcji?

Przykład


'a
'a->'b->'a
let rec agregacja pocz funkcja = function
| Pusta -> pocz
| Element (g,o) -> funkcja (agregacja pocz funkcja o) g

Przykład

```
let rec agregacja pocz funkcja = function
| Pusta -> pocz
| Element (g,o) -> funkcja (agregacja pocz funkcja o) g
```



Przykład

```
let rec agregacja pocz funkcja = function
| Pusta -> pocz
| Element (g,o) -> funkcja (agregacja pocz funkcja o) g
```

```
let f1 a x = a+x;;
let f2 a x = a*x;;

agregacja 0 f1 lista;;
agregacja 1 f2 lista;;
```

Przykład

```
let rec agregacja pocz funkcja = function
| Pusta -> pocz
| Element (g,o) -> funkcja (agregacja pocz funkcja o) g
```

```
let f1 a x = a+x;;
let f2 a x = a*x;;

agregacja 0 f1 lista;;
agregacja 1 f2 lista;;
```

```
agregacja 0 (+) lista;;
agregacja 1 (*) lista;;
```

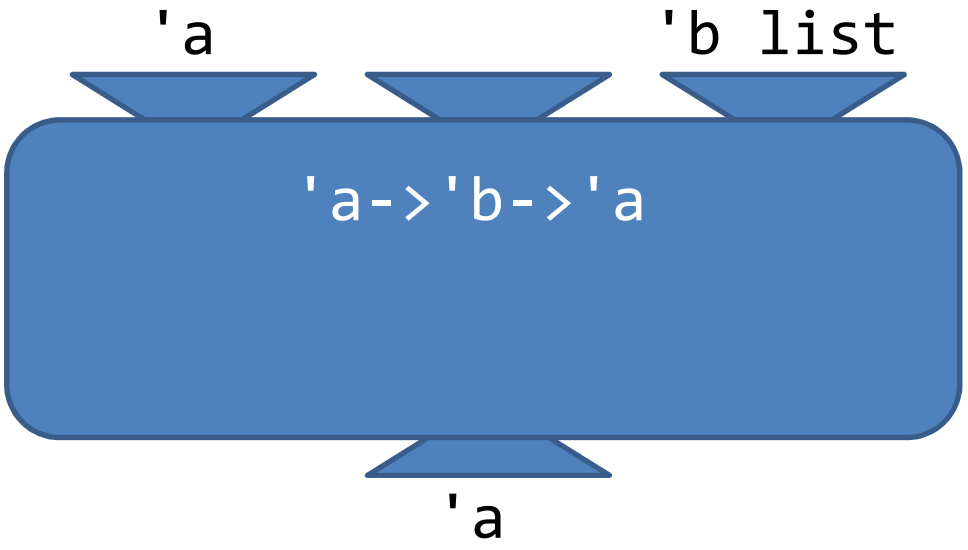
Przykład

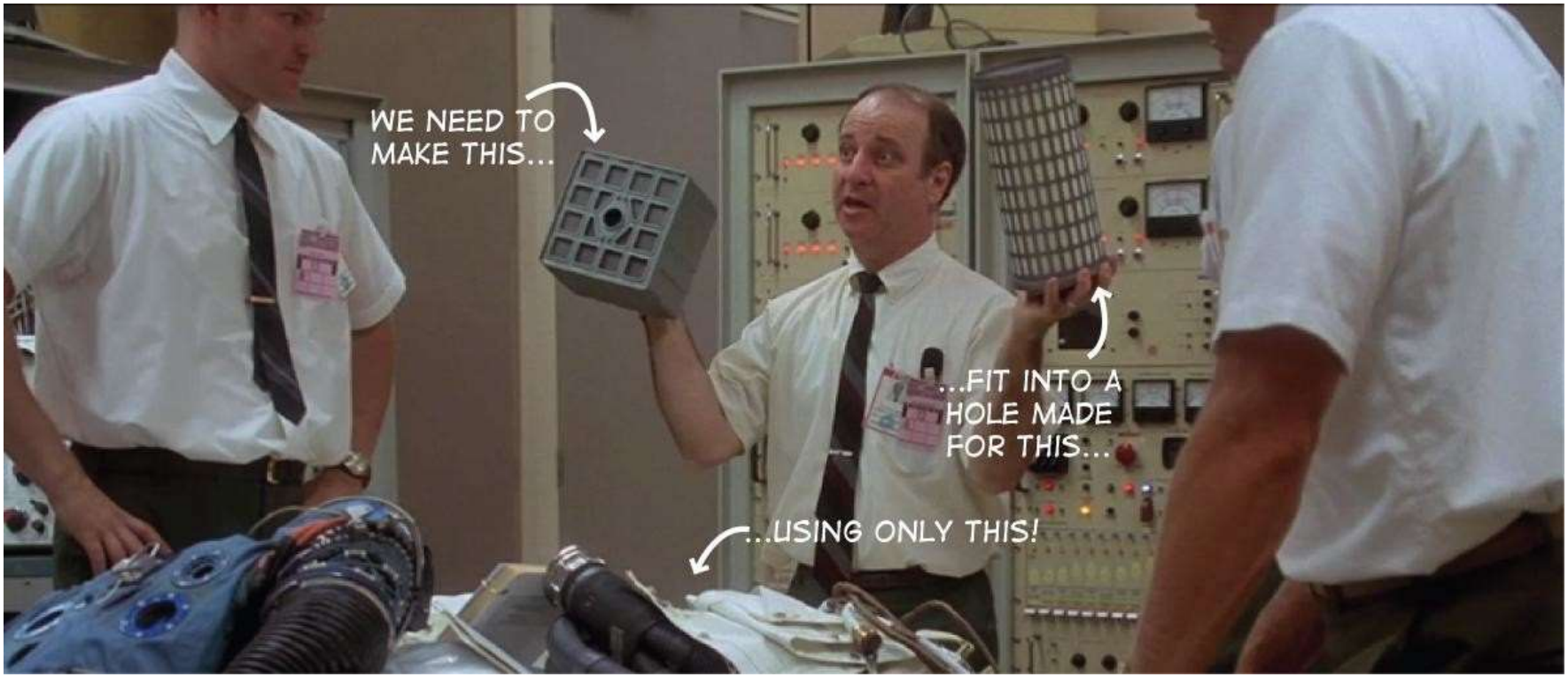
```
let rec agregacja pocz funkcja = function
| Pusta -> pocz
| Element (g,o) -> funkcja (agregacja pocz funkcja o) g
```

```
agregacja 0 (fun a x -> if x%2 = 0 then a+x else a) lista
agregacja 0 (fun a x -> if x%3 = 0 then a+x else a) lista
```

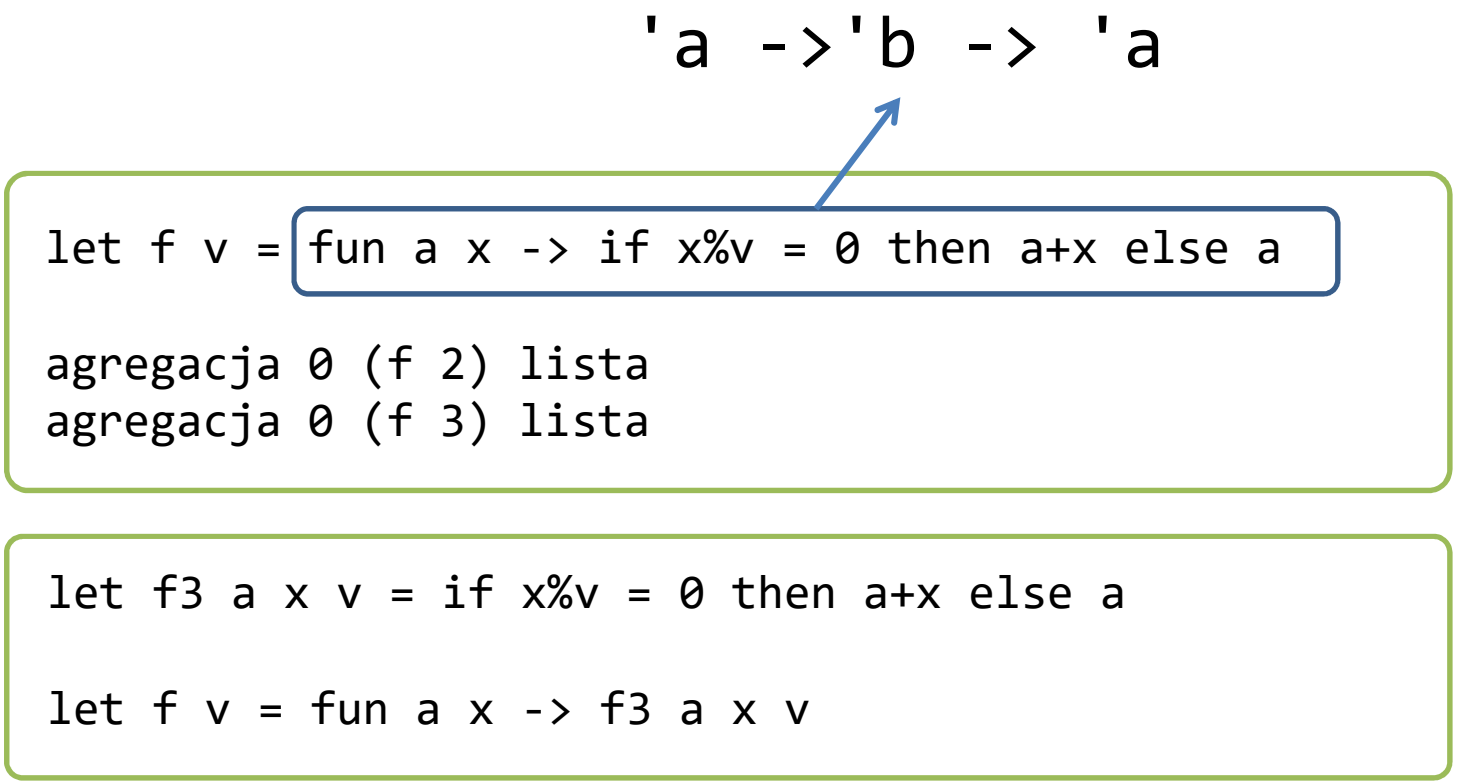
Przykład

```
let f1 a x = if x%2 = 0 then a+x else a
let f2 a x = if x%3 = 0 then a+x else a

let f3 a x v = if x%v = 0 then a+x else a
```

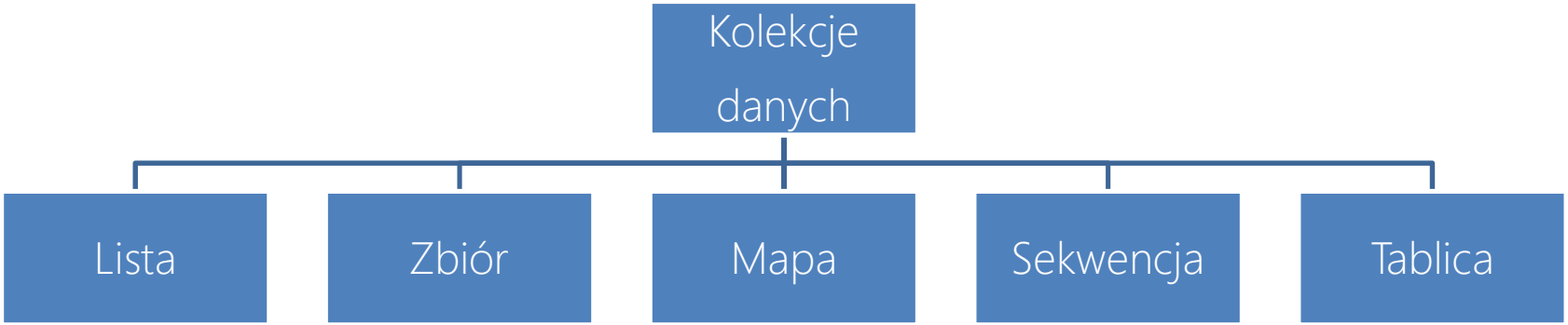



WE NEED TO MAKE THIS...
...FIT INTO A HOLE MADE FOR THIS...
...USING ONLY THIS!

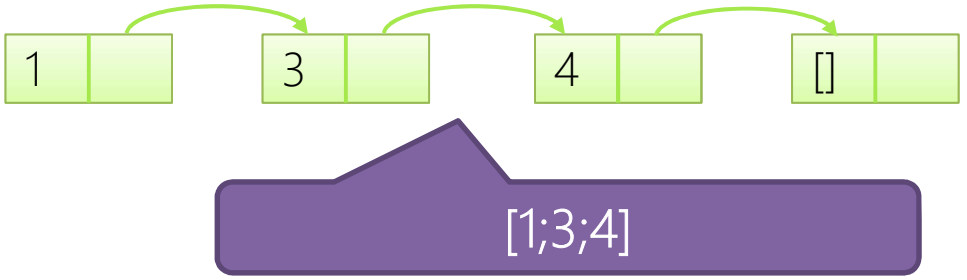
Przykład


'a ->'b -> 'a
let f v = fun a x -> if x%v = 0 then a+x else a
agregacja 0 (f 2) lista
agregacja 0 (f 3) lista
let f3 a x v = if x%v = 0 then a+x else a
let f v = fun a x -> f3 a x v

KOLEKCJE DANYCH W F#


Kolekcje danych
Lista
Zbiór
Mapa
Sekwencja
Tablica

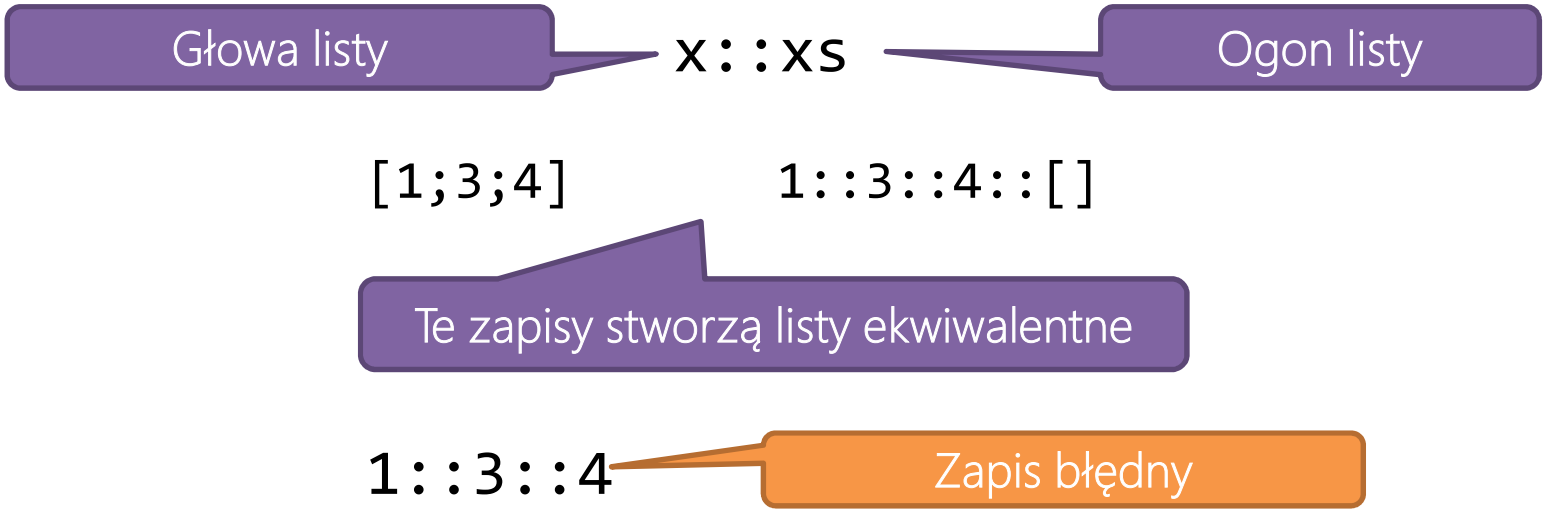
F# - Listy (1)



- Typ danych **T list**, gdzie **T** jest dowolnym typem danych języka F#
- Listy są tworzone za pomocą operatora **`[]`** oraz **`::`**
- **`[]`** oznacza pustą listę
- **`[1;3;4]`** – oznacza listę liczb całkowitych 1,3,4,[]

F# - Listy (2)

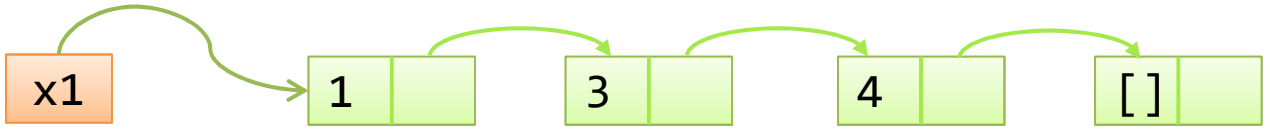
- Listy można również tworzyć za pomocą operatora `::`
- `x::xs` oznacza listę z pierwszym elementem `x` typu `T` a, a `xs` typu `T list`



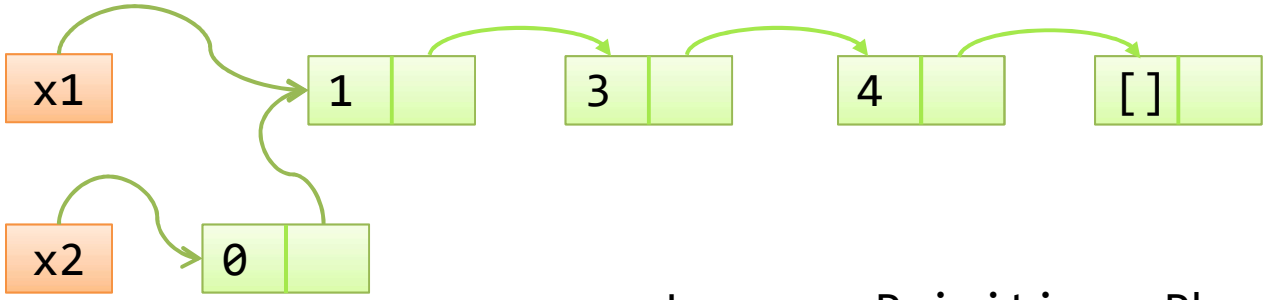
- Listy są niemodyfikowalne. Operator :: tworzy nową listę zamiast modyfikować istniejącą

F# - Listy (3)

```
let x1 = 1::3::4::[]
```




```
let x2 = 0::x1
```




```
LanguagePrimitives.PhysicalEquality
(List.tail x2) x1;;
```

F# - Listy (4)

Listy mogą być tworzone również za pomocą zakresów:

```
let x = [1..10];;
```

[1;2;3;4;5;6;7;8;9;10]

```
let x = [1..2..10];;
```

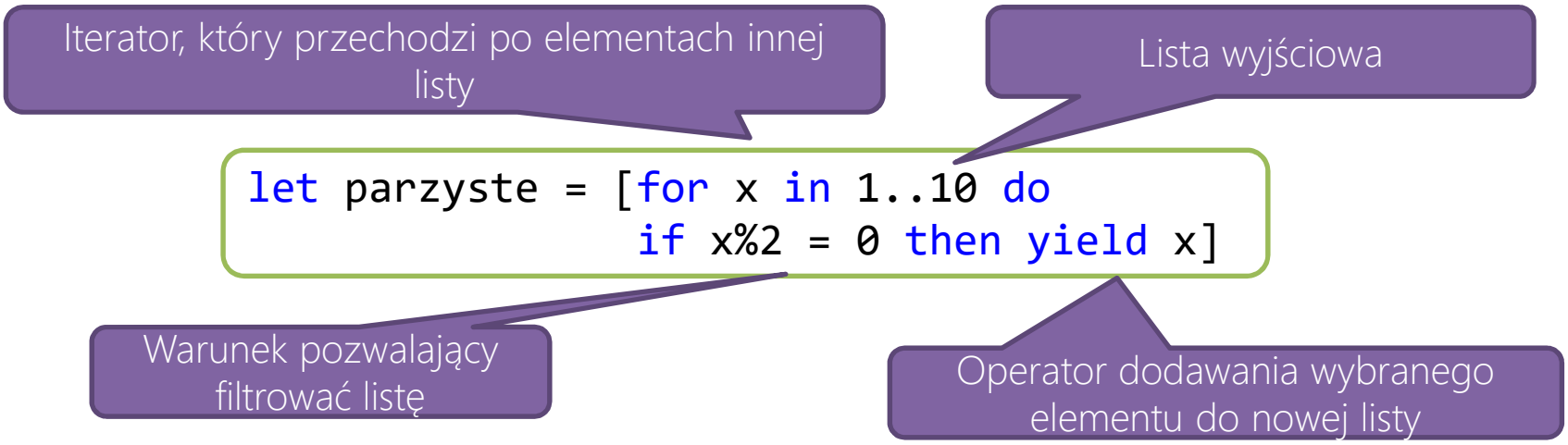
[1;3;5;7;9]

```
let x = ['a'..'e'];;
```

['a';'b';'c';'d';'e']

Skracanie list (1)

- Jest to mechanizm pozwalający na definiowanie list z użyciem istniejących list



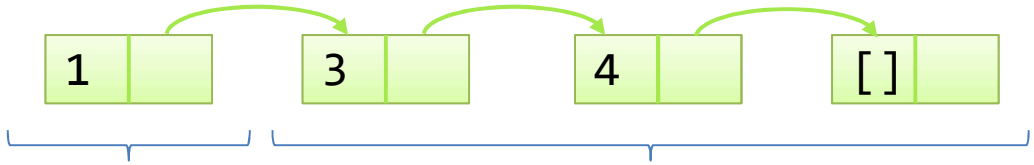
Skracanie list (2)

W ramach jednej operacji tworzenia list można wykorzystać wiele operatorów np..

```
let dzielniki n = [for x in 1 .. n do
                   if n%x = 0 then yield x]
```

```
let wspolneDzielniki m n = [for x in dzielniki m do
                            for y in dzielniki n do
                            if x=y then yield x]
```

Operacje działające na listach



Głowa listy
`'a`

Ogon listy
`'a list`

$$lista = \begin{cases} [] & gdy\ lista\ jest\ pusta \\ x :: xs & gdy\ lista\ ma\ głowę\ i\ ogon \end{cases}$$

Przykładowe funkcje operujące na listach (1)

```
let rec LiczbaElementow = function
  | [] -> 0
  | x::xs -> 1+LiczbaElementow xs
```

```
let rec DodajNaKoniec lista wartosc =
   match lista with
   | [] -> wartosc::[]
   | x::xs -> x::(DodajNaKoniec xs wartosc)
```

Przykładowe funkcje operujące na listach (2)

```
let rec PolaczListy xs ys =
    match xs with
      | [] -> ys
      | x::xss -> x::(PolaczListy xss ys)
```

```
PolaczListy [1;3;4] [2;3;4]
```

Łączenie list jest operacją tak popularną, że w F# istnieje do tego specjalny operator @

```
[1;3;4] @ [2;3;4]
```

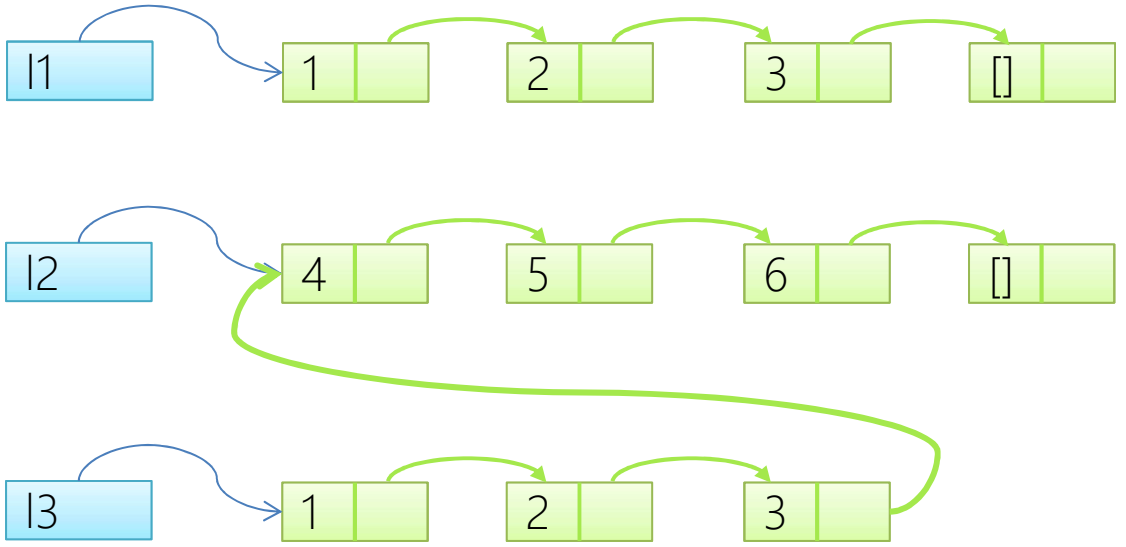
Przykładowe funkcje operujące na listach (3)

Uwaga na operator @! Jest wolny!

```
let l1 = [1;2;3];;
```

```
let l2 = [4;5;6];;
```

```
let l3 = l1@l2;;
```



Mapy

Niemodyfikowalny słownik, w którym elementy dostępne są poprzez klucz

```
let mapa = Map.ofList [("Ala", 3);("Tomek", 5);("Adam", 4)]

mapa.["Ala"]
```

Sekwencje

Sekwencja to leniwa kolekcja obiektów tego samego typu

```
let silnia n =
  let rec silnia n wynik =
    if n = 1I then
      wynik
    else
      silnia (n-1I) (n*wynik)
  silnia n 1I

let lista = [for n in 1I..20000I do yield (n, silnia n)]
List.iter (fun (n,v)->printfn "%A %A" n v) lista
```

Sekwencje

Sekwencja to leniwa kolekcja obiektów tego samego typu

```
let silnia n =
  let rec silnia n wynik =
    if n = 1I then
      wynik
    else
      silnia (n-1I) (n*wynik)
  silnia n 1I

let sekwencja = seq {for n in 1I..20000I do yield (n,silnia n)}
Seq.iter (fun (n,v)->printfn "%A %A" n v) sekwencja
```

Sekwencje

```
let dane = [
  {imie="Cezary"; nazwisko="Adamski"; wiek=20; rokStudiow=2};
  {imie="Franek"; nazwisko="Relke"; wiek=20; rokStudiow=2};
  {imie="Ala"; nazwisko="Kot"; wiek=19; rokStudiow=1};
  {imie="Ewa"; nazwisko="Nowacka"; wiek=22; rokStudiow=2};
  {imie="Adam"; nazwisko="Kowalski"; wiek=21; rokStudiow=3};
]

let dane  = List.filter (fun o->o.rokStudiow = 2) dane
let dane  = List.sortBy (fun o->o.nazwisko) dane
let wynik = List.map (fun o->{|imie=o.imie;
                               nazwisko=o.nazwisko|}) dane
List.iter (fun o->printfn "%A" o) wynik
```

Sekwencje

```
let dane = [
  {imie="Cezary"; nazwisko="Adamski"; wiek=20; rokStudiow=2};
  {imie="Franek"; nazwisko="Relke"; wiek=20; rokStudiow=2};
  {imie="Ala"; nazwisko="Kot"; wiek=19; rokStudiow=1};
  {imie="Ewa"; nazwisko="Nowacka"; wiek=22; rokStudiow=2};
  {imie="Adam"; nazwisko="Kowalski"; wiek=21; rokStudiow=3};
]

let dane  = Seq.filter (fun o->o.rokStudiow = 2) dane
let dane  = Seq.sortBy (fun o->o.nazwisko) dane
let dane  = Seq.map (fun o->{|imie=o.imie;
                              nazwisko=o.nazwisko|}) dane
let wynik = Seq.toList dane
List.iter (fun o->printfn "%A" o) wynik
```

WZORCE PROGRAMOWANIA FUNKCYJNEGO

Aplikacja funkcji

Aplikacja funkcji jest procesem wyznaczenia wartości funkcji dla określonych argumentów

```
let ink a = a + 1                    //ink 1 -> 2
```

```
ink: int -> int
```

```
let dodaj a b = a+b                  //dodaj 1 2 -> 3
```

```
dodaj: int -> int -> int
```

Aplikacja częściowa

```
let dodaj a b = a+b;;
```

```
let dodaj2 = dodaj 2;;
```

Powstaje nowa funkcja jednego parametru

```
let wynik = dodaj2 4;;
```

Aplikacja częściowa

Aplikacja częściowa jest to proces aplikacji funkcji tylko do części parametrów

```
let f3 a x v = if x%v = 0 then a+x else a

let g v = fun a x -> if x%v = 0 then a+x else a;;

List.fold  (g 3)  0 [1;2;3;4;5]
```

Ten argument musi mieć typ: ('a->'b->'a)

Aplikacja częściowa

Aplikacja częściowa jest to proces aplikacji funkcji tylko do części parametrów

```
let f3 v a x = if x%v = 0 then a+x else a

// int -> int -> int -> int

List.fold (f3 3) 0 [1;2;3;4;5]
```

Currying

Umożliwia traktowanie funkcji wielu parametrów jako funkcji jednego parametru

```
let dodaj a b = a+b;;
```

```
val dodaj : a:int -> b:int -> int
```

Operator pipe |>

1. Dana lista
2. Odfiltrować elementy mniejsze niż 3
3. Pozostałe podnieść do kwadratu
4. Z wyników zbudować kolejną listę

```
let res = List.map square
          (List.filter (fun x->x>3) [1;2;3;4;5;6;7] )
```

Operator pipe |>

```
let res = List.map square
          (List.filter (fun x->x>3) [1;2;3;4;5;6;7])
```

Operator pipe pozwala na odwrócenie kolejności:
najpierw piszemy dane, a później funkcję

```
let (|>) x f = f x
```

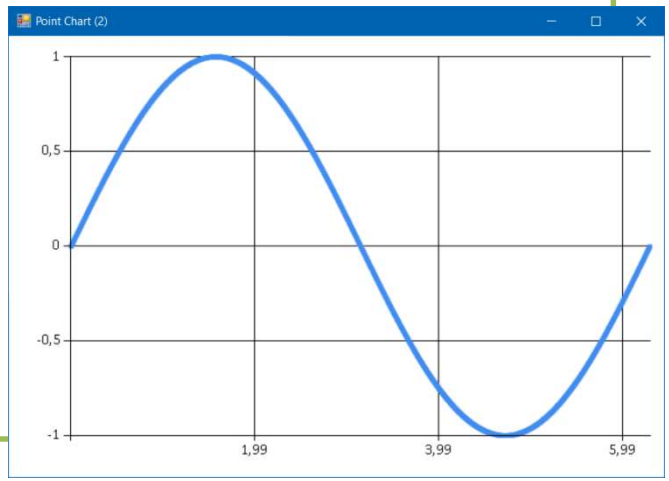
Operator pipe |>

```
let res = List.map square
          (List.filter (fun x->x>3) [1;2;3;4;5;6;7])
```

```
let res = [1;2;3;4;5;6;7]
          |> List.filter (fun x->x>3)
          |> List.map square
```

Operator pipe |>

```
#load
"../packages/FSharp.Charting.0.90.14/FSharp.Charting.fsx"

open System
open FSharp.Charting



[0.0..0.01..2.0*Math.PI]
  |> List.map (fun x -> (x, sin x))
  |> Chart.Point
```



Operator pipe |>

```
let rec agregacja pocz funkcja = function
| Pusta -> pocz
| Element (g,o) -> funkcja (agregacja pocz funkcja o) g
```

```
lista |> agregacja 0 (podzielnePrzez 3)
```

Inne pipe

```
(1,2)  ||>  min
```

```
(1,2,3) |||> (fun a b c -> a+b+c)
```

```
(fun a -> a+1)  <| 1
```

```
min  <| (1,2)
```

```
(fun a b c -> a+b+c)  <| (1,2,3)
```

Złożenie funkcji

```
let res = [1;2;3;4;5;6;7]
          |>List.filter (fun x->x>3)
          |>List.map square
```

```
let f lista = lista
              |> List.filter (fun x->x>3)
              |> List.map square
```

Złożenie funkcji

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

```
let square x = x*x
let double x = x+x
```

```
let doubleSquare = square >> double
```

=

```
let doubleSquare x = double (square x)
```

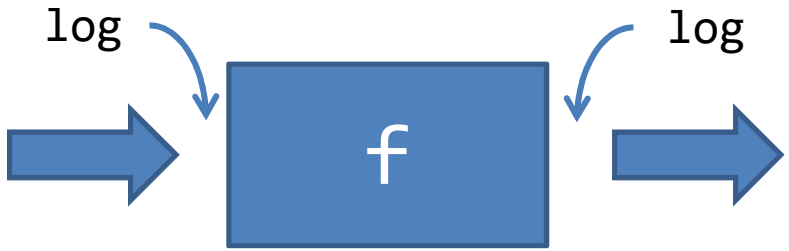
Złożenie funkcji

```
let f lista = lista
              |>  List.filter (fun x->x>3)
              |>  List.map square
```

```
let f = List.filter (fun x->x>3)
        >> List.map square
```

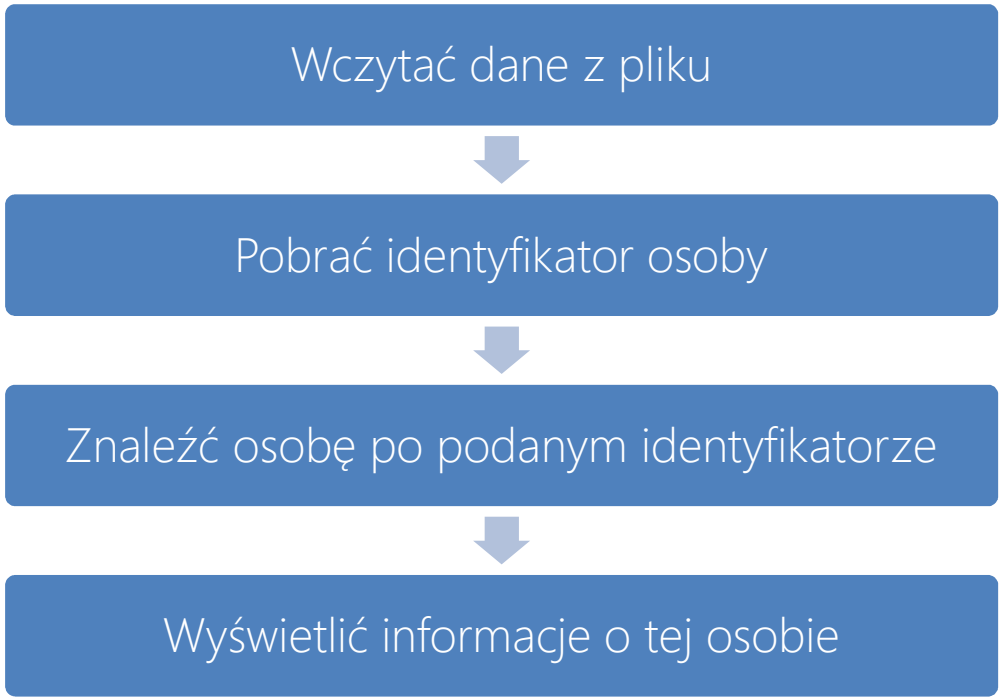
Złożenie funkcji

```
let f = List.filter (fun x->x>3)
        >> List.map square
```




```
let log x = printfn "%A" x
            x
let fWithLog = log>>f>>log
```

Przykładowa aplikacja



Przykładowa aplikacja

Dane dla naszego przykładu

```
[<AllowNullLiteral>]
type Osoba(id:int, imie:string, nazwisko:string) =
    member this.id = id
    member this.imie = imie
    member this.nazwisko = nazwisko
```

Przykładowa aplikacja

```
let wczytajDane nazwaPliku =
    File.ReadAllText nazwaPliku
    |> JsonConvert.DeserializeObject<Osoba list>

let wyswietlDane (osoba:Osoba) =
    printfn "%d" osoba.id
    printfn "%s" osoba.imie
    printfn "%s" osoba.nazwisko

let znajdzOsobePoId (lista:Osoba list) id =
    let poId (osoba:Osoba) = osoba.id = id
    if lista |> Seq.exists poId then
        lista |> List.find poId
    else
        null
```

Przykładowa aplikacja

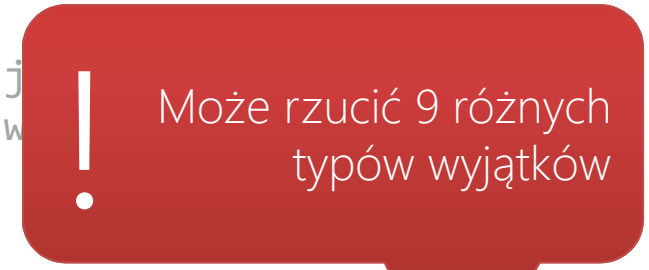
```
[<EntryPoint>]
let main argv =
    let dane = wczytajDane "dane.json"
    printf "Podaj id osoby: "
    Console.ReadLine ()
    |> int
    |> znajdzOsobePoId dane
    |> wyswietlDane
```

Co w tej aplikacji może pójść źle?

```
[<EntryPoint>]
let main argv =
    let dane = wczytajDane "dane.json"
    printfn "Podaj id osoby: "
    Console.ReadLine ()
    |> int
    |> znajdzOsobePoId dane
    |> wyswietlDane
```

Co w tej aplikacji może pójść źle?

```
let main argv =
    let dane = wczytajDane "dane.json"
    printf "Podaj id osoby: "
    Console.ReadLine ()
    |> int
    |> znaj
    |> wysw
```



```
let wczytajDane nazwaPliku =
    File.ReadAllText nazwaPliku
    |> JsonConvert.DeserializeObject<Osoba list>
```

Co w tej aplikacji może pójść źle?

```
let main argv =
    let dane = wczytajDane "dane.json"
    printf "Podaj id osoby: "
    Console.ReadLine ()
    |> int
    |> znajdzOsobePoId dane
    |> wyswietlDane
```

Mogą być rzucone wyjątki, ale są mało prawdopodobne

Co w tej aplikacji może pójść źle?

```
let main argv =
    let dane = wczytajDane "dane.json"
    printf "Podaj id osoby:"
    Console.ReadLine ()
    |> int
    |> znajdzOsobePoId dane
    |> wyswietlDane
```

String może być w złym formacie

Co w tej aplikacji może pójść źle?

```
let main argv =
    let dane = wczytajDane "dane.json"
    printf "Podaj id osoby:"
    Console.ReadLine ()
    |> int
    |> znajdzOsobePoId dane
    |> wyswietlDane
```

Ten fragment jest bezpieczny

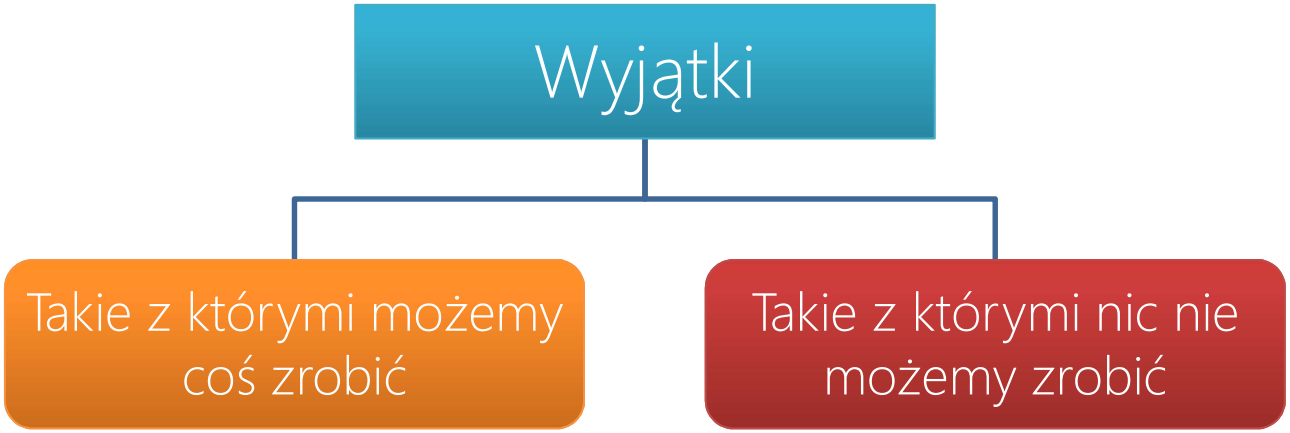
Co w tej aplikacji może pójść źle?

```
let main argv =
    let dane = wczytajDane "dane.json"
    printf "Podaj id osoby:"
    Console.ReadLine ()
    |> int
    |> znajdzOsobePoId dane
    |> wyswietlDane
```

```
let wyswietlDane (osoba:Osoba) =
    printfn "%d" osoba.id
    printfn "%s" osoba.imie
    printfn "%s" osoba.nazwisko
```

Jeżeli osoba będzie null to będzie wyjątek

Obsługa wyjątków



Możemy je obsłużyć
i naprawić błąd

Nie możemy ich naprawić
i zamykamy aplikację

Naprawa aplikacji

```
[<EntryPoint>]
let main argv =
    try
        let dane  = wczytajDane "dane.json"
        let id    = wczytajId ()
        let osoba = znajdzOsobePoId dane id
        if  osoba <> null then
            wyswietlDane osoba
        else
            pokazBlad "Nie ma osoby o takim id"
    with
    | _ -> pokazBlad "Tego błędu już nie naprawię"
    0
```

Pomyłka za miliardy dolarów

Tony Hoare

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Naprawa aplikacji

```
let rec wczytajId () =
    try
        printf "Podaj id osoby: "
        Console.ReadLine () |> int
    with
        | _ -> Console.Clear();
               pokazBlad "Błędny identyfikator użytkownika"
               wczytajId ()
```

Naprawa aplikacji

```
let pokazBlad (wiadomosc:string) =
    Console.ForegroundColor <- ConsoleColor.Red
    printfn "%s" wiadomosc
    Console.ResetColor();
```

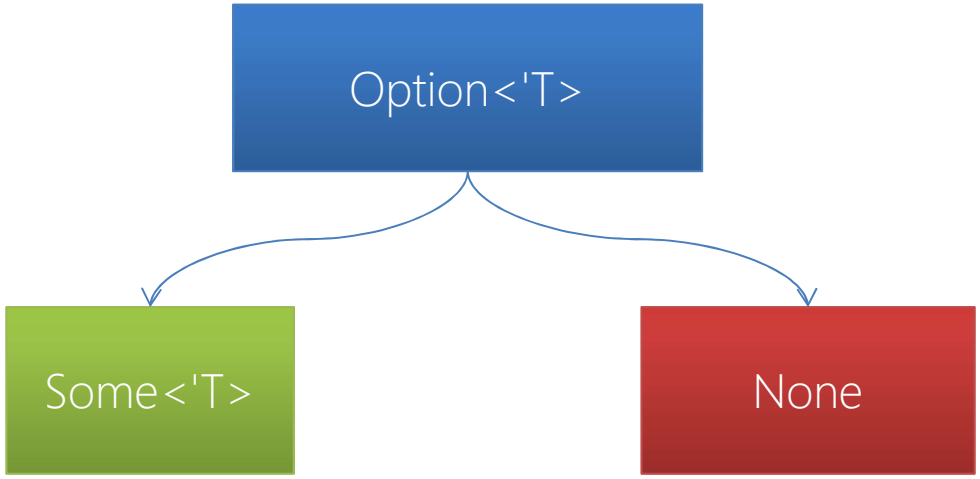
```
let rec wczytajId () =
    try
        printf "Podaj id osoby: "
        Console.ReadLine () |> int
    with
        | _ -> Console.Clear();
               pokazBlad "Błędny identyfikator użytkownika"
               wczytajId ()


[<EntryPoint>]
let main argv =
    try
        let dane  = wczytajDane "dane.json"
        let id    = wczytajId ()
        let osoba = znajdzOsobePoId dane id
        if osoba <> null then
            wyswietlDane osoba
        else
            pokazBlad "Nie ma osoby o takim id"
    with
    | _ -> pokazBlad "Tego błędu już nie naprawię"
    0
```

Czy można to napisać lepiej?

BO JAK NIE NULL TO CO?

Opcje

Typ Option pozwala na określanie czy dana wartość istnieje czy nie




```
type Option<'T> =
| Some of 'T
| None
```

```
let znajdzOsobePoId (lista:Osoba list) id =
    let poId (osoba:Osoba) = osoba.id = id
    if lista |> Seq.exists poId then
        lista |> List.find poId
    else
        null
```

Osoba list->int->Osoba

```
let znajdzOsobePoId (lista:Osoba list) id =
    let poId (osoba:Osoba) = osoba.id = id

    if lista |> Seq.exists poId then
       Some (lista |> List.find poId)
    else
       None
```

Osoba list->int->Option<Osoba>

```
let dane  = wczytajDane "dane.json"
let id    = wczytajId ()
let osoba = znajdzOsobePoId dane id
if osoba <> null then
   wyswietlDane osoba
else
   pokazBlad "Nie ma osoby o takim id"
```

```
let dane = wczytajDane "dane.json"
let id    = wczytajId ()
let osoba = znajdzOsobePoId dane id
match osoba with
| Some osoba -> wyswietlDane osoba
| None -> pokazBlad "Nie ma osoby o takim id"
```

```fsharp
let pokazWynik = function
    | Some osoba -> wyswietlDane osoba
    | None -> pokazBlad "Nie ma osoby o takim id"
```

```fsharp
let dane  = wczytajDane "dane.json"
let id    = wczytajId ()
let osoba = znajdzOsobePoId dane id
pokazWynik osoba
```

```fsharp
let dane  = wczytajDane "dane.json"
wczytajId ()
|> znajdzOsobePoId dane
|> pokazWynik
```

funkcja main

```
[<EntryPoint>]
let main argv =
    try
        let dane  = wczytajDane "dane.json"
        wczytajId ()
        |> znajdzOsobePoId dane
        |> pokazWynik
    with
    | _ -> pokazBlad "Tego błędu już nie naprawię"
    0
```

funkcja wczytajId

```
let rec wczytajId () =
    try
        printfn "Podaj id osoby: "
        Console.ReadLine () |> int
    with
        | _ -> Console.Clear();
               pokazBlad "Błędny identyfikator użytkownika"
               wczytajId ()
```

funkcja wczytajId

```
let rec wczytajId () =
    try
        printfn "Podaj id osoby: "
        Console.ReadLine () |> int
    with
        | _ -> Console.Clear();
               pokazBlad "…"
               wczytajId ()
```

```
let wczytajId () =
    printfn "Podaj id osoby: "
    Console.ReadLine () |> parse
```

```
let parse str =
    try
        Some (int str)
    with
        | _ -> None
```

Opcje w tworzeniu pętli

```
let rec loop f =
    match f () with
    | Some x -> x
    | None -> loop f
```

```
[<EntryPoint>]
let main argv =
  let dane  = wczytajDane "dane.json"

  loop wczytajId

  |> znajdzOsobePoId2 dane
  |> pokazWynik
  0
```

Funkcja wczytajDane

```
let wczytajDane nazwaPliku =
    try
        File.ReadAllText nazwaPliku
        |> JsonConvert.DeserializeObject<Osoba list>
        |> Some
    with
    | _ ->  pokazBlad ("Tego błędu to już nie naprawię")
            None
```

```
[<EntryPoint>]
let main argv =
    let dane  = wczytajDane "dane.json"
    loop wczytajId
    |> znajdzOsobePoId dane
    |> pokazWynik
0
```

string → Osoba list option

Nie pasuje

Osoba list → int -> Osoba option

```
[<EntryPoint>]
let main argv =
    let dane  = wczytajDane "dane.json"
    match dane with
    | Some osoby ->
          loop wczytajId
          |> znajdzOsobePoId osoby
          |> pokazWynik
    | None -> ()
0
```

Komendy/akcje i wyrażenia

Komendy / akcje

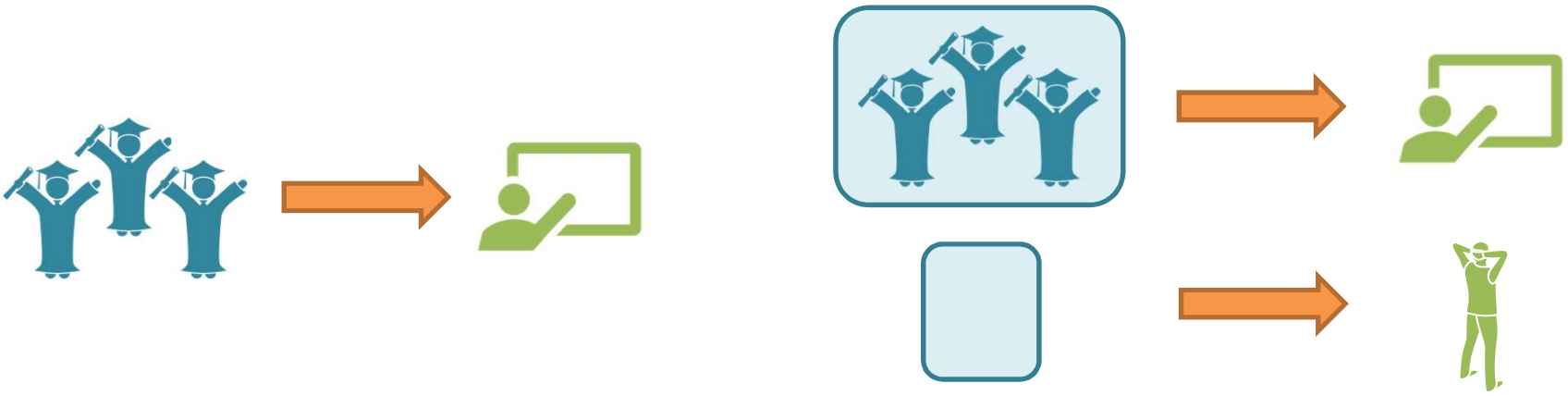
- Modyfikują stan programu
- Ich działanie można obserwować tylko poprzez analizę stanu

Wyrażenia

- Tworzą wartości
- Ich rezultat można obserwować poprzez analizę wartości zwracanej

Funkcja wykonaj

```
let wykonaj akcja opcja =
    match opcja with
    | Some x -> akcja x
    | None -> ()
```

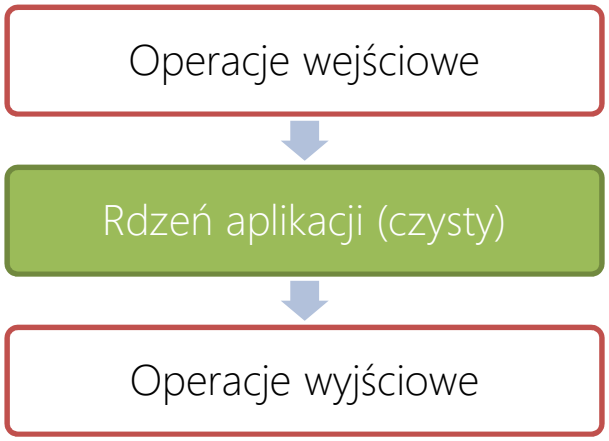


Funkcja wykonaj

```
[<EntryPoint>]
let main argv =
    let dane  = wczytajDane "dane.json"
    wykonaj (
      fun dane->
            loop wczytajId
            |> znajdzOsobePoId dane
            |> pokazWynik
    ) dane
    0
```

Funkcja main

```
[<EntryPoint>]
let main argv =
    wczytajDane "dane.json"
    |> wykonaj (fun dane->
                    loop wczytajId
                    |> znajdzOsobePoId dane
                    |> pokazWynik
        )
    0
```



To co byśmy chcieli uzyskać

```
[<EntryPoint>]
let main argv =
    wczytajDane "dane.json"
    |> … (fun dane->
                    loop wczytajId
                    |> znajdzOsobePoId dane
                    |> …
        )
    |> printfn "%A"
    0
```

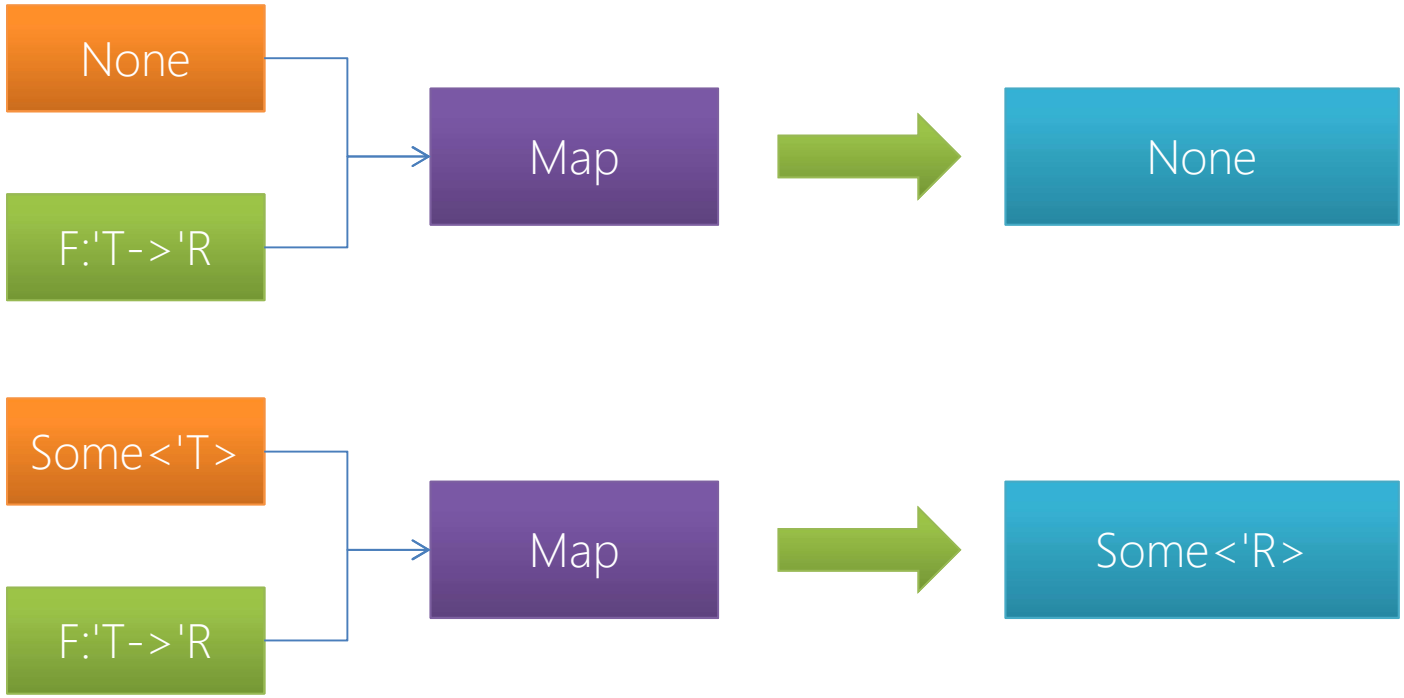
MAPOWANIE, FUNKTORY I MONADY

Mapowanie

$$\text{mapuj}: M <' T > \rightarrow ('T \rightarrow 'R) \rightarrow M < 'R >$$

$$\text{mapuj}: \text{option} < 'T > \rightarrow ('T \rightarrow' R) \rightarrow \text{option} < 'R >$$

Mapowanie



Mapowanie

```
let mapuj mapa opcja =
    match opcja with
    | Some x -> Some (mapa x)
    | None -> None
```

Funktory

$$\mathbf{mapuj}: \mathrm{M} <' \mathrm{T} > \rightarrow (' \mathrm{T} \rightarrow ' \mathrm{R}) \rightarrow \mathrm{M} < ' \mathrm{R} >$$

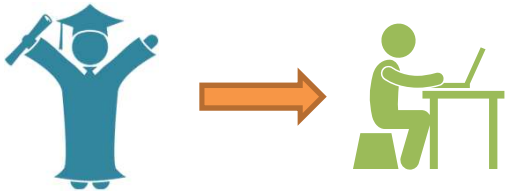
W programowaniu funkcyjnym dowolny typ wyposażony w funkcję mapuj nazywamy funktorem

Prawa funktorów

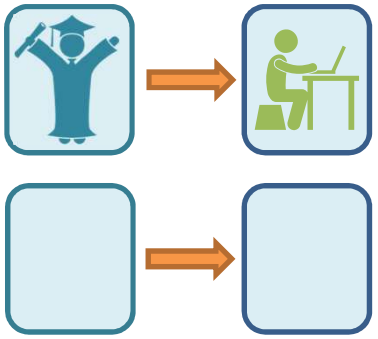
Mapowanie musi zachowywać strukturę kontenera

Kandydat: Option<Student>

uczelnia: Student->Inżynier



mapuj uczelnia: Option<Student>->Option<Inżynier>



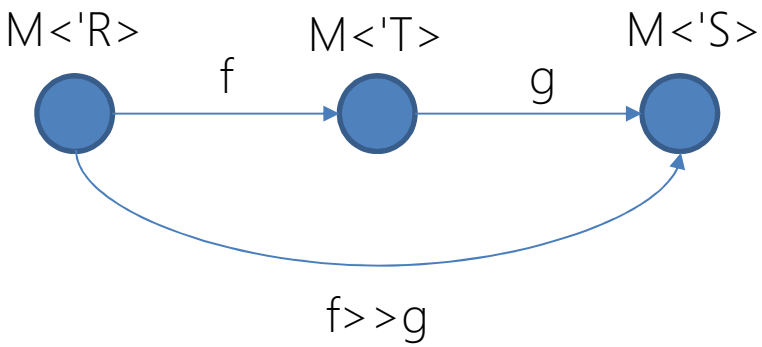
Prawa funktorów

Mapowanie musi zachowywać strukturę kontenera

1. `mapuj id = id`

id: x->x

2. `mapuj (f>>g) = mapuj f >> mapuj g`



Prawa funktorów - opcje

```
let id x = x

let mapujId = mapuj id
```

```
let mapujId opcja =
    match opcja with
    | Some x -> Some (id x)
    | None -> None
```

```
(mapujId (Some x)) = (Some x)

(id (Some x)) = (Some x)
```

```
(mapujId None) = None

(id None) = None
```

```
mapuj id == id
```

Prawa funktorów - opcje

```
(mapuj f (Some x)) = (Some (f x))
(mapuj g (Some (f x)) = (Some (g (f x)))
(mapuj (f>>g) (Some x)) = (Some (f>>g x))
```

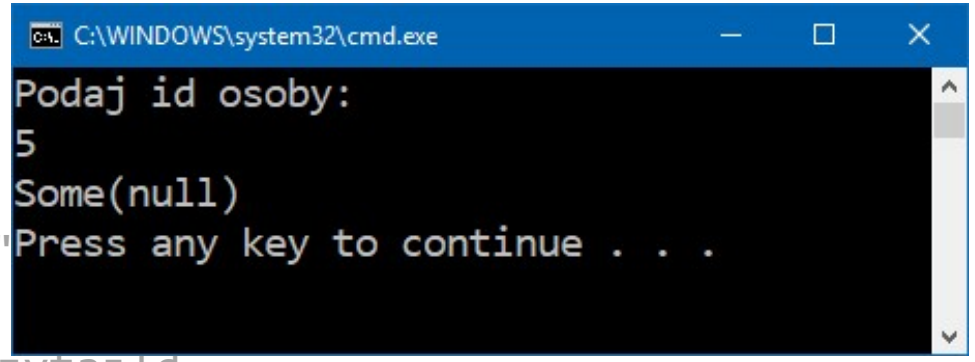
```
(mapuj f None) = None
(mapuj g None) = None
(mapuj (f>>g) None) = None
```

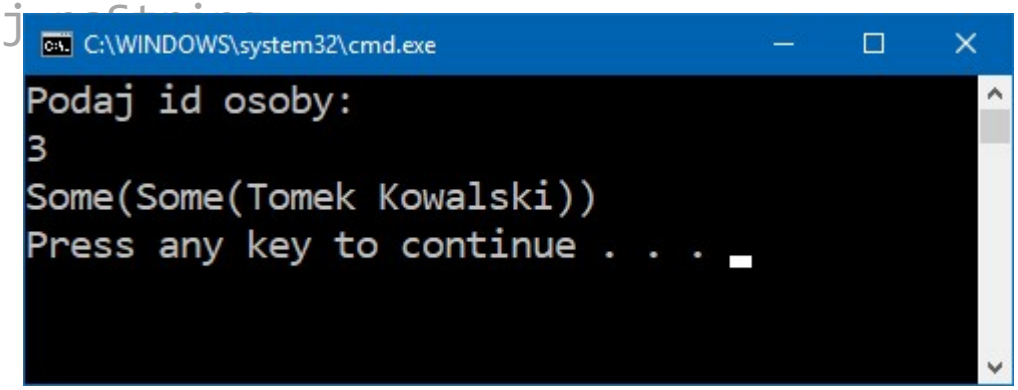
Mapowania

```
let naString (osoba:Osoba) =
     sprintf "%d\n%s\n%s" osoba.id osoba.imie osoba.nazwisko

[<EntryPoint>]
let main argv =
    wczytajDane "dane.json"
    |> … (fun dane->
                    loop wczytajId
                    |> znajdzOsobePoId dane
                    |> mapuj naString
        )
     |> printfn "%A"
    0
```

Mapowania

```
[<EntryPoint>]
let main argv =
    wczytajDane "dane.json"
    |> mapuj (fun dane->
                    loop wczytajId
                    |> znajdzOsobePoId dane
                    |> mapuj wyświetl
        )
    |> printfn "%A"
    0
```





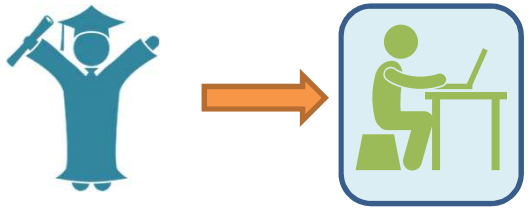
Funktory

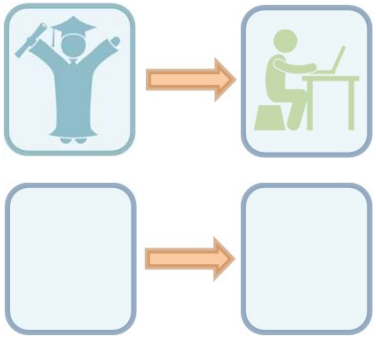
Czy funkcja uczelnia jest poprawnie zdefiniowana?

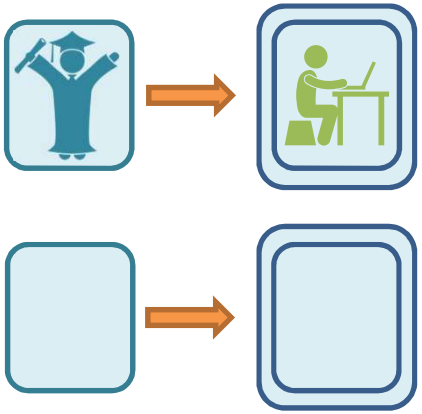
Kandydat: Option<Student>

mapuj uczelnia: Option<Student>
->Option<Option<Inżynier>

uczelnia: Student->Opcja<Inżynier>

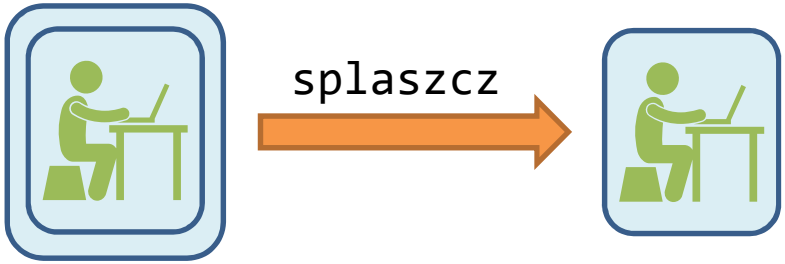






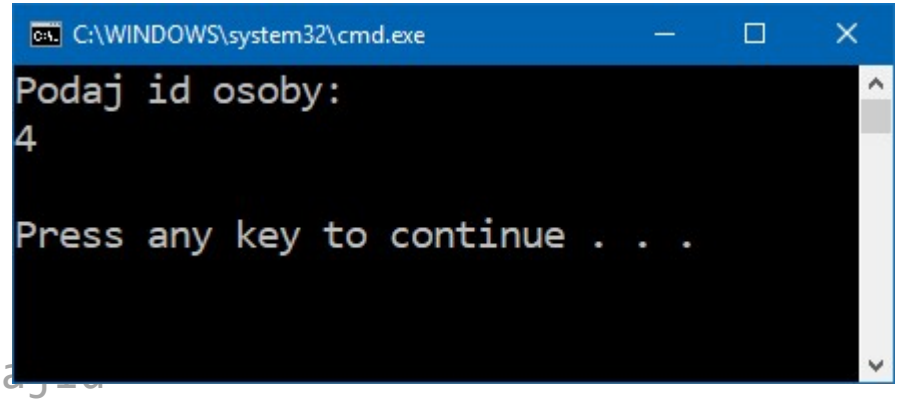
Spłaszczanie opcji

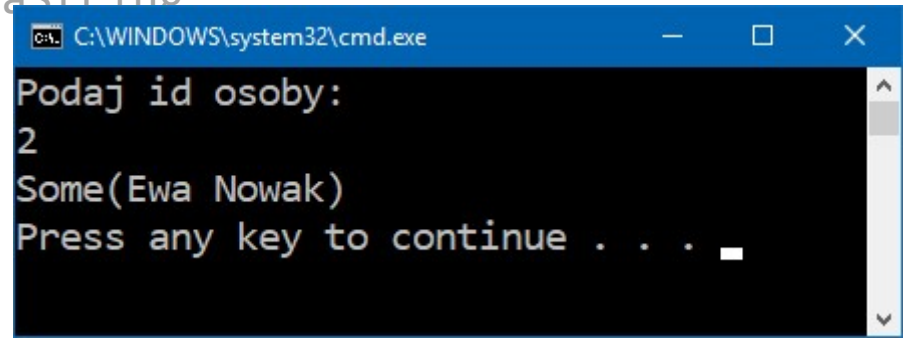
```
let splaszcz opcja =
    match opcja with
    | Some (Some x) -> Some x
    | None -> None
```



Spłaszczanie opcji

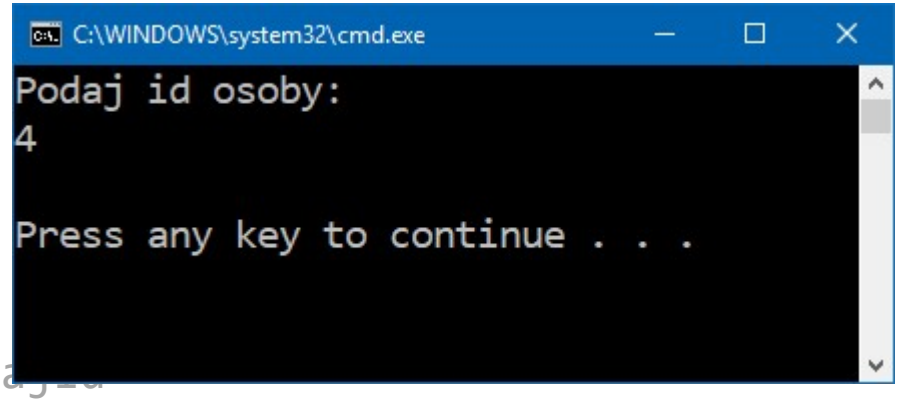
```
[<EntryPoint>]
let main argv =
    wczytajDane "dane.json"
    |> mapuj (fun dane->
                    loop wczytaj
                    |> znajdzOsobePoId dane
                    |> mapuj naString
        )
    |> splaszcz
    |> printfn "%A"
    0
```

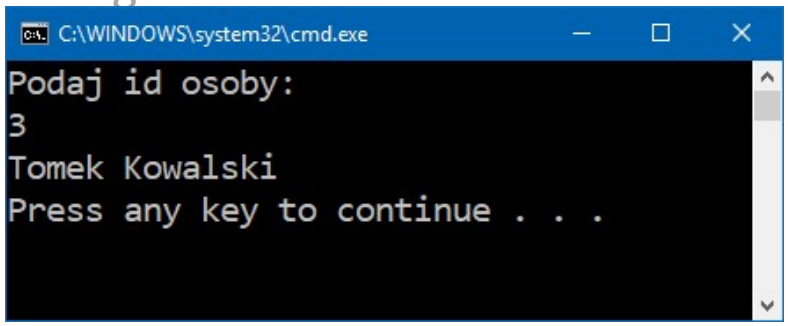




Opcje

```
[<EntryPoint>]
let main argv =
    wczytajDane "dane.json"
    |> mapuj (fun dane->
                    loop wczytaj
                    |> znajdzOsobePoId dane
                    |> mapuj naString
        )
     |> splaszcz
     |> wykonaj (printfn "%s")
    0
```



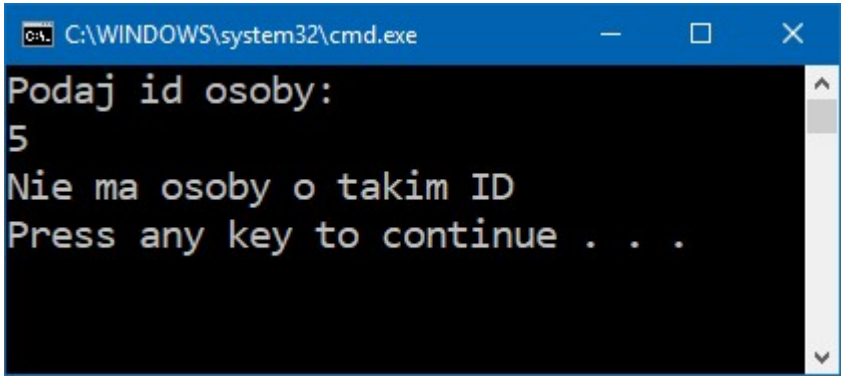


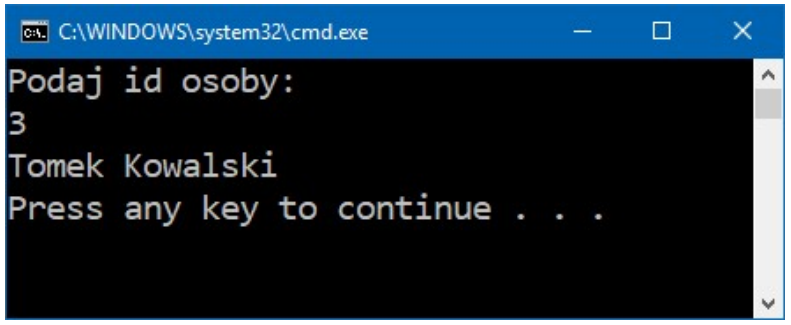
Opcje

```
let gdyNieMaWartosci zastepcza opcja =
    match opcja with
    | Some x -> Some x
    | None -> Some zastepcza
```

Mapowania

```
[<EntryPoint>]
let main argv =
    wczytajDane "dane.json"
    |> mapuj (fun dane->
                    loop wczytajId
                    |> znajdzOsobePoId dane
                    |> mapuj naString
                    |> gdyNieMaWartosc
       )
    |> splaszcz
    |> wykonaj (printfn "%A")
    0
```





Bindowanie

$$\text{zlacz: } M < 'T > \rightarrow ('T \rightarrow M < 'R >) \rightarrow M < 'R >$$

$$\text{zlacz: option} < 'T > \rightarrow ('T \rightarrow \text{option} <' R >) \rightarrow \text{option} < 'R >$$

```
let zlacz opcja mapa =
        match opcja with
        | Some x -> mapa x
        | None -> None
```

Monady

$$\mathbf{zwroc}: \mathrm{T} \rightarrow \mathrm{M} < {'\mathrm{T}} >$$

$$\mathbf{zwroc}: (\mathrm{T}) \rightarrow \mathbf{option} < {'\mathrm{T}} >$$

W programowaniu funkcyjnym dowolny typ wyposażony w funkcje zlacz i zwroc nazywamy monadą

Prawa monad

1. Tożsamość lewostrona

```
zlacz zwroc(x) g = g(x)
```

2. Tożsamość prawostronna

```
zlacz M(x) zwroc = M(x)
```

3. Łączność

```
zlacz M(x) (fun x-> zlacz f(x) g)
zlacz (zlacz M(x) f) g
```

Sprawdzenie praw monad dla opcji (1)

```
let zlacz opcja mapa =
        match opcja with
        | Some x -> mapa x
        | None -> None
```

g: 'A->Option<'B>

1. Tożsamość lewostrona

```
zlacz zwroc(x) g = g(x)
```

```
zlacz Some(x) g  ⟺  g(x)
```

Sprawdzenie praw monad dla opcji (2)

```
let zlacz opcja mapa =
        match opcja with
        | Some x -> mapa x
        | None -> None
```

2. Tożsamość prawostronna

`zlacz M(x) zwroc = M(x)`

`zlacz Some(x) Some` ⟺ `Some(x)`

Sprawdzenie praw monad dla opcji (3)

```
let zlacz opcja mapa =
        match opcja with
        | Some x -> mapa x
        | None -> None
```

3. Łączność

```
zlacz M(x) (fun x-> zlacz f(x) g) =
zlacz (zlacz M(x) f) g
```

```
zlacz Some(x) (fun x -> zlacz f(x) g)  ⟹  zlacz f(x) g

zlacz (zlacz Some(x) f) g  ⟹  zlacz f(x) g
```

Bindowanie

```
[<EntryPoint>]
let main argv =
    wczytajDane "dane.json"
    |> zlacz (fun dane->
                    loop wczytajId
                    |> znajdzOsobePoId dane
                    |> mapuj naString
                    |> gdyNieMaWartosci "Nie ma osoby o takim ID"
        )
     |> wykonaj (printfn "%s")
    0
```

Moduł Option

Operacja	Funkcja
Mapuj	Option.map
Złącz	Option.bind
Wykonaj	Option.iter
Spłaszcz	Option.flatten
GdyNieMaWartości	Option.defaultValue

Złożenie funkcji

```
let inline (>>) func1 func2 x = func2 (func1 x)
```

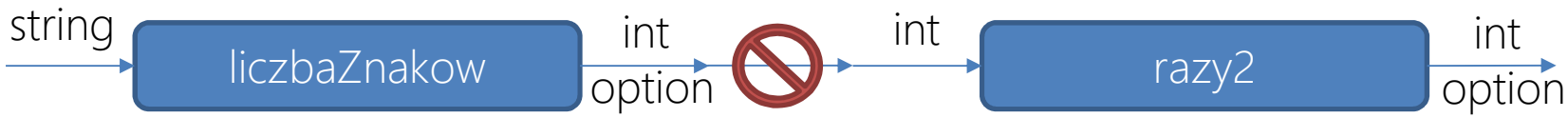
```
let liczbaZnakow str = String.length str
let razy2 x = x*2
```

```
liczbaZnakow >> razy2
```

Złożenie funkcji monadycznych

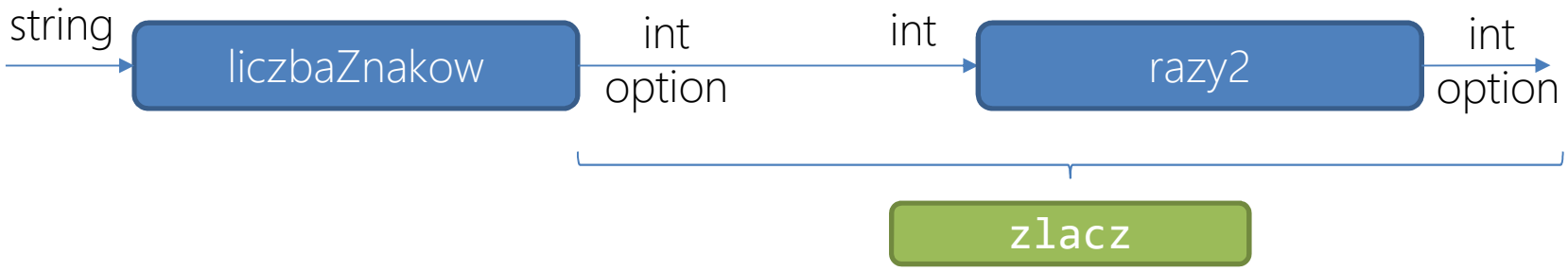
```
let inline (>>) func1 func2 x = func2 (func1 x)
```

```
let liczbaZnakow str = Some(String.length str)
let razy2 x = Some(x*2)
```



Złączenie funkcji monadycznych

$$\mathbf{zlacz:}\ \mathrm{M}<'\mathrm{T}>\rightarrow('\mathrm{T}\rightarrow\mathrm{M}<'\mathrm{R}>)\rightarrow\mathrm{M}<'\mathrm{R}>$$




```
let (>>=) func1 func2 x = (func1 x) |> Option.bind func2
```

```
liczbaZnakow >>= razy2
```

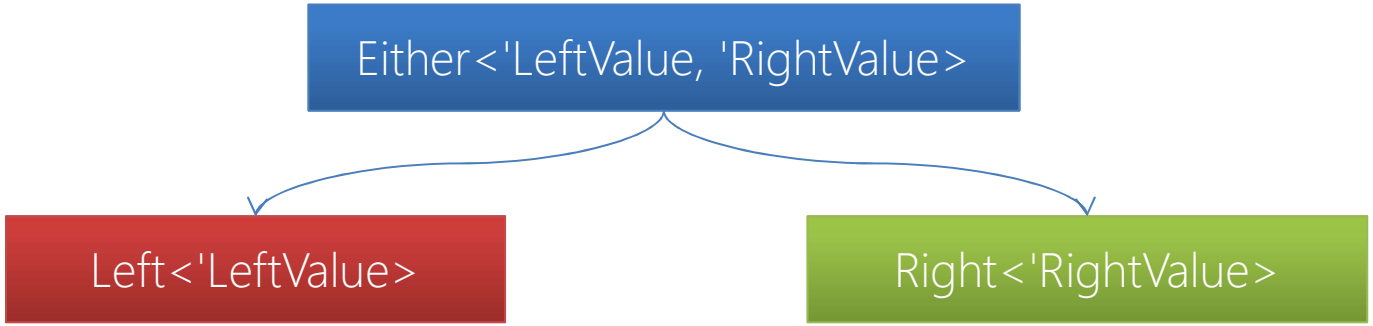
MONADA EITHER (RESULT)

Either (Result)

Monada Either jest funkcyjną metodą pozwalającą na rozwiązanie problemu obsługi błędów

Realizacja Railway Oriented Programming

Monada Either

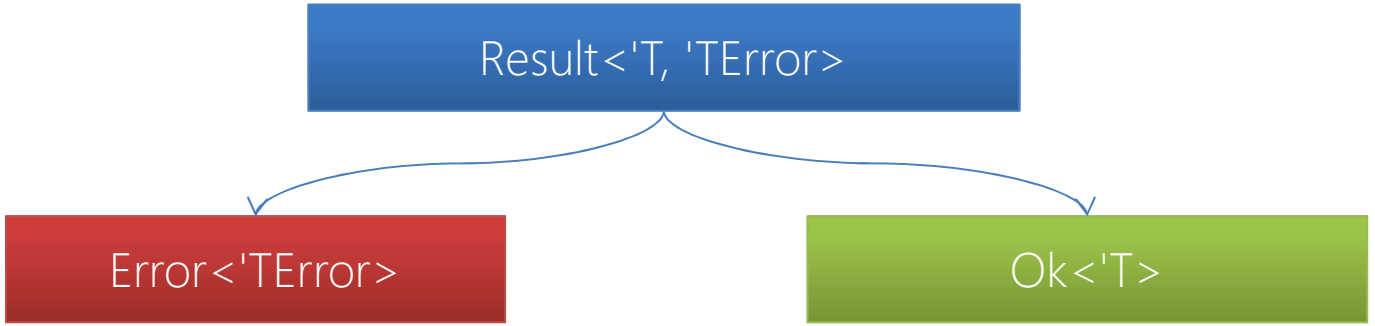


Najczęściej przenosi informacje o błędzie

Najczęściej przenosi informacje poprawne

```
type either<'LeftValue,'RightValue> =
    | Left of 'LeftValue
    | Right of 'RightValue
```

Monada Result



Przenosi informacje o błędzie

Przenosi informacje poprawne

```
type Result<'T,'TError> =
    | Ok of 'T
    | Error of 'TError
```

Monada Result

```
let map mapaPrawa opcje =
        match opcje with
        | Error e -> Error e
        | Ok x -> Ok (mapaPrawa x)
```

```
let bind mapaPrawa opcje =
        match opcje with
        | Error e -> Error e
        | Ok x -> mapaPrawa x
```

Monada Result

```
let akcjaBlad akcja wynik =
    match wynik with
    | Error e -> akcja e
    | Ok _ -> ()
    wynik

let akcjaOk akcja wynik =
    match wynik with
    | Error _ -> ()
    | Ok x -> akcja x
    wynik
```

```
let akcja akcjaOk akcjaBlad =
  function
  | Error e -> akcjaBlad e
  | Ok x -> akcjaOk x
```

Monada Result

```
let wczytajDane nazwaPliku =
        try
            File.ReadAllText nazwaPliku
            |> JsonConvert.DeserializeObject<Osoba list>
            |> Ok
        with
        | _ -> Error "Tego błędu to już nie naprawię"
```

Monada Result

```
let znajdzOsobePoId (lista:Osoba list) id =
        let poId (osoba:Osoba) = osoba.id = id

        if lista |> Seq.exists poId then
           Ok (lista |> List.find poId)
        else
           Error "Nie ma osoby o takim id"
```

Monada Result

```
let parse str =
        try
            Ok (int str)
        with
            | _ -> Error "To musi być liczba"

let rec loop f =
        match f () with
        | Ok x -> x
        | Error e -> Console.Clear()
                     pokazBlad e
                     loop f
```

Monada Result

```
let przetworz dane =
        loop wczytajId
        |> znajdzOsobePoId dane
        |> Result.map naString

[<EntryPoint>]
let main argv =
        wczytajDane "dane.json"
        |> Result.bind przetworz
        |> akcja (printfn "%s") pokazBlad
```

Co powinna zwrócić ta funkcja?

Czy poniższa funkcja powinna zwracać wartość typu Result?

```
let znajdzOsobePoId (lista:Osoba list) id =
        let poId (osoba:Osoba) = osoba.id = id

        if lista |> Seq.exists poId then
           Ok (lista |> List.find poId)
        else
           Error "Nie ma osoby o takim id"
```

Co powinna zwrócić ta funkcja?

A jeżeli będzie zwracała opcję?

```
let znajdzOsobePoId (lista:Osoba list) id =
        let poId (osoba:Osoba) = osoba.id = id

        if lista |> Seq.exists poId then
            Some (lista |> List.find poId)
        else
            None
```

Jak to zmieni funkcję main?

```
[<EntryPoint>]
let main argv =

    wczytajDane "dane.json"
    |> Result.map (pair (loop wczytajId))
    |> Result.map (fun (d, id) -> znajdzOsobePoId d id)
    |> Result.map (Option.map naString)
    |> Result.map
         (Option.defaultValue "Nie ma osoby o takim id")
    |> action (printfn "%s") pokazBlad
```

Osoba list->int->Osoba option

Result<Osoba option, string>

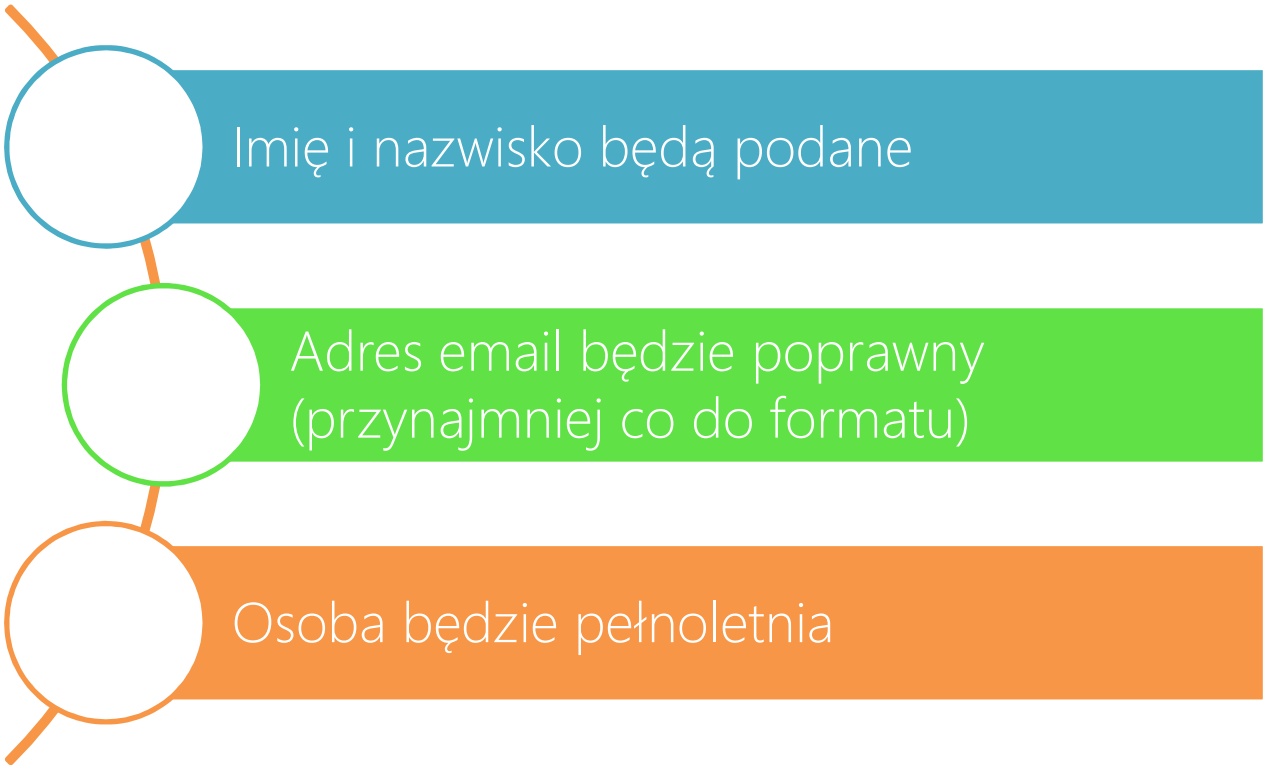
WALIDACJA W STYLU FUNKCYJNYM

Niech dane będą dane

```
type Osoba = {
    id:int
    imie:string
    nazwisko:string
    email:string
    wiek:int
}
```

```
let stworzOsobe id imie nazwisko email wiek =
    {id= id;
     imie = imie;
     nazwisko = nazwisko;
     email = email;
     wiek=wiek}
```

Chcemy mieć pewność, że:



Postać walidatora

```
let pelnoletnia wiek =
    if wiek>=18 then Ok(wiek)
    else Error("Osoba musi być pełnoletnia")
```

'TypWartosci -> Result<'TypWartosci, string>

Walidatory

```
let stringNiePusty (str:string) =
    if not (String.IsNullOrWhiteSpace(str)) then Ok(str)
    else Error("String nie może być pusty")

let pelnoletnia wiek =
    if wiek>=18 then Ok(wiek)
    else Error("Osoba musi być pełnoletnia")

let emailPoprawny email =
    let wzorzec = @"^([\w\.\-]+)@([\w\-]+)((\.(\w){2,3})+)$"
    if Regex.IsMatch(email, wzorzec) then Ok(email)
    else Error("To musi być poprawny adres email")
```

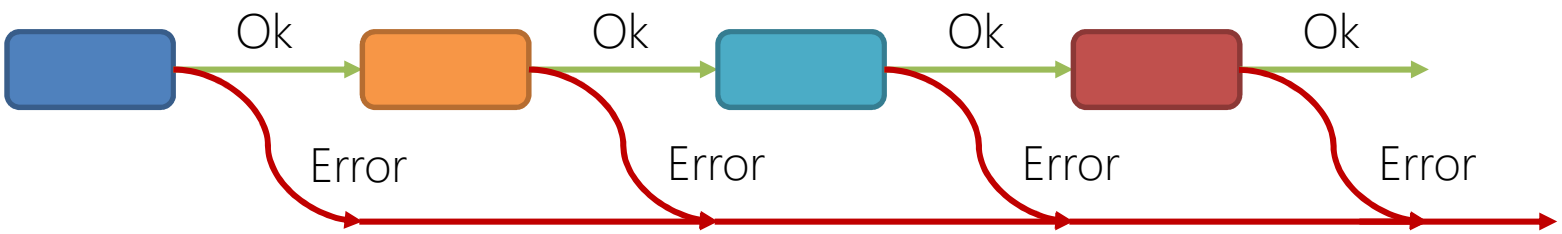
Połączenie za pomocą bind

```
let walidujOsobe id imie nazwisko email wiek =
    stringNiePusty imie
    |> Result.bind (fun _-> stringNiePusty nazwisko)
    |> Result.bind (fun _-> emailPoprawny email)
    |> Result.bind (fun _-> pelnoletni wiek)
    |> Result.bind (fun _-> Ok(stworzOsobe id
                                           imie
                                           nazwisko
                                           email
                                           wiek)
                    )
```

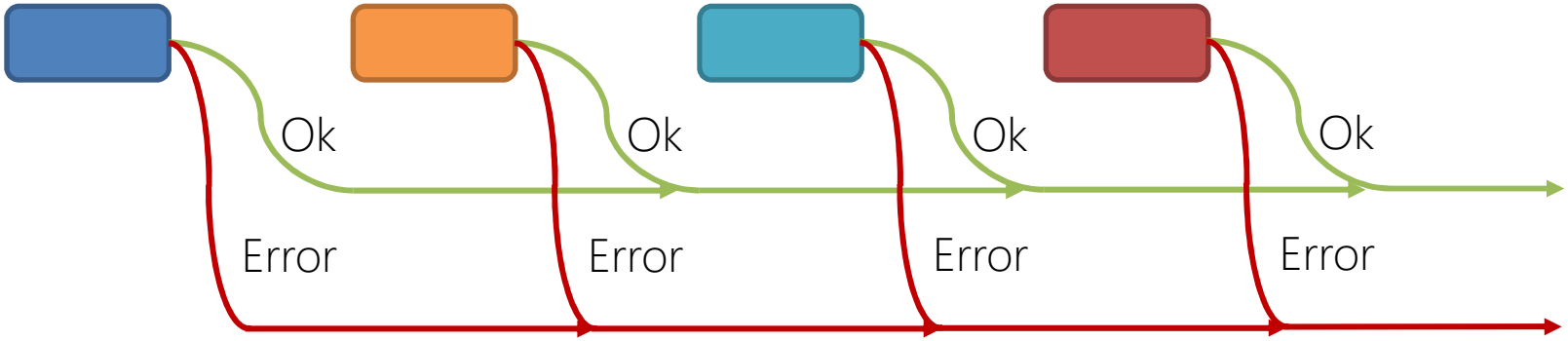
Połączenie za pomocą bind

```
let walidujOsobe id imie nazwisko email wiek =
    stringNiePusty imie
    >>= (fun _ -> stringNiePusty nazwisko)
    >>= (fun _ -> emailPoprawny email)
    >>= (fun _ -> pelnoletni wiek)
    >>= (fun _ -> Ok(stworzOsobe id imie nazwisko email wiek))
```

Problem z tym sposobem walidacji



Problem z tym sposobem walidacji




```
stworzOsobe id (stringNiePusty imie)
               (stringNiePusty nazwisko)
               (emailPoprawny email)
               (pelnoletnia wiek)
```

Aplikatywy

f: int->cokolwiek

opcje: int option

$$\text{mapuj: M} <' \text{ T} > \rightarrow (' \text{T} \rightarrow ' \text{R}) \rightarrow \text{M} < ' \text{R} >$$

```
let mapuj mapa opcja =
    match opcja with
    | Some x -> Some (mapa x)
    | None -> None
```

Aplikatywy

```
let dodaj a b = a+b
```

```
mapuj dodaj (Some 1)
```

```
match (Some 1) with
    | Some x -> Some (dodaj x)
    | None -> None
```

int->int

(int->int) option

Aplikatywy

```
let aplikuj param funkcja =
    match funkcja, param with
    | Some fn, Some v -> Some(fn v)
    | None, Some _ -> None
    | Some _, None -> None
    | None, None -> None
```

$$\text{aplikuj: } M <' T > \rightarrow M\langle' T \rightarrow ' R\rangle \rightarrow M < ' R >$$

Aplikatywy

$$\text{aplikuj}: M <' T > \rightarrow M\langle' T \rightarrow ' R\rangle \rightarrow M < ' R >$$

```
Some(dodaj)
|> aplikuj (Some 1)
|> aplikuj (Some 2)
```

```
mapuj dodaj (Some 1)
|> aplikuj (Some 2)
```

Aplikatywy

$$\text{aplikuj: M} <' \text{ T} > \rightarrow \text{M}\langle'\text{T} \rightarrow 'R\rangle \rightarrow \text{M} < 'R >$$

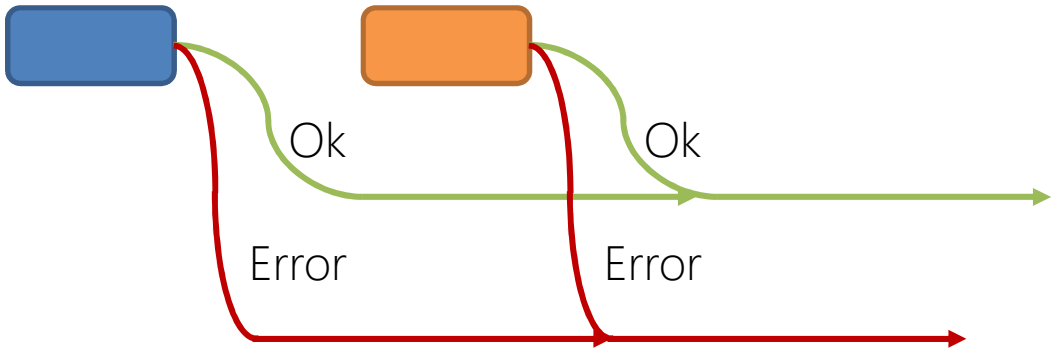
```
let (<*>) funkcja wartosc = mapuj funkcja wartosc
let (<**>) funkcja wartosc = aplikuj wartosc funkcja
```

```
Some(dodaj) <**> (Some 1) <**> (Some 2)
```

```
dodaj <*> (Some 1) <**> (Some 2)
```

Aplikatywy - Result

```
let aplikuj parametr funkcja =
    match funkcja, parametr with
    | Ok fn, Ok v -> Ok(fn v)
    | Error e, Ok _ -> Error e
    | Ok _, Error e -> Error [e]
    | Error e1, Error e2 -> Error (e2::e1)
```



Aplikatywy - Result

```
let aplikuj parametr funkcja =
    match funkcja, parametr with
    | Ok fn, Ok v -> Ok(fn v)
    | Error e, Ok _ -> Error e
    | Ok _, Error e -> Error [e]
    | Error e1, Error e2 -> Error (e2::e1)

let (<**>) funkcja wartosc = aplikuj wartosc funkcja
```

Walidacja za pomocą apply

```
let walidujOsobe id imie nazwisko email wiek =
    Ok (stworzOsobe id)
    |> aplikuj (stringNiePusty imie)
    |> aplikuj (stringNiePusty nazwisko)
    |> aplikuj (emailPoprawny email)
    |> aplikuj (pelnoletni wiek)
```

```
let walidujOsobe id imie nazwisko email wiek =
    Ok(stworzOsobe id)
    <**> stringNiePusty imie
    <**> stringNiePusty nazwisko
    <**> emailPoprawny email
    <**> pelnoletni wiek
```

PROGRAMOWANIE GENERYCZNE

Funkcja zamien (C#)

```
…
void zamien(ref int a, ref int b)
{
    int tmp = a;
    a=b;
    b=tmp;
}
…
```

```
…
void zamien(ref double a, ref double b)
{
    double tmp = a;
    a=b;
    b=tmp;
}
…
```

```
…
void zamien(ref string a,ref string b)
{
    string tmp = a;
    a=b;
    b=tmp;
}
…
```

```
…
void zamien(ref char a, ref char b)
{
    char tmp = a;
    a=b;
    b=tmp;
}
…
```

Klasa Punkt (C#)

```
class Para {
  public int Pierwszy { get; }
  public int Drugi { get; }

  public Para(int pierwszy,
              int drugi)
  {
     Pierwszy = pierwszy;
     Drugi = drugi;
  }
}
```

```
class Para {
  public char Pierwszy { get; }
  public char Drugi { get; }

  public Para(char pierwszy,
              char drugi)
  {
     Pierwszy = pierwszy;
     Drugi = drugi;
  }
}
```

Klasa Punkt (C#)

```
class Para {
  public object Pierwszy { get; }
  public object Drugi { get; }

  public Para(object pierwszy,
              object drugi)
  {
     Pierwszy = pierwszy;
     Drugi = drugi;
  }
}
```

Programowanie generyczne

Programowanie generyczne polega na tworzeniu algorytmów i struktur danych operujących na danych dowolnego typu (lub prawie dowolnego typu).

Paradygmat programowania generycznego jest najczęściej rozumiany jako uogólnienie paradygmatu programowania obiektowego.

Programowanie generyczne w językach programowania

Generyki

Szablony

Metody generyczne w C#

```
class Klasa
{
   public static void zamien<T> (ref T a, ref T b)
   {
      T tmp = a;
      a=b;
      b=tmp;
   }
}
```

Klasy generyczne w C#

```
class Para<T> {
  public T Pierwszy { get; }
  public T Drugi { get; }

  public Para(T pierwszy,
              T drugi)
  {
     Pierwszy = pierwszy;
     Drugi = drugi;
  }
}
```

Klasy generyczne w C#

```
class Para<T> {
  public T Pierwszy { get; }
  public T Drugi { get; }

  public Para()
  {
     Pierwszy = default(T);
     Drugi = dafault(T);
  }
}
```

Metody generyczne

```
class Para<T> {
  public T Pierwszy { get; private set;}
  public T Drugi { get; }

  public Para()
  {
     Pierwszy = default(T);
     Drugi = dafault(T);
  }

  void Zmien(T pierwszy) { Pierwszy = pierwszy; }
}
```

Wiele zmiennych typu

```
class Para<TPierwszy, TDrugi> {
  public TPierwszy Pierwszy { get; private set;}
  public TDrugi Drugi { get; }

  public Para(TPierwszy pierwszy, TDrugi drugi)
  {
     Pierwszy = pierwszy;
     Drugi = drugi;
  }
}
```

Ograniczenia generyczne (1)

```
class Para<T>
{
      public T Pierwszy { get; }
      public T Drugi { get; }

      public Para(T pierwszy, T drugi)
      {
         Pierwszy = pierwszy;
         Drugi = drugi;
      }

      public T max()
      {
        return Pierwszy < Drugi ? Drugi : Pierwszy;
      }
}
```

Ograniczenia generyczne (1)

```
class Para<T> where T:IComparable<T>
{
     public T Pierwszy { get; }
     public T Drugi { get; }

     public Para(T pierwszy, T drugi)
     {
        Pierwszy = pierwszy;
        Drugi = drugi;
     }

     public T max()
     {
       return Pierwszy.CompareTo(Drugi) < 0
                  ? Drugi : Pierwszy;
     }
}
```

Ograniczenia generyczne (2)

```
class Para<T> where T: IComparable<T>, ICloneable
{
     public new T Pierwszy { get; }
     public new T Drugi { get; }

     public Para(T pierwszy, T drugi)
     {
        Pierwszy = pierwszy;
        Drugi = drugi;
     }

     public T max()
     {
       return Pierwszy.CompareTo(Drugi) < 0
                  ? Drugi : Pierwszy;
     }
}
```

Ograniczenia generyczne (3)

```
class ListaUporzadkowana<K, V>
    where K : IComparable<K>
{

}
```

```
class ListaUporzadkowana<K, V>
    where K : IComparable<K>
    where V : ICloneable
{

}
```

Rodzaje ograniczeń generycznych

- ograniczenia dziedziczenia (parametr typu musi dziedziczyć po określonym typie)

```
class KlasaBazowa {}
class MojaKlasa<T> where T : KlasaBazowa {}
```

```
class MojaKlasa<T,U> where T : U { }
```

- ograniczenia konstruktora (parametr typu musi posiadać konstruktor domyślny)

```
 class MojaKlasa<T> where T : new() {}
```

- ograniczenia typu

```
class MojaKlasa<T> where T : struct {}
class MojaKlasa<T> where T : class {}
```

Kowariancja, kontrawariancja i inwariancja

Kowariancja pozwala na wykorzystanie bardziej szczegółowego typu niż pierwotnie określono

Kontrawariancja pozwala na wykorzystanie mniej szczegółowego typu niż pierwotnie określono

Inwariancja pozwala na wykorzystanie tylko takiego typu, który był oryginalnie określony.

Kowariancja, kontrawariancja i inwariancja

Kowariancja

Typ może być wykorzystany tylko jako typ zwracany

```
interface
przyklad<out T> {
   T metoda();
}
```

Kontrawariancja

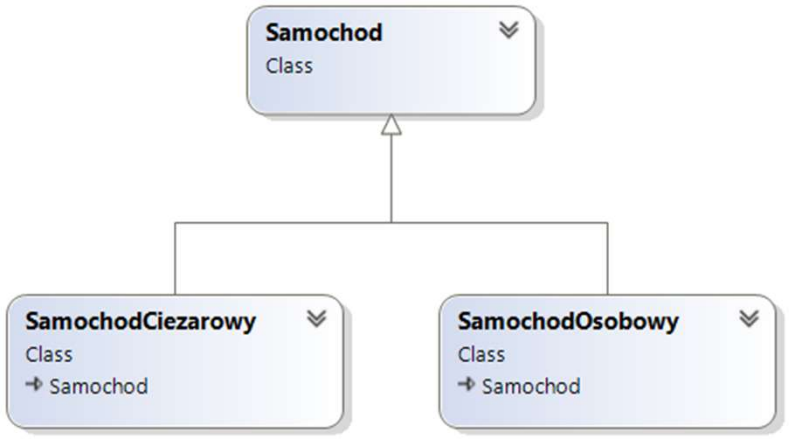
Typ może być wykorzystany tylko jako typ parametru

```
interface
przyklad<in T> {
   void metoda(T p);
}
```

Inwariancja

Typ może być wykorzystany jako typ zwracany lub typ parametru

```
interface
przyklad<T> {
   T metoda();
   void metoda(T p);
}
```




```
class Samochod {}
```

```
class SamochodOsobowy : Samochod {}
```

```
class SamochodCiezarowy : Samochod {}
```

Dziedziczenie, a przypisanie

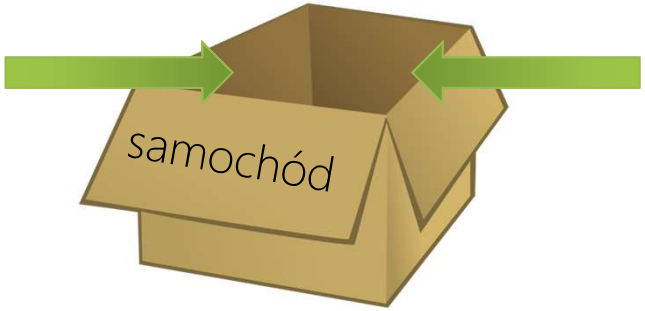
```
interface IFabryka<T>
{
    T Stworz();
}
```

```
class FabrykaFerrari : IFabryka<SamochodOsobowy> {
  public SamochodOsobowy Stworz() {
    throw new NotImplementedException();
  }
}
```

```
class FabrykaScania : IFabryka<SamochodCiezarowy> {
  public SamochodCiezarowy Stworz() {
    throw new NotImplementedException();
  }
}
```

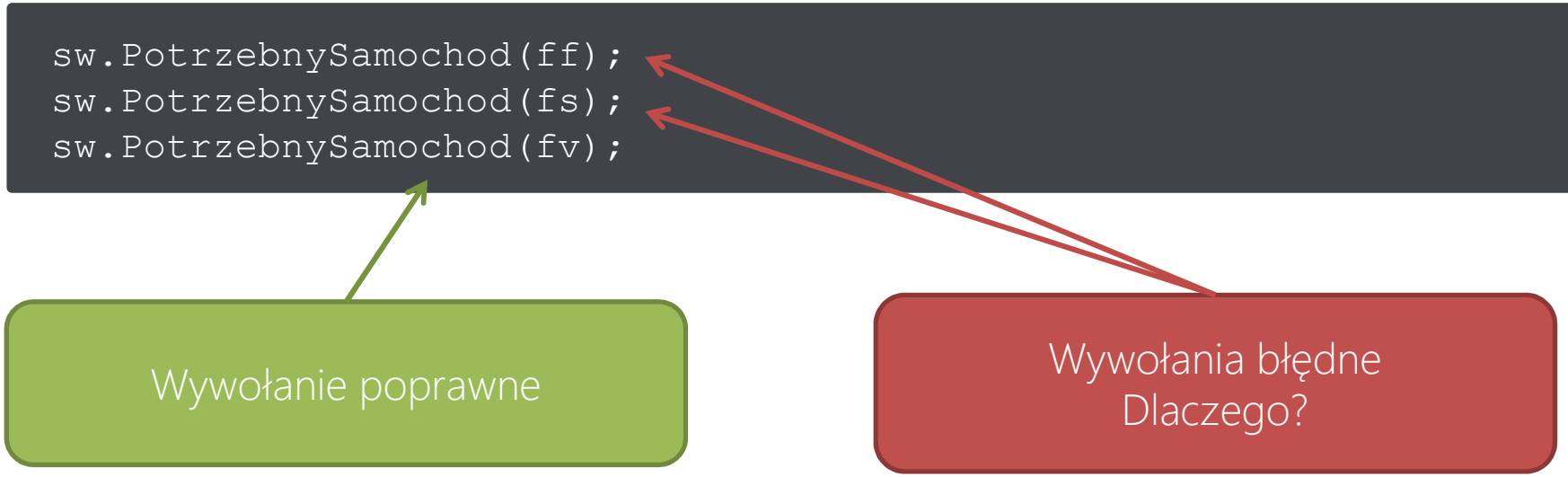
```
class FabrykaVolvo : IFabryka<Samochod> {
  public Samochod Stworz() {
    throw new NotImplementedException();
  }
}
```

```
class SwiatWirutalny {

  private Samochod samochod;

  public void PotrzebnySamochod(IFabryka<Samochod> fabryka) {
    samochod = fabryka.Stworz();
  }
}
```



```
class Program
{
  static void Main(string[] args)
  {
     SwiatWirutalny sw = new SwiatWirutalny();
     FabrykaFerrari ff = new FabrykaFerrari();
     FabrykaScania  fs = new FabrykaScania();
     FabrykaVolvo   fv = new FabrykaVolvo();
     …
```

```
sw.PotrzebnySamochod(ff);
sw.PotrzebnySamochod(fs);
sw.PotrzebnySamochod(fv);
```



Interfejs `IFabryka<T>` jest inwariantny

Inwariancja zwala na wykorzystanie tylko takiego typu, który był oryginalnie określony.

Aby nasz świat stał się idealny musimy interfejs przekształcić w kowariantny

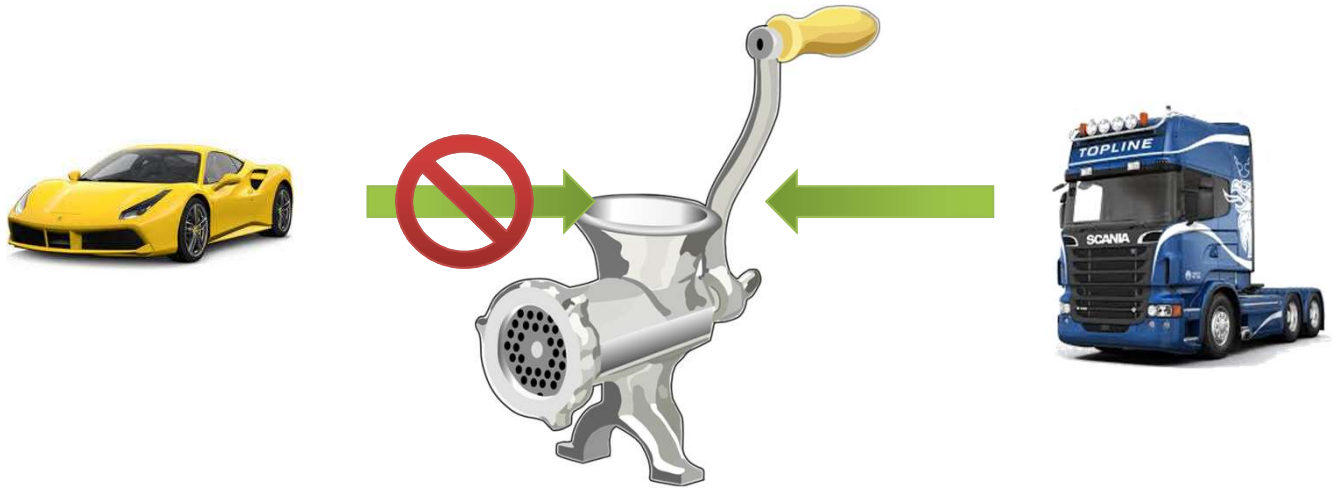
```
interface IFabryka<out T>
{
    T Stworz();
}
```

Kowariancja pozwala na wykorzystanie bardziej szczegółowego typu niż pierwotnie określono

```
interface IService<T>
{
    void Napraw(T obiekt);
}
```

```
class ASOFerrari : ISerwis<SamochodOsobowy> {
  public void Napraw(SamochodOsobowy obiekt) {
    throw new NotImplementedException();
  }
}
```

```
class ASOScania : ISerwis<SamochodCiezarowy> {
  public void Napraw(SamochodCiezarowy obiekt) {
    throw new NotImplementedException();
  }
}
```



```
class SerwisGienka : ISerwis<Samochod> {
  public void Napraw(Samochod obiekt) {
    throw new NotImplementedException();
  }
}
```

```
class SamochodOsobowy : Samochod {
  public void Napraw(ISerwis<SamochodOsobowy> serwis)
  {
    serwis.Napraw(this);
  }
}
```

```
 class SamochodCiezarowy : Samochod {
   public void Napraw(ISerwis<SamochodCiezarowy> serwis)
   {
     serwis.Napraw(this);
   }
 }
```

```
SamochodOsobowy so = new SamochodOsobowy();

so.Napraw(new ASOFerrari());
so.Napraw(new ASOScania());
so.Napraw(new SerwisGienka());
```

Wywołanie poprawne

Wywołania błędne
Dlaczego?

Interfejs `ISerwis<T>` jest inwariantny

Aby nasz świat stał się idealny musimy interfejs przekształcić w kontrawariantny

```
interface ISerwis<in T>
{
    void Napraw(T obiekt);
}
```

WARTOŚCI NIEMODYFIKOWALNE W C#

Dlaczego mutowalne wartości nie zawsze są dobre?

```
class Osoba {
  public string Imie { get; set; }
  public string Nazwisko { get; set; }
  public DateTime DataUrodzenia { get; set; }

  public Osoba(string imie, string nazwisko, DateTime dataUrodzenia)
  {
    Imie = imie;
    Nazwisko = nazwisko;
    DataUrodzenia = dataUrodzenia;
  }
}
```

Załóżmy, że DateTime jest **mutowalny**
(w rzeczywistości jest niemutowalny)

Dlaczego mutowalne wartości nie zawsze są dobre?

```
class Program {

  static void Main(string[] args) {
    var osoba = new Osoba("Ala", "Kot", new DateTime(1998, 01, 12);

    var nowaData = osoba.DataUrodzenia.AddDays(10);

    Console.WriteLine($"Data urodzenia: {osoba.DataUrodzenia}");
  }
}
```

Jaka data urodzenia nam się wyświetli (przy założeniu z poprzedniego slajdu)?

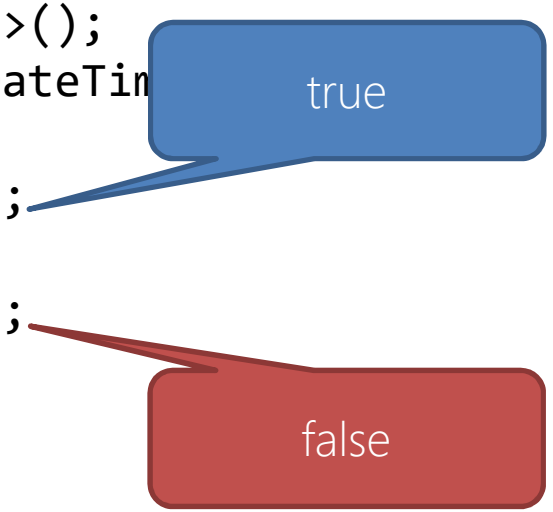
Jaka data urodzenia wyświetli się w rzeczywistości?

Dlaczego mutowalne wartości nie zawsze są dobre?

```
class Osoba {
  public string Imie { get; set; }
  public string Nazwisko { get; set; }
  public DateTime DataUrodzenia { get; set; }

  public Osoba(string imie, string nazwisko, DateTime dataUrodzenia) {
    Imie = imie;
    Nazwisko = nazwisko;
    DataUrodzenia = dataUrodzenia;
  }

  public override int GetHashCode() {
    return Nazwisko.GetHashCode();
  }
}
```

Dlaczego mutowalne wartości nie zawsze są dobre?

```
class Program {

  static void Main(string[] args) {
    HashSet<Osoba> hs = new HashSet<Osoba>();
    var osoba = new Osoba("Ala", "Kot", DateTim
    hs.Add(osoba);
    Console.WriteLine(hs.Contains(osoba));
    osoba.Nazwisko = "Nowak";
    Console.WriteLine(hs.Contains(osoba));
  }

}
```



Wartości modyfikowalne

```
class WartoscWZakresie {
    public int Min { get; set; }
    public int Max { get; set; }
    public int Wartosc { get; set; }

    public WartoscWZakresie(int min, int max, int wartosc) {
      Min=min;
      Max=max;
      Wartosc = wartosc;
    }

    public void ZwiekszWartosc(int oWartosc) {
        Wartosc += oWartosc;
    }
}
```

Wartości modyfikowalne

```
WartoscWZakresie w
    = new WartoscWZakresie(min:-10, max:10, wartosc: 5);

WartoscWZakresie w1
    = new WartoscWZakresie(10, 0, 5);
```

Obiekt jest całkowicie w błędnym stanie

Jeżeli mam obiekt to jest on poprawny

Wartości modyfikowalne

```
class WartoscWZakresie {
    public int Min { get; set; }
    public int Max { get; set; }
    public int Wartosc { get; set; }

    public WartoscWZakresie(int min, int max, int wartosc) {
      Min=min;
      Max=max;
      Wartosc = wartosc;
      SprawdzZakres();
      SprawdzWartosc();
    }
    …
}
```

Wartości modyfikowalne

```
class WartoscWZakresie {
    …
    private void SprawdzZakres() {
       if(Min>Max)
         throw new Exception();
    }

    private void SprawdzWartosc() {
       if(wartosc<Min)
          wartosc = Min;
       if(wartosc>Max)
          wartosc = Max;
    }
}
```

Wartości modyfikowalne

```
class WartoscWZakresie {

    public int Min { get; private set; }
    public int Max { get; private set; }
    public int Wartosc { get; private set; }

    void ZwiekszWartosc(int oWartosc) {
        Wartosc += oWartosc;
    }

}
```

Wartości modyfikowalne

```
class WartoscWZakresie {

    void ZwiekszWartosc(int oWartosc) {
        Wartosc += oWartosc;
        SprawdzWartosc();
    }

}
```

Wartości modyfikowalne

```
class WartoscWZakresie {

   private int _min;

   public int Min {
      get { return _min; }
      set { _min = value; }
   }

}
```

Wartości modyfikowalne

```
class WartoscWZakresie {

   private int _min;
   private int _max;

   public int Min {
      get { return _min; }
      set {
         if(value>_max)
            throw new Exception();
         _min = value;
         SprawdzWartosc();
      }
   }
}
```

Wartości niemodyfikowalne

Wartości niemodyfikowalne są to wartości, których składowe mogą być ustalone tylko w konstruktorze.

Wartości niemodyfikowalne

```
class WartoscWZakresie {

    private readonly int _min;
    private readonly int _max;
    private readonly int _wartosc;

    public int Min { get => _min; }
    public int Max { get => _max; }
    public int Wartosc { get => _wartosc; }

    public WartoscWZakresie(int min, int max, int wartosc) {
      _min=min;
      _max=max;
      _wartosc = wartosc;
      SprawdzZakres();
      SprawdzWartosc();
    }
    …
```

Wartości niemodyfikowalne

```
class WartoscWZakresie {

   …

    public WartoscWZakresie ZwiekszWartosc(int oWartosc) {
        return new WartoscWZakresie(_min, _max, _wartosc+oWartosc);
    }

    public WartoscWZakresie ZmienMin(int min) {
        return new WartoscWZakresie(min, _max, _wartosc);
    }
}
```

Wartości niemodyfikowalne

```
class WartoscWZakresie {

   …

    public WartoscWZakresie ZwiekszWartosc(int oWartosc) {
        return new WartoscWZakresie(_min, _max, _wartosc+oWartosc);
    }

    public WartoscWZakresie ZmienMin(int min) {
        return new WartoscWZakresie(min, _max, _wartosc);
    }
}
```

Wartości niemodyfikowalne

```
class WartoscWZakresie {

    public int Min { get; }
    public int Max { get; }
    public int Wartosc { get; }

    public WartoscWZakresie(int min, int max, int wartosc) {
      Min=min;
      Max=max;
      Wartosc = wartosc;
      SprawdzZakres();
      SprawdzWartosc();
    }
    …
```

Czy ta klasa jest poprawnie zapisana?

Zasada pojedynczej odpowiedzialności

Nigdy nie powinno być więcej niż jednego powodu do **istnienia** klasy lub metody

Tom DeMarco, Meilir Page-Jones

Nigdy nie powinno być więcej niż jednego powodu do **modyfikacji** klasy lub metody

Robert C. Martin

Stwórzmy dwie klasy

```
public class Zakres {
  public int Min { get; }
  public int Max { get; }

  public Zakres(int min, int max) {
    if (min > max) throw new ArgumentException("...");
    Min = min;
    Max = max;
  }

  public Zakres ZmienMin(int min) => new Zakres(min, Max);
  public Zakres ZmienMax(int max) => new Zakres(Min, max);
  public bool WZakresie(int wartosc) => Min <= wartosc && wartosc <= Max;
}
```

Stwórzmy dwie klasy

```
public class WartoscWZakresie {

  public Zakres Zakres { get; }
  public int Wartosc { get; }

  public WartoscWZakresie(Zakres zakres, int wartosc) {

    Zakres = zakres;
    if (wartosc < zakres.Min) Wartosc = zakres.Min;
    else if (zakres.Max < wartosc) Wartosc = zakres.Max;
    else Wartosc = wartosc;
  }

}
```

Czy klasa Zakres jest rzeczywiście niemutowalna?

```
public class NiemutowalnyZakres {

  public int Min { get; }
  public int Max { get; }

  public NiemutowalnyZakres(int min, int max) {
    if (min > max) throw new ArgumentException("...");
    Min = min;
    Max = max;
  }

  public override string ToString() {
    return $"[{Min} {Max}]";
  }
}
```

Czy klasa Zakres jest rzeczywiście niemutowalna?

```
class Program {

  static void Main(string[] args) {

    NiemutowalnyZakres niezmiennik = new NiemutowalnyZakres(-10, 10);


    Console.WriteLine(niezmiennik);
    …
    Console.WriteLine(niezmiennik);
  }

}
```

Czy klasa Zakres jest rzeczywiście niemutowalna?

```
public class ZakresZNazwa : NiemutowalnyZakres {

  public string Nazwa { get; set; }

  public ZakresZNazwa(int min, int max, string nazwa)
            : base(min, max)
  {
    Nazwa = nazwa;
  }

  public override string ToString() {
    return $"{Nazwa}: [{Min} {Max}]";
  }

}
```

Czy klasa Zakres jest rzeczywiście niemutowalna?

```
class Program {

  static void Main(string[] args) {

    var zakres = new ZakresZNazwa(-10, 10, "Jakiś zakres");
    NiemutowalnyZakres niezmiennik = zakres;

    Console.WriteLine(niezmiennik);

    zakres.Nazwa = "Nowa nazwa";

    Console.WriteLine(niezmiennik);
  }

}
```

Jakiś zakres [-10,10]

Nowa nazwa [-10,10]

Jak się przed taką sytuacją obronić?

```
public sealed class NiemutowalnyZakres {

  public int Min { get; }
  public int Max { get; }

  public NiemutowalnyZakres(int min, int max) {
    if (min > max) throw new ArgumentException("...");
    Min = min;
    Max = max;
  }

  public override string ToString() {
    return $"[{Min} {Max}]";
  }
}
```

Przykład bardziej praktyczny

Musimy stworzyć klasę Osoba w której:

- Imię musi być podane, ale mieć nie więcej niż 30 znaków
- Nazwisko musi być podane, ale mieć nie więcej niż 30 znaków
- Wiek może być opcjonalny, ale z sensownego zakresu

Klasa Osoba

```
public sealed class Osoba {
  public string Imie { get; }
  public string Nazwisko { get; }
  public int Wiek { get; }

  public Osoba(string imie, string nazwisko, int wiek)
  {
     ...
  }
}
```

Klasy definiujące wartości

```
class Str30 {
  public string Wartosc { get; }

  public Str30(string wartosc) {
    if (string.IsNullOrWhiteSpace(wartosc) || 30 < wartosc.Length)
       throw new ArgumentException();
    else
       Wartosc = wartosc;
  }

  public static implicit operator string(Str30 str30)
    => str30.Wartosc;
  public static implicit operator Str30(string wartosc)
    => new Str30(wartosc);
}
```

Klasy definiujące wartości

```
class Wiek {
  public int Wartosc { get; }

  public Wiek(int wartosc) {
    if (wartosc < 0 || 140 < wartosc)
       throw new ArgumentException("...");
    Wartosc = wartosc;
  }

  public static implicit operator int(Wiek wiek)
    => wiek.Wartosc;
  public static implicit operator Wiek(int wartosc)
    => new Wiek(wartosc);
}
```

Klasa Osoba

```
class Osoba {

  public Str30 Imie { get; }
  public Str30 Nazwisko { get; }
  public Wiek Wiek { get; }

  public Osoba(Str30 imie, Str30 nazwisko, Wiek wiek) {
    if (imie == null)
       throw new ArgumentNullException(nameof(imie));
    else
       Imie = imie;
    if (nazwisko == null)
       throw new ArgumentNullException(nameof(nazwisko));
    else
       Nazwisko = nazwisko;
  }
}
```

Klasa Osoba

```
class Osoba {
  public Str30 Imie { get; }
  public Str30 Nazwisko { get; }
  public Wiek Wiek { get; }

  public Osoba(Str30 imie, Str30 nazwisko, Wiek wiek) {
    Imie = imie ?? throw new ArgumentNullException(nameof(imie));
    Nazwisko =
       nazwisko ?? throw new ArgumentNullException(nameof(nazwisko));
    Wiek = wiek;
  }
}
```

Klasa Osoba

```
var osoba1 = new Osoba("Tomek", "Nowak", 23);

var osoba2 = new Osoba("Ala", "Kot", null);


var osoba3 = new Osoba("Tomek", "Nowak")
{
   Wiek = 23
};
```

Klasa Osoba

```
class Osoba {
  public Str30 Imie { get; }
  public Str30 Nazwisko { get; }
  public Wiek Wiek { get; set; }

  public Osoba(Str30 imie, Str30 nazwisko) {
    Imie =
       imie ?? throw new ArgumentNullException(nameof(imie));
    Nazwisko =
       nazwisko ?? throw new ArgumentNullException(nameof(nazwisko));
  }
}
```

Właściwości tylko do inicjalizacji

```
class Osoba {
  public Str30 Imie { get; }
  public Str30 Nazwisko { get; }
  public Wiek Wiek { get; init; }

  public Osoba(Str30 imie, Str30 nazwisko) {
    Imie =
       imie ?? throw new ArgumentNullException(nameof(imie));
    Nazwisko =
       nazwisko ?? throw new ArgumentNullException(nameof(nazwisko));
  }
}
```

Co zrobić, aby zrezygnować ze sprawdzania null?

```
struct Str30 {
  public string Wartosc { get; }

  public Str30(string wartosc) {
    if (string.IsNullOrWhiteSpace(wartosc) || 30 < wartosc.Length)
       throw new ArgumentException();
    else
       Wartosc = wartosc;
  }

  public static implicit operator string(Str30 str30)
    => str30.Wartosc;
  public static implicit operator Str30(string wartosc)
    => new Str30(wartosc);
}
```

Klasa Osoba

```
class Osoba
{
  public Str30 Imie { get; }
  public Str30 Nazwisko { get; }
  public Wiek Wiek { get; }

  public Osoba(Str30 imie, Str30 nazwisko)
  {
    Imie = imie;
    Nazwisko = nazwisko;
  }
}
```

Klasa Osoba

```
class Osoba
{
  public Str30 Imie { get; }
  public Str30 Nazwisko { get; }
  public Wiek? Wiek { get; }

  public Osoba(Str30 imie, Str30 nazwisko)
  {
    Imie = imie;
    Nazwisko = nazwisko;
  }
}
```

Rekordy

C# 9 wprowadza nowy rodzaj typu danych - rekord

```
public record Osoba (int Id, string Imie, string Nazwisko)
```

```
public record Osoba (int Id, string Imie, string Nazwisko)
{
  public int Wiek {get;set;}
}
```

- Właściwości pozycyjne są niemutowalne.
- Rekordy porównywane są strukturalnie

Rekordy

Nieniszcząca mutacja

```
var osoba = new Osoba("Ala", "Kot", 23);
var nowaOsoba = osoba with {Nazwisko = "Nowak"}

public record Osoba (int Id, string Imie, string Nazwisko)
{
  public int Wiek {get;set;}
}
```

Dziedziczenie rekordów

Rekordy mogą dziedziczyć tylko po rekordach

Niedopuszczalne jest dziedziczenie rekordów po klasach oraz klas po rekordach

```
public record Osoba(string Imie, string Nazwisko);

public record Student(string Imie, string Nazwisko, int NrIndeksu)
  : Osoba(Imie, Nazwisko);
```

Dekonstrukcja rekodrów

```
var osoba = new Osoba("Ala", "Kot", 23);

var (imie, nazwisko) = osoba;
```

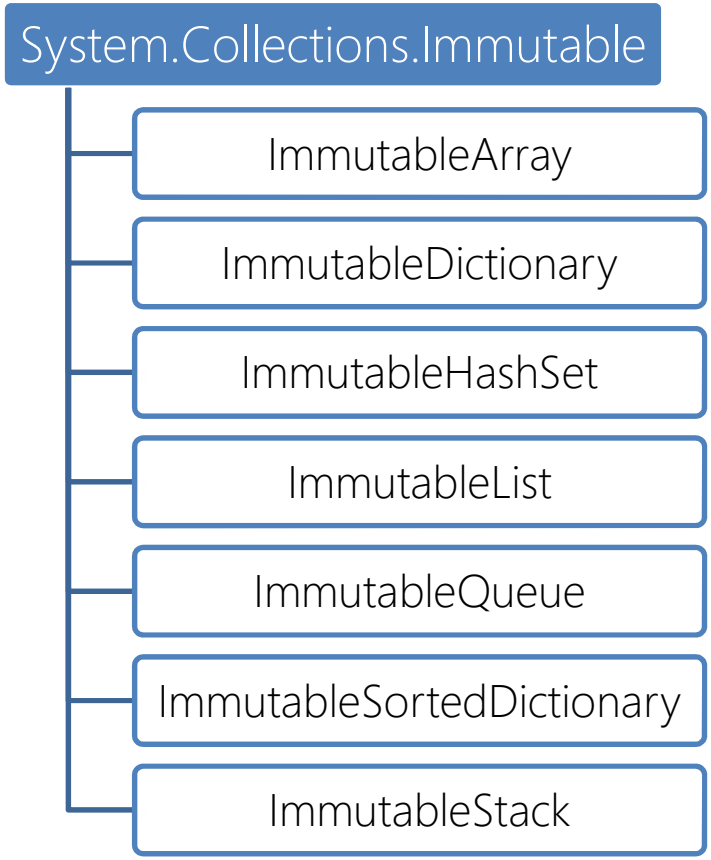
Domyślna dekonstrukcja dotyczy wyłączenie właściwości pozycyjnych

Dekonstrukcja obiektów

```
class Osoba {
  public string Imie { get; set; }
  public string Nazwisko { get; set; }

  public Osoba(string imie, string nazwisko) {
    Imie = imie;
    Nazwisko = nazwisko;
  }

  public void Deconstruct(out string imie, out string nazwisko) {
    imie = Imie;
    nazwisko = Nazwisko;
  }
}
```

System.Collections.Immutable

Niemutowalne kolekcje dostępne są w przestrzeni nazw System.Collections.Immutable



System.Collections.Immutable

```
var lista = ImmutableList.Create<Osoba>();

var nowaLista = lista.Add(new Osoba("Ala", "Kot", DateTime.Now));


Console.WriteLine(lista.Count);
Console.WriteLine(nowaLista.Count);
```

Immutable vs Readonly

```
var lista = ImmutableList.Create<Osoba>();
var nowaLista = lista.Add(new Osoba("Ala", "Kot", DateTime.Now));


var listaDO = new List<Osoba>
                  {new Osoba("Ala", "Kot", DateTime.Now)}
                  .AsReadOnly();

listaDO.Add(new Osoba("Ala", "Kot", DateTime.Now));
```

Immutable vs Readonly

```
var lista = new List<Osoba>
                  { new Osoba("Ala", "Kot", DateTime.Now) };
listaDO = lista.AsReadOnly();

Console.WriteLine(listaDO.Count);

lista.Add(new Osoba("Ewa", "Nowak", DateTime.Now));

Console.WriteLine(listaDO.Count);
```

FUNKCJE WYŻSZYCH RZĘDÓW

Filtrowanie listy

Problem: przefiltrować listę

```
class Program
{
    static void Main(string[] args)
    {
        List<string> przedmioty = new List<string>() {
            "Paradygmaty programowania",
            "Programowanie stron internetowych",
            "Algebra",
            "Logika dla informatyków",
            "Sieci komputerowe"
        };
    }
}
```

Filtrowanie listy

```
class OperacjeNaLiscie {
   public static List<string> TylkoDlugieNazwy(List<string> lista) {
     List<string> rezultat = new List<string>();
     for(int i=0; i<lista.Count; i++)
       if(lista[i].Length > 18)
         rezultat.Add(lista[i]);
     return rezultat;
   }

   public static List<string> TylkoNaLitereP(List<string> lista) {
     List<string> rezultat = new List<string>();
     for(int i=0; i<lista.Count; i++)
       if(lista[i][0] == 'P')
         rezultat.Add(lista[i]);
     return rezultat;
   }
}
```

Filtrowanie listy

```
class Program
{
  static void Main(string[] args)
  {
    List<string> przedmioty = new List<string>() {
      "Paradygmaty programowania",
      …
    };

    List<string> dlugieNazwy
         = OperacjeNaLiscie.TylkoDlugieNazwy(przedmioty);
    List<string> przedmiotyNaLitereP
         = OperacjeNaLiscie.TylkoNaLitereP(przedmioty);
  }
}
```

Filtrowanie listy

Czy można to rozwiązać bardziej elegancko?

Filtrowanie listy

```
class OperacjeNaLiscie {
   public static List<string> TylkoDlugieNazwy(List<string> lista) {
     List<string> rezultat = new List<string>();
     for(int i=0; i<lista.Count; i++)
       if(lista[i].Length > 18)
         rezultat.Add(lista[i]);
     return rezultat;
   }

   public static List<string> TylkoNaLitereP(List<string> lista) {
     List<string> rezultat = new List<string>();
     for(int i=0; i<lista.Count; i++)
       if(lista[i][0] == 'P')
         rezultat.Add(lista[i]);
     return rezultat;
   }
}
```

Wzorzec Strategii

Czynnościowy wzorzec projektowy, który definiuje rodzinę wymiennych algorytmów i kapsułkuje je w postaci klas.

Wzorzec strategii

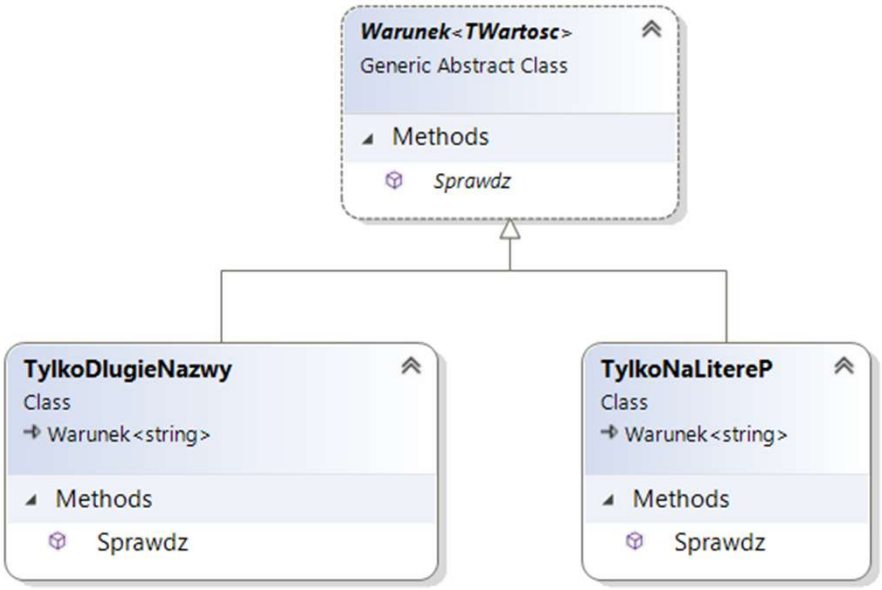
```
abstract class Warunek {
   public abstract bool Sprawdz(string wartosc);
}
```

```
abstract class Warunek <TWartosc> {
   public abstract bool Sprawdz(TWartosc wartosc);
}
```

Wzorzec strategii

```
class TylkoDlugieNazwy : Warunek<string> {

   public override bool Sprawdz(string wartosc) {
     return wartosc.Length > 18;
   }

}

class TylkoNaLitereP : Warunek<string> {

   public override bool Sprawdz(string wartosc) {
     return wartosc[0] == 'P';
   }

}
```

Wzorzec strategii



Wzorzec strategii

```
class OperacjeNaLiscie {
   public static List<string> Filtruj(
       List<string> lista,
       Warunek<string> warunek)
   {
       List<string> rezultat = new List<T>();
          for(int i=0; i<lista.Count; i++)
            if(warunek.Sprawdz(lista[i]))
               rezultat.Add(lista[i]);
       return rezultat;
   }
}
```

Wzorzec strategii

```
class OperacjeNaLiscie {
   public static List<T> Filtruj<T>(
       List<T> lista,
       Warunek<T> warunek)
   {
       List<T> rezultat = new List<T>();
          for(int i=0; i<lista.Count; i++)
            if(warunek.Sprawdz(list[i]))
               rezultat.Add(lista[i]);
       return rezultat;
   }
}
```

Wzorzec strategii

```
class Program {
  static void Main(string[] args)  {
    List<string> przedmioty = new List<string>() {
      "Paradygmaty programowania",
      …
    };

    List<string> dlugieNazwy
      = OperacjeNaLiscie.Filtruj(przedmioty, new TylkoDlugieNazwy());
    List<string> przedmiotyNaLitereP
      = OperacjeNaLiscie.Filtruj(przedmioty, new TylkoNaLitereP());
  }
}
```

Funkcje wyższych rzędów

Funkcje wyższych rzędów są to funkcje, które przyjmują inne funkcje jako parametry, lub funkcje zwracają.

Typy funkcyjne w C#

Typy do opisu funkcji

```
Func<out TResult>
Func<in T, out TResult>
Func<in T1, in T2, out TResult>
Func<in T1,…,in T16, out TResult>
```

Typy do opisu akcji

```
Action<in T>
Action<in T1, in T2>
Action<in T1,…,in T16>
```

Funkcje wyższych rzędów

```
class OperacjeNaLiscie {
   public static List<T> Filtruj<T>(
       List<T> lista,
       Func<T, bool> warunek)
   {
       List<T> rezultat = new List<T>();
          for(int i=0; i<lista.Count; i++)
            if(warunek(list[i]))
               rezultat.Add(lista[i]);
       return rezultat;
   }
}
```

Funkcje wyższych rzędów

```
class Program {

  static bool TylkoDlugie(string wartosc) => wartosc.Length > 18;
  static bool TylkoNaP(string wartosc) => wartosc[0] == 'P';
  static void Main(string[] args)  {

    List<string> przedmioty = new List<string>() {
      "Paradygmaty programowania",
      …
    };

    List<string> dlugieNazwy
      = OperacjeNaLiscie.Filtruj(przedmioty, TylkoDlugie);
    List<string> przedmiotyNaLitereP
      = OperacjeNaLiscie.Filtruj(przedmioty, TylkoNaP);
  }
}
```

Funkcje wyższych rzędów

```
class Program {

  static bool TylkoDlugie(string wartosc) => wartosc.Length > 18;

  static void Main(string[] args)  {

    List<string> przedmioty = new List<string>() {
      "Paradygmaty programowania",
      …
    };

    List<string> dlugieNazwy
      = OperacjeNaLiscie.Filtruj(przedmioty, (nazwa)=>nazwa.Length>18);
  }
}
```

Wzorzec strategii

```
class TylkoNaLitere : Warunek<string> {

   char _litera;
   public TylkoNaLitere(char litera) {
      _litera = litera;
   }

   public bool Sprawdz(string wartosc) {
     return wartosc[0] == _litera;
   }

}
```

Funkcje wyższych rzędów

```
class Program {

  static bool TylkoNaLitere(string nazwa, char litera) => nazwa[0] == litera;

  static Func<string, bool> TylkoNaLitere(char litera) {
     return (nazwa) => TylkoNaLitere(nazwa, litera);
  }

  static void Main(string[] args)  {

  List<string> przedmioty = new List<string>() {
    "Paradygmaty programowania",  …
  };

  List<string> dlugieNazwy
      = OperacjeNaLiscie.Filtruj(przedmioty, TylkoNaLitere('S'));
  }
}
```

Funkcje wyższych rzędów

```
class Program {

  static Func<string, bool> TylkoNaLitere(char litera) {
     return (nazwa) => nazwa[0] == litera;
  }

  static void Main(string[] args)  {

  List<string> przedmioty = new List<string>() {
    "Paradygmaty programowania",  …
  };

  List<string> dlugieNazwy
      = OperacjeNaLiscie.Filtruj(przedmioty, TylkoNaLitere('S'));
  }
}
```

Mierzenie czasu wykonania funkcji

```
static double ZmierzCzas(Action akcja) {
   var stoper = new Stopwatch();
   stoper.Start();
   akcja();
   stoper.Stop();
   return stoper.Elapsed.TotalMilliseconds;
}
```

OBLICZENIA LENIWE

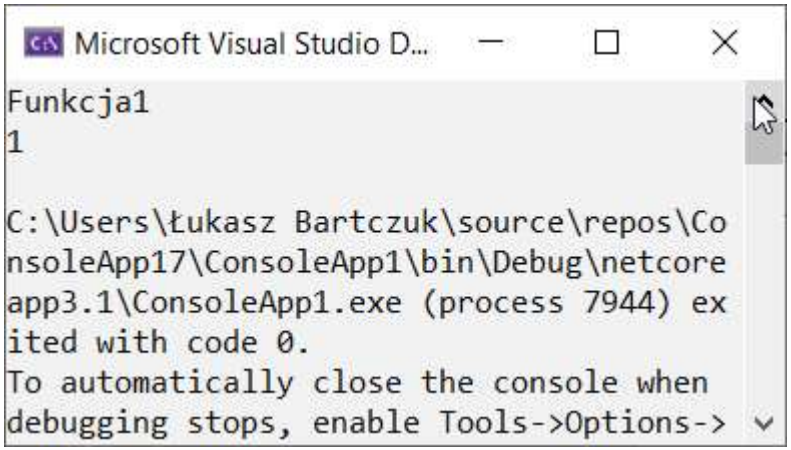
Obliczenia zachłanne (eager computation)

Wyrażenie jest obliczane, gdy tylko jest przypisane do zmiennej

Wartości argumentów są obliczane przed wywołaniem funkcji

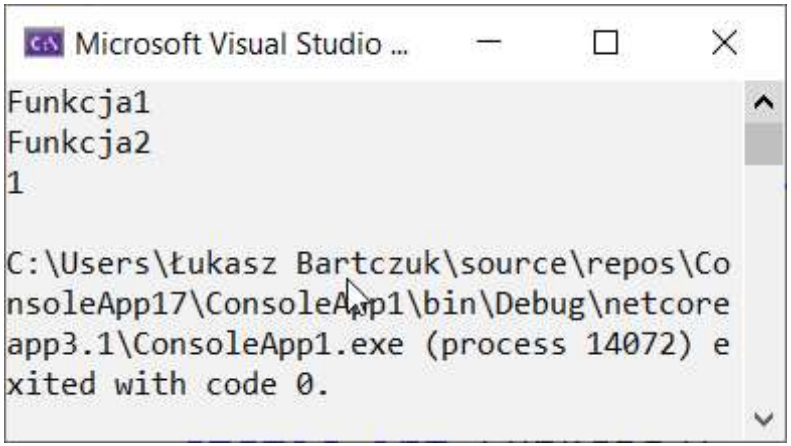
Prosty przykład

```
static int Funkcja1() {
  Console.WriteLine(nameof(Funkcja1));
  return 1;
}

static int Funkcja2() {
  Console.WriteLine(nameof(Funkcja2));
  return 2;
}

static void Main(string[] args) {
  var wynik = true ? Funkcja1() : Funkcja2();
  Console.WriteLine(wynik);
}
```



Prosty przykład

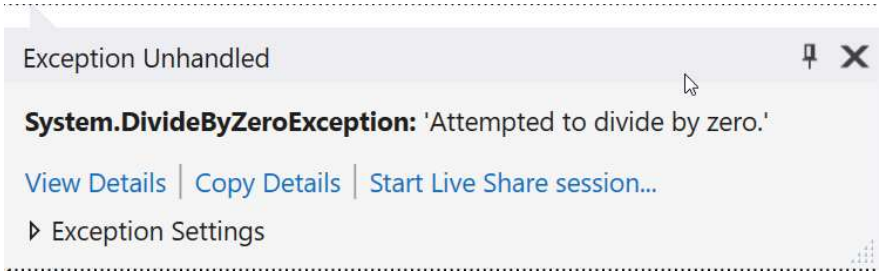
```
static int Funkcja3(
  bool warunek,
  int wartosc1,
  int wartosc2) {
  return warunek ? wartosc1 : wartosc2;
}
```




```
static void Main(string[] args) {
  var wynik = Funkcja3(true,  Funkcja1(), Funkcja2());
  Console.WriteLine(wynik);
}
```

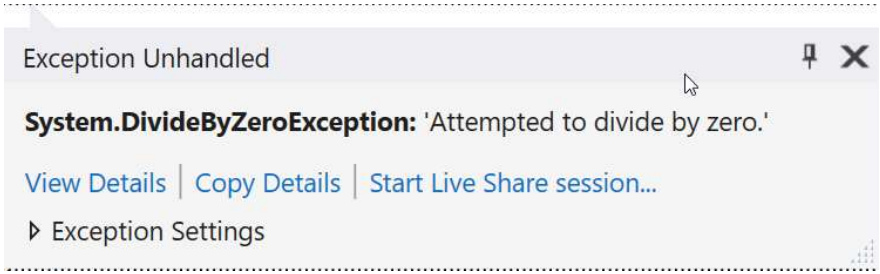
Zachłanne i leniwe obliczenia

```
static int Funkcja1() { var x=0; return 1/x };

static int Funkcja2() => 2;

static int Funkcja3(
  bool warunek,
  int wartosc1,
  int wartosc2) {
  return warunek ? wartosc1 : wartosc2;
}

static void Main(string[] args) {
  var wynik = Funkcja3(true, Funkcja1(), Funkcja2());
  Console.WriteLine(wynik);
}
```



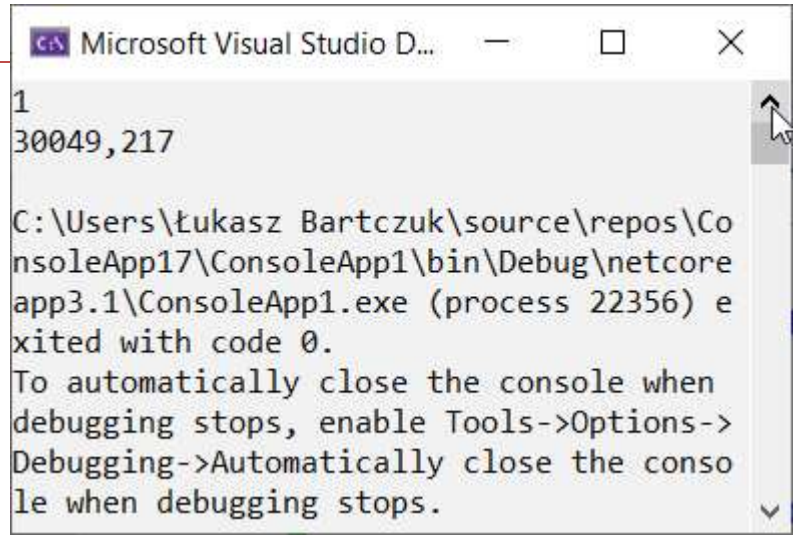
Zachłanne i leniwe obliczenia

```
static int Funkcja1() { var x=0; return 1/x };

static int Funkcja2() => 2;

static int Funkcja3(
  bool warunek,
  int wartosc1,
  int wartosc2) {
  return warunek ? wartosc1 : wartosc2;
}

static void Main(string[] args) {
  var wynik = Funkcja3(false, Funkcja1(), Funkcja2());
  Console.WriteLine(wynik);
}
```



Zachłanne i leniwe obliczenia

```
static int Funkcja1() {
  Thread.Sleep(20000); return 1;
}

static int Funkcja2() {
  Thread.Sleep(10000); return 2;
}

static void Przyklad() {
  var wynik = Funkcja3(false, Funkcja1(), Funkcja2());
  Console.WriteLine(wynik);
}

static void Main(string[] args) {
  Console.WriteLine(ZmierzCzas(Przyklad));
}
```



Obliczenia leniwe (lazy computation)

Obliczenia leniwe wykonywane są tylko wtedy, gdy są konieczne

Wartości argumentów nie są obliczane przed wywołaniem funkcji, tylko w trakcje jej działania

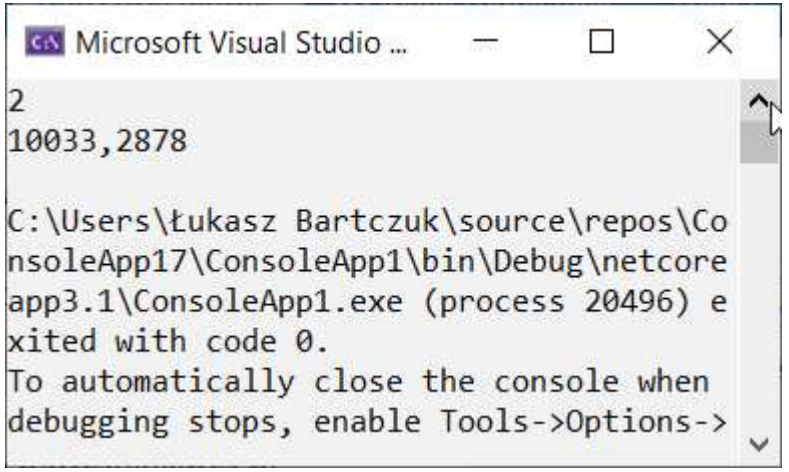
Leniwe obliczenia

Implementacja za pomocą Func<>

```
static int Funkcja4(
  bool warunek,
  Func<int> wartosc1,
  Func<int> wartosc2) {
  return warunek ? wartosc1() : wartosc2();
}
```

Obliczenia leniwe

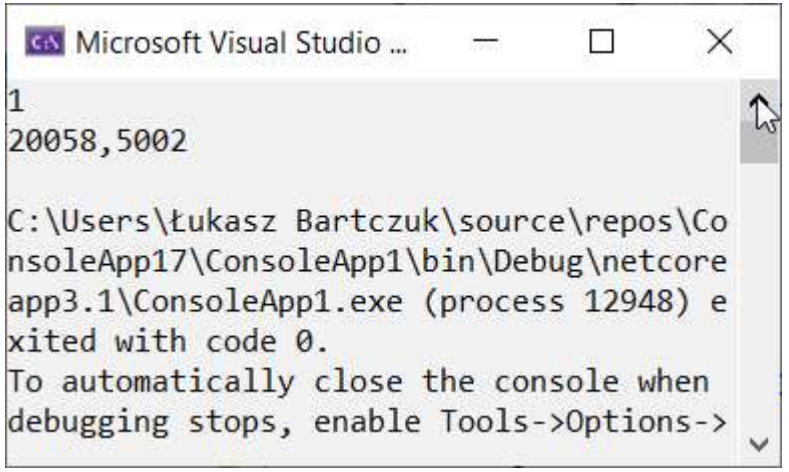
```
static void Przyklad() {
  var wynik = Funkcja4(false,
                       Funkcja1,
                       Funkcja2);
  Console.WriteLine(wynik);
}
```




```
static void Main(string[] args) {
  Console.WriteLine(ZmierzCzas(Przyklad));
}
```

Obliczenia leniwe

```
static void Przyklad() {
  var wynik = Funkcja4(true,
                       Funkcja1,
                       Funkcja2);
  Console.WriteLine(wynik);
}
```



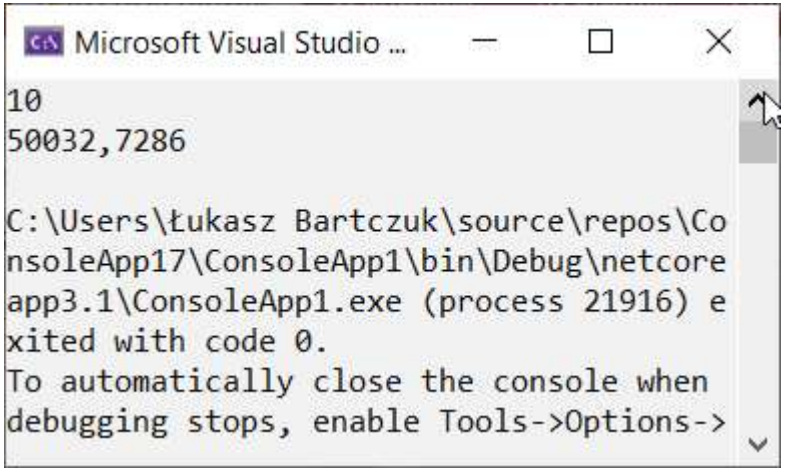

```
static void Main(string[] args) {
  Console.WriteLine(ZmierzCzas(Przyklad));
}
```

Obliczenia leniwe

```
static int Funkcja5(int liczbaPowtorzen, Func<int> wartosc) {

  int suma = 0;
  for(int i=0; i<liczbaPowtorzen; i++) {
    suma += wartosc();
  }

  return suma;

}
```

Obliczenia leniwe

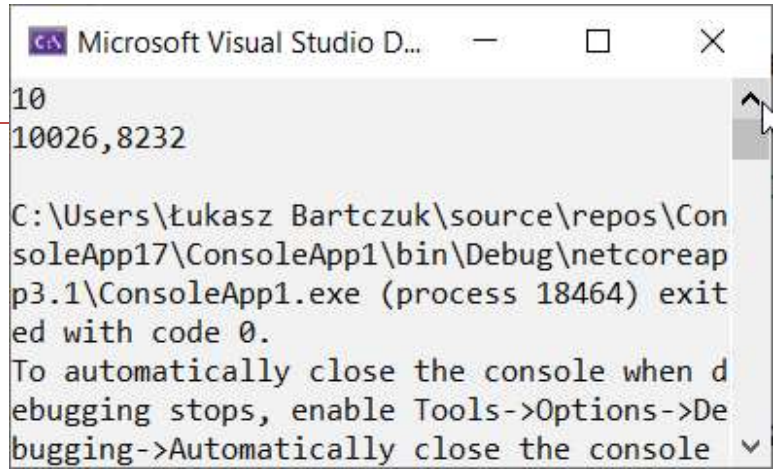
```
static void Przyklad() {
  var wynik = Funkcja5(5, Funkcja2);
  Console.WriteLine(wynik);
}

static void Main(string[] args) {
  Console.WriteLine(ZmierzCzas(Przyklad));
}
```



Obliczenia leniwe

```
static int Funkcja6(int liczbaPowtorzen, Lazy<int> wartosc) {

  int suma = 0;
  for(int i=0; i<liczbaPowtorzen; i++) {
    suma += wartość.value;
  }

  return suma;

}
```

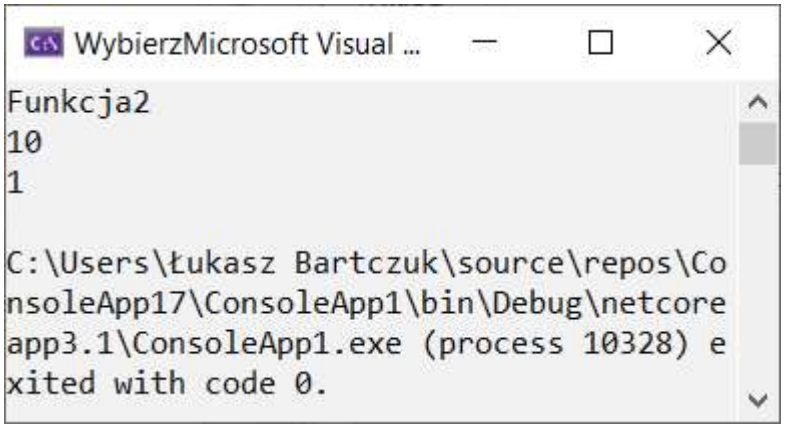
Obliczenia leniwe




```
static void Przyklad() {
  var wynik =
    Funkcja6(5, new Lazy<int>(Funkcja2));
  Console.WriteLine(wynik);
}

static void Main(string[] args) {
  Console.WriteLine(ZmierzCzas(Przyklad));
}
```

Obliczenia leniwe

```
static int liczbaWywolan;

static int Funkcja2() {
  Console.WriteLine(nameof(Funkcja2));
  liczbaWywolan++;
  return 2;
}

static void Main(string[] args) {
  Przyklad();
  Console.WriteLine(liczbaWywolan);
}
```



Zachłanna inicjalizacja obiektów

```
class Autor
{
   public int Id {get;}
   public string Imie {get;}
   public string Nazwisko {get;}
   public Ksiazki Ksiazki{get;}

   public Autor(int id, string imie, string nazwisko, Ksiazki ksiazki) {
     Id = id;
     Imie = imie;
     Nazwisko = nazwisko;
     Ksiazki = ksiazki;
   }
}
```

Zachłanna inicjalizacja obiektów

```
class RepozytoriumKsiazek {
  public Ksiazki PobierzDlaAutora(int autorId) { return null; }
}

static void Main(string[] args)
{
  var repozytoriumKsiazek = new RepozytoriumKsiazek();
  var autor = new Autor(1, "Adam", "Mickiewicz",
      repozytoriumKsiazek.PobierzDlaAutora(1));
}
```

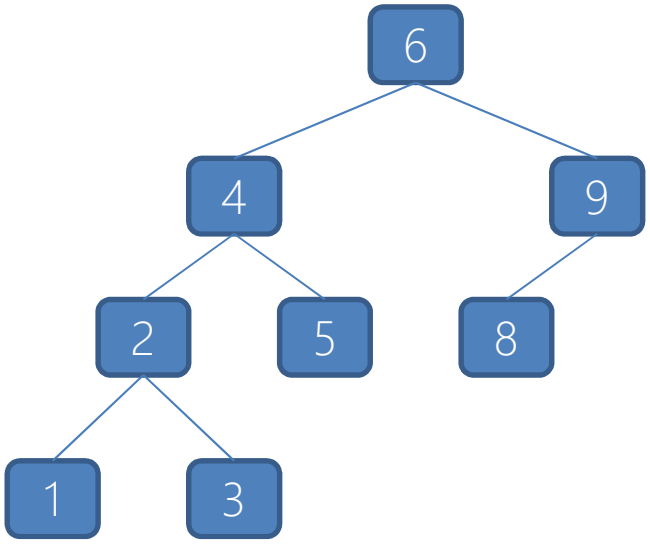
Leniwa inicjalizacja obiektów

```
class Autor
{
    private Lazy<Ksiazki> ksiazki;

    public int Id {get;}
    public string Imie {get;}
    public string Nazwisko {get;}
    public Ksiazki Ksiazki {get => ksiazki.Value}

    public Autor(int id, string imie, string nazwisko, Lazy<Ksiazki> ksiazki) {
      Id = id;
      Imie = imie;
      Nazwisko = nazwisko;
      this.ksiazki = ksiazki;
    }
}
```

Leniwa inicjalizacja obiektów

```
static void Main(string[] args) {

  var repozytoriumKsiazek = new RepozytoriumKsiazek();
  var autor = new Autor(1, "Adam", "Mickiewicz",
    new Lazy<Ksiazki>(()=>repozytoriumKsiazek.PobierzDlaAutora(1)));

}
```

Wzorzec iterator



1,2,3,4,5,6,8,9

1,2,3,4

Wzorzec iterator

Wzorzec iterator zapewnia sekwencyjny dostęp do elementów większego obiektu

- ukrycie wewnętrznej reprezentacji obiektu
- udostępnia jednolity interfejs do poruszania się po zagregowanych strukturach

IEnumerable<T>, IEnumerator<T>

```
public interface IEnumerator
{
   object Current { get; }
   bool MoveNext();
   void Reset();
}

public interface IEnumerator<out T> : IDisposable, IEnumerator
{
  T Current { get; }
}
```

IEnumerable<T>, IEnumerator<T>

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
```

Lista

```
class Lista<T>
{
   public T Wartosc { get; }
   public Lista<T> Nastepny { get; }

   public Lista(T wartosc, Lista<T> nastepny)
   {
      Wartosc = wartosc;
      Nastepny = nastepny;
   }
}
```

Lista – iterator (1)

```
class Iterator<T> : IEnumerator<T>
{
   public T Current { … }

   object IEnumerator.Current => Current;

   public bool MoveNext() { … }
   public void Reset() {…}

   public void Dispose() { }
}
```

Lista – iterator (2)

```
class Iterator<T> : IEnumerator<T>
{
   private Lista<T> obecny;
   private Lista<T> poczatek;

   public Iterator(Lista<T> lista) {
     obecny = null;
     poczatek = lista;
   }
```

Lista – iterator (3)

```
class Iterator<T> : IEnumerator<T> {
   …
   public T Current {
     get => obecny == null ? default : obecny.Wartosc;
   }

   object IEnumerator.Current => Current;

   public bool MoveNext() {
     obecny = obecny == null ? poczatek : obecny.Nastepny;
     return obecny != null;
   }

   public void Reset() { obecny = poczatek; }

   public void Dispose() { }
}
```

Lista

```
class Lista<T> : IEnumerable<T>
{
    public T Wartosc { get; }
    public Lista<T> Nastepny { get; }

    public Lista(T wartosc, Lista<T> nastepny)
    {
        Wartosc = wartosc;
        Nastepny = nastepny;
    }

    public IEnumerator<T> GetEnumerator() => new Iterator<T>(this);

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

Wykorzystanie

```
class Program {

  static void Main(string[] args) {
    var lista = Lista(1, Lista(2, Lista(3, Lista(4, null))));
    var wsk = lista.GetEnumerator();

    while (wsk.MoveNext()) {
       var el = wsk.Current;
       Console.WriteLine(el);
    }
  }

}
```

Pętla foreach

```
var wsk = lista.GetEnumerator();

while (wsk.MoveNext()) {
    var el = wsk.Current;
    Console.WriteLine(el);
}
```

```
foreach(var el in lista)
  Console.WriteLine(el);
```

Leniwe sekwencje

Sekwencje, w których kolejne elementy nie są obliczane natychmiast, ale w momencie gdy są potrzebne.

Generatory

Obiekty, które implementują jednocześnie interfejsy IEnumerator i IEnumerable

Fibonacci

```
class Fibonacci : IEnumerator<Tuple<int, int>>, IEnumerable<Tuple<int, int>>
{
   private int f_1 = 1;
   private int f_2 = 0;
   private int element = -1;

   public Tuple<int, int> Current => Tuple.Create(f_1, element);

   object IEnumerator.Current => Tuple.Create(f_1, element);

   public bool MoveNext() {
      var tmp = f_1;
      f_1 = f_2;
      f_2 = tmp + f_1;
      element++;
      return true;
   }
```

Fibonacci

```
    public void Reset() {
        f_1 = 1;
        f_2 = 0;
        element = -1;
    }

    public void Dispose() { }

    public IEnumerator<Tuple<int, int>> GetEnumerator() => this;

    IEnumerator IEnumerable.GetEnumerator() => this;
}
```

Fibonacci

```
var fib = new Fibonacci();

foreach (var f in fib) {
    Console.WriteLine($"{f.Item2} {f.Item1}");
    if (f.Item1 > 10)
        break;
}
```

Funkcje iteracyjne

```
public static IEnumerable<int> Example()
{
    yield return 1;
    yield return 2;
    yield return 3;
}
```

Funkcje iteracyjne

```
private sealed class <Example>d__3 :
  IEnumerable<int>, IEnumerable, IEnumerator<int>,
  IDisposable, IEnumerator
{
  private int <>1__state;
  private int <>2__current;
  private int <>l__initialThreadId;

  public <Example>d__3(int _param1) {
     base..ctor();
     this.<>1__state = _param1;
     this.<>l__initialThreadId = Environment.CurrentManagedThreadId;
  }
```

Funkcje iteracyjne

```
bool IEnumerator.MoveNext() {
   switch (this.<>1__state) {
      case 0:
        this.<>1__state = -1;  this.<>2__current = 1;
        this.<>1__state = 1;   return true;
      case 1:
        this.<>1__state = -1;  this.<>2__current = 2;
        this.<>1__state = 2;   return true;
      case 2:
        this.<>1__state = -1;  this.<>2__current = 3;
        this.<>1__state = 3;   return true;
      case 3:
        this.<>1__state = -1;  return false;
      default:
        return false;
   }
}
```

Funkcje iteracyjne

```
void IDisposable.Dispose() {}

int IEnumerator<int>.Current { get { return this.<>2__current; } }

void IEnumerator.Reset() { throw new NotSupportedException(); }

object IEnumerator.Current { get { return (object) this.<>2__current; } }
```

Funkcje iteracyjne - lista

```
class Lista<T> : IEnumerable<T> {
  public T Wartosc { get; }
  public Lista<T> Nastepny { get; }

  public Lista(T wartosc, Lista<T> nastepny) {
    Wartosc = wartosc;
    Nastepny = nastepny;
  }

  public IEnumerable<T> PobierzWszystkie() {
    var obecny = this;
    while (obecny != null) {
       yield return obecny.Wartosc;
       obecny = obecny.Nastepny;
    }
  }
}
```

Funkcje iteracyjne - lista

```
public IEnumerable<T> PobierzTylko2()
{
   var obecny = this;
   int i = 0;
   while (obecny != null)
   {
     if (i == 2)
        yield break;
     yield return obecny.Wartosc;
     obecny = obecny.Nastepny;
     i++;
   }
}
```

Funkcje iteracyjne

```
public static IEnumerable<Tuple<int,int>> Fib() {
   int f_1 = 0;
   int f_2 = 1;
   int element = 0;

   do {
      int tmp = f_1;
      f_1 = f_2;
      f_2 = tmp + f_2;
      yield return Tuple.Create(tmp, element++);
   }
   while (true);
}
```

METODY ROZSZERZAJĄCE

Problem

Przygotowanie metody, która pozwala z łańcucha znaków wybrać co drugi znak.

Podejście 1 – modyfikacja klasy string

```
class String {
  public string CoDrugi() {
     var sb = new StringBuilder();
     for(int i=0; i<Length; i++) {
        if(i%2 == 0)
          sb.Append(this[i]);
     }
     return sb.ToString();
  }
}
```

Podejście 2 – tworzenie klasy pochodnej

```
class MyString : String {
  public string CoDrugi() {
     var sb = new StringBuilder();
     for(int i=0; i<Length; i++) {
        if(i%2 == 0)
          sb.Append(this[i]);
     }
     return sb.ToString();
  }
}
```

Podejście 3 – tworzenie nowej klasy statycznej

```
class static MyString {
  public static string CoDrugi(string text) {
     var sb = new StringBuilder();
     for(int i=0; i< text.Length; i++) {
        if(i%2 == 0) sb.Append(text[i]);
     }
     return sb.ToString();
  }
}
```

```
string text = "Ala ma kota";
MyString.CoDrugi(text);
```

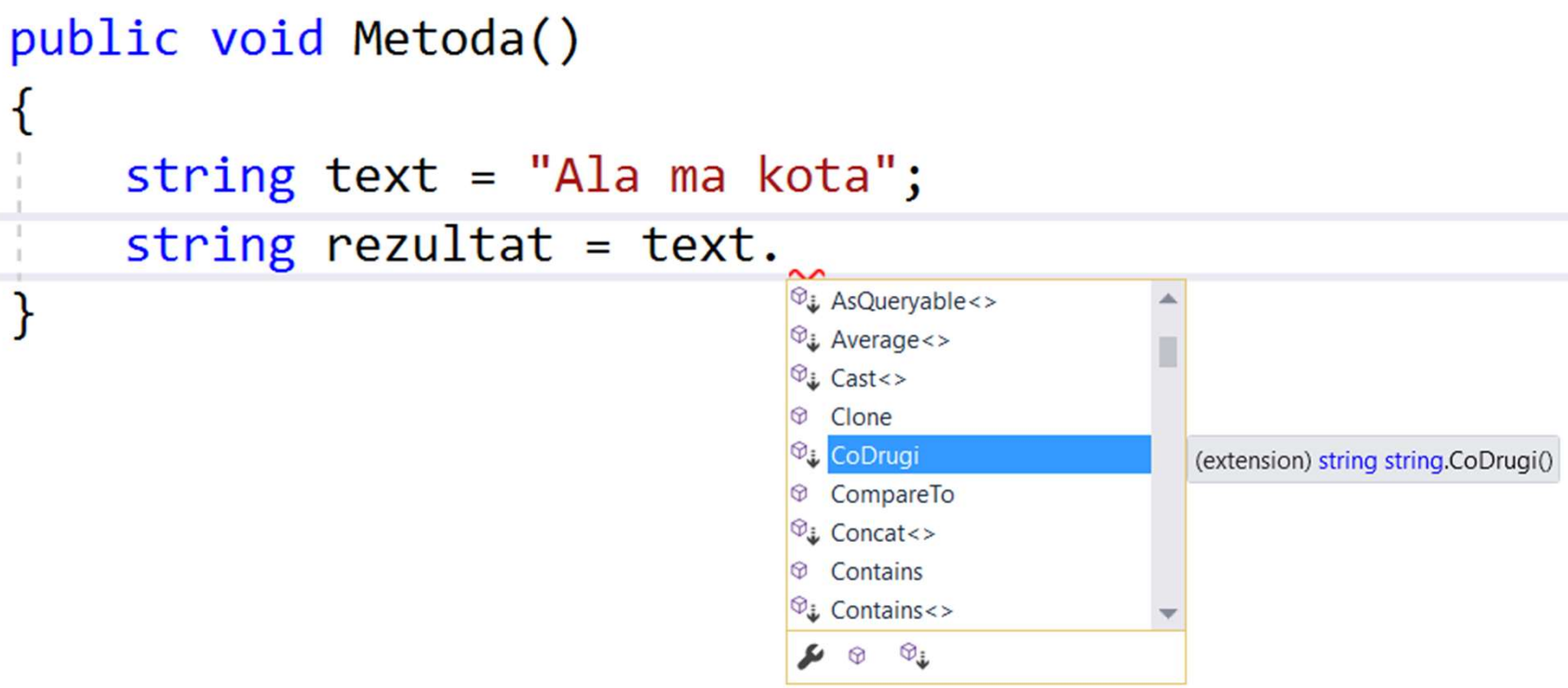
Metody rozszerzające

Metody rozszerzające pozwalają na dodanie nowych metod do już istniejących typów danych bez konieczności tworzenia nowego typu lub modyfikowania istniejącego kodu.

Podejście 4 – metody rozszerzające

```
static class Extensions {
  public static string CoDrugi(this string text) {
     var sb = new StringBuilder();
     for(int i=0; i< text.Length; i++) {
        if(i%2 == 0) sb.Append(text[i]);
     }
     return sb.ToString();
  }
}
```

```
string text = "Ala ma kota";
text.CoDrugi();
```


public void Metoda()
{
string text = "Ala ma kota";
string rezultat = text.
}
AsQueryable<>
Average<>
Cast<>
Clone
CoDrugi
CompareTo
Concat<>
Contains
Contains<>
(extension) string string.CoDrugi()

LINQ

LINQ – Language Integrated Query

Biblioteka LINQ pozwala na ujednolicony sposób odpytywania dowolnej sekwencyjnej kolekcji danych.

Składnia LINQ wzorowana jest na języku SQL.

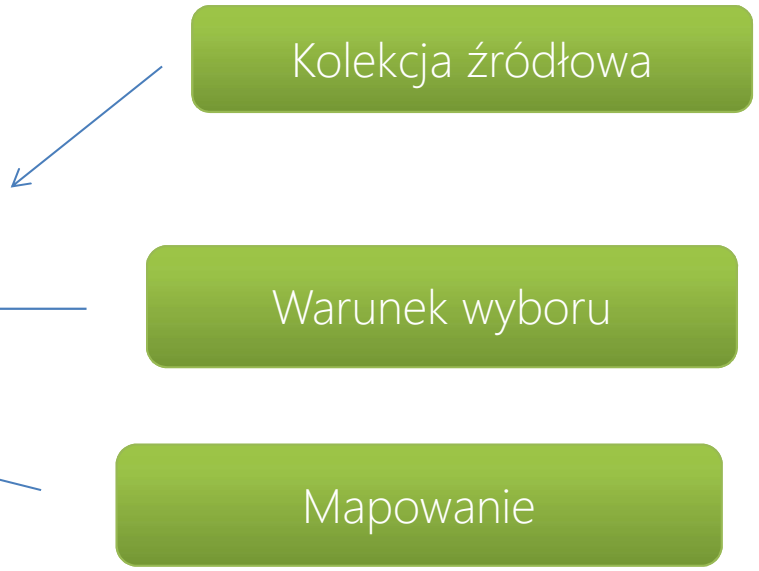
Linq to Objects

LINQ to Objects służy do odpytywania kolekcji obiektów takich jak:

- listy
- tablice
- kolejki
- ...

Zapytania

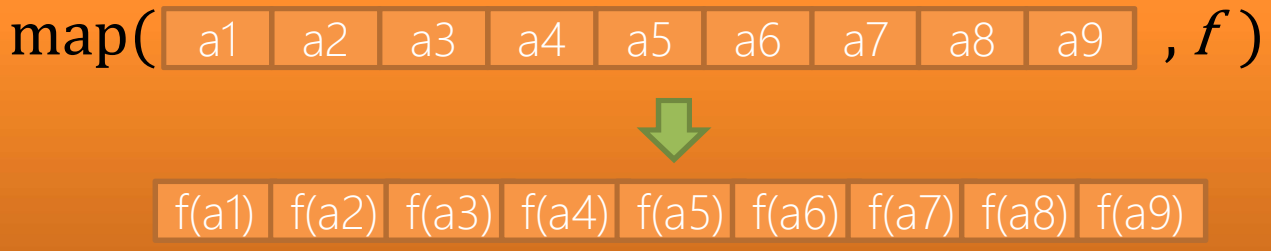
```
var rezultat =
      (from v in listaLosowa
       where v > 10
       select v * v
      ).ToList();
```



Operacje mapowania elementów

Operacja mapowania listy tworzy nową listę poprzez zastosowanie funkcji f do każdego elementu listy wejściowej

$$\text{map}: (A[], A \to B) \to B[]$$



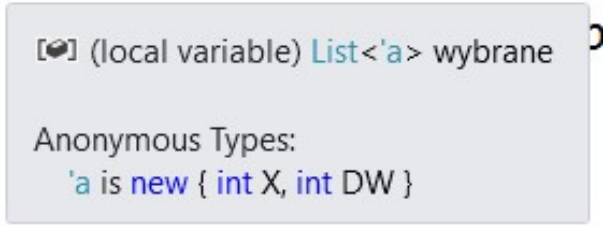
Operacje mapowania elementów

```
public static IEnumerable<TResult> Select<TSource, TResult>
     (this IEnumerable<TSource> source,
           Func<TSource, TResult> selector)
```

```
var lista =    (from v in Enumerable.Range(0, 100)
                    select rand.Next(0, 99)
                ).ToList();

var kwadraty = (from v in lista
                    select v * v
                ).ToArray();
```

Operacje mapowania elementów

```
List<Punkt> punkty = new List<Punkt> {
  new Punkt { Kolor = Color.Red, X=1, Y=1 },
  new Punkt { Kolor = Color.Blue, X=10, Y=10 },
  new Punkt { Kolor = Color.Green, X=-10, Y=-10 }
};
```




```
var wybrane = (from p in punkty
               select new { p.X, DW = p.Y }
              ).ToList();
```

Odroczone wykonanie

```
IEnumerable<int> numbers =
                Enumerable.Range(0, 10)
                     .Select(n => n * n);
```

Nie spowoduje wykonania zapytania

- ToList
- ToArray
- ToDictionary
- ToLookup

Operacja filtrowania listy

Operacja filtrowania listy tworzy nową listę poprzez wybranie tych elementów listy, które spełniają określony warunek

$$filtr\colon (A[], A \to \text{bool}) \to A[]$$

$filtr($ a1 | a2 | a3 | a4 | a5 | a6 | a7 | a8 | a9 $, p\ )$

a1 | a3 | a4 | a7 | a8 | a9

Operacja filtrowania listy

```
public static IEnumerable<TSource> Where<TSource>(
       this IEnumerable<TSource> source,
       Func<TSource, bool> predicate);
```

```
var parzyste = (from v in lista
                where v % 2 == 0
                select v
               );
```

Przykładowe dane

```
class Osoba {
  public int Id { get; init; }
  public string Imie { get; init; }
  public string Nazwisko { get; init; }
}


class Pojazd {
  public string NrRej { get; init; }
  public int IdOsoby { get; init; }
}
```

Przykładowe dane

```
var osoby = new [] {
   new Osoba {Id = 1, Imie="Ala", Nazwisko = "Kot" },
   new Osoba {Id = 2, Imie="Cezary", Nazwisko = "Adamski"},
   new Osoba {Id = 3, Imie="Franek", Nazwisko="Relke"},
   new Osoba {Id = 4, Imie="Tomasz", Nazwisko="Nowak"}
};

var pojazdy = new [] {
   new Pojazd {IdOsoby = 1, NrRej="SC12345"},
   new Pojazd {IdOsoby = 2, NrRej="WE12345"},
   new Pojazd {IdOsoby = 1, NrRej="KW12345"}
};
```

Filtrowanie kolekcji

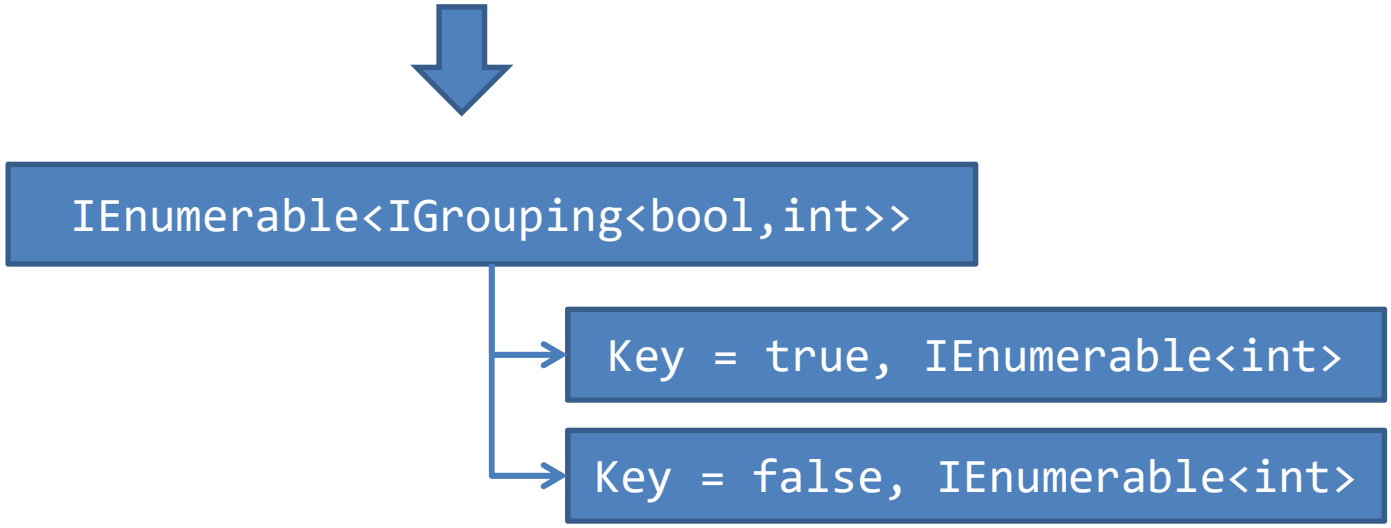
```
var pojazdyZCzwy = pojazdy
                     .Where(p => p.NrRej.StartsWith("SC"))
                     .ToList();
```

```
var pojazdyZCzwy = (from pojazd in pojazdy
                    where pojazd.NrRej.StartsWith("SC")
                    select pojazd
                   ).ToList();
```

Operacja grupowania

Operacja grupowania pozwala podzielić kolekcje na podzbiory

```
var grupy = (from v in lista
             group v by v % 2 == 0
            );
```



Operacja grupowania

```
var pojazdyOsob = (from pojazd in pojazdy
                   group pojazd by pojazd.IdOsoby
                  ).ToList();
```

```
var pojazdyOsob = pojazdy.GroupBy(g=>g.IdOsoby).ToList();
```

List<IGrouping<int,Pojazd>>

Operacja grupowania

```
var pojazdyOsob =
    (from pojazd in pojazdy
     group pojazd by pojazd.IdOsoby into grupa
     orderby grupa.Key
     select new {IDOsoby = grupa.Key, Pojazdy = grupa.ToArray()}
    ).ToList();

var pojazdyOsob = pojazdy
   .GroupBy(g=>g.IdOsoby,
            (idOsoby, pojazdy)=> new { IDOsoby = idOsoby,
                                       Pojazdy = pojazdy.ToArray()
                                     }).ToList();
```

Złączenia

```
var osobyIPojazdy =
  (from osoba in osoby
   join pojazd in pojazdy
     on osoba.Id equals pojazd.IdOsoby
   select new { osoba.Imie, osoba.Nazwisko, pojazd.NrRej }
  ).ToList();
```

```
var osobyIPojazdy =
    osoby.Join(pojazdy,
                o => o.Id,
                p => p.IdOsoby,
                (o, p) => new { o.Imie, o.Nazwisko, p.NrRej }
               ).ToList();
```

Złączenia grupujące

```
var osobyIPojazdy =
    (from osoba in osoby
     join pojazd in pojazdy on osoba.Id equals pojazd.IdOsoby
     into grupy
     select new { osoba.Imie,
                  osoba.Nazwisko,
                  pojazdy = grupy.ToList()
                }
    ).ToList();
```

Złączenia grupujące

```
var osobyIPojazdy =
    osoby.GroupJoin(pojazdy,
                    o => o.Id,
                    p => p.IdOsoby,
                    (o, ps) =>
                       new { o.Imie,
                             o.Nazwisko,
                             Pojazdy = ps.ToList()
                           }
                   ).ToList();
```

Złączenie zewnętrzne

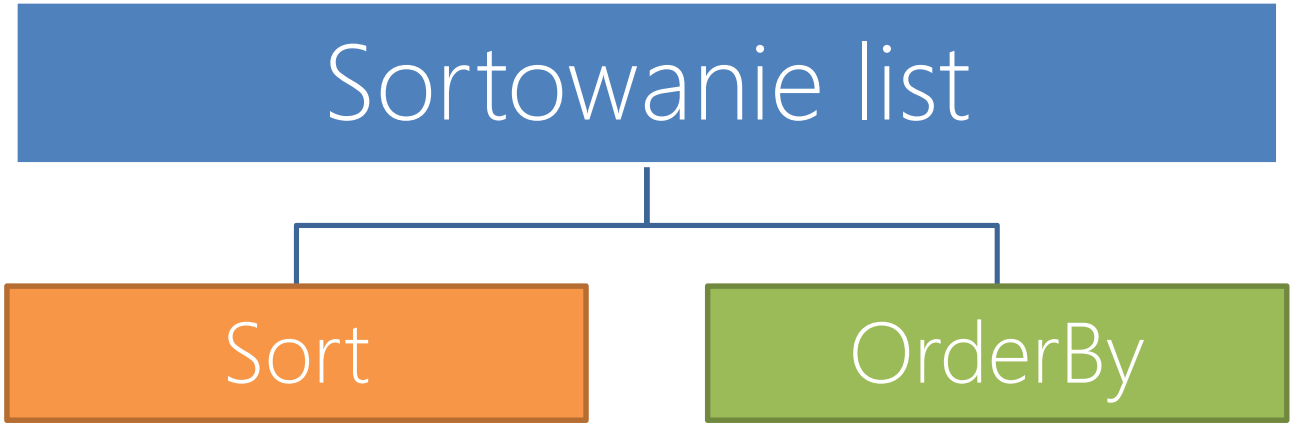
```
var osobyIPojazdy =
  (from osoba in osoby
   join pojazd in pojazdy on osoba.Id equals pojazd.IdOsoby
   into grupy
   from g in grupy.DefaultIfEmpty()
   select new { osoba.Imie,
                osoba.Nazwisko,
                NeRej = g?.NrRej ?? "Brak pojazdu"
              }).ToList();
```

Zmienne zakresu i let

Zmienne zakresu reprezentują każdy element sekwencji w kolejności

```
var wybrane = (from ksiazka in ksiazki
               let pLitera = ksiazka.Tytul[0] == 'P'
               let lStron = ksiazka.Strony > 250
               where pLitera && lStron
               select ksiazka
              ).ToList();
```

Sortowanie list




```
var listaLosowa
   = new List<int>();
…
listaLosowa.Sort();
```

```
var listaLosowa
   = new List<int>();
…
listaLosowa.OrderBy(v=>v)
           .ToList();
```

Operacja agregowania elementów (1)

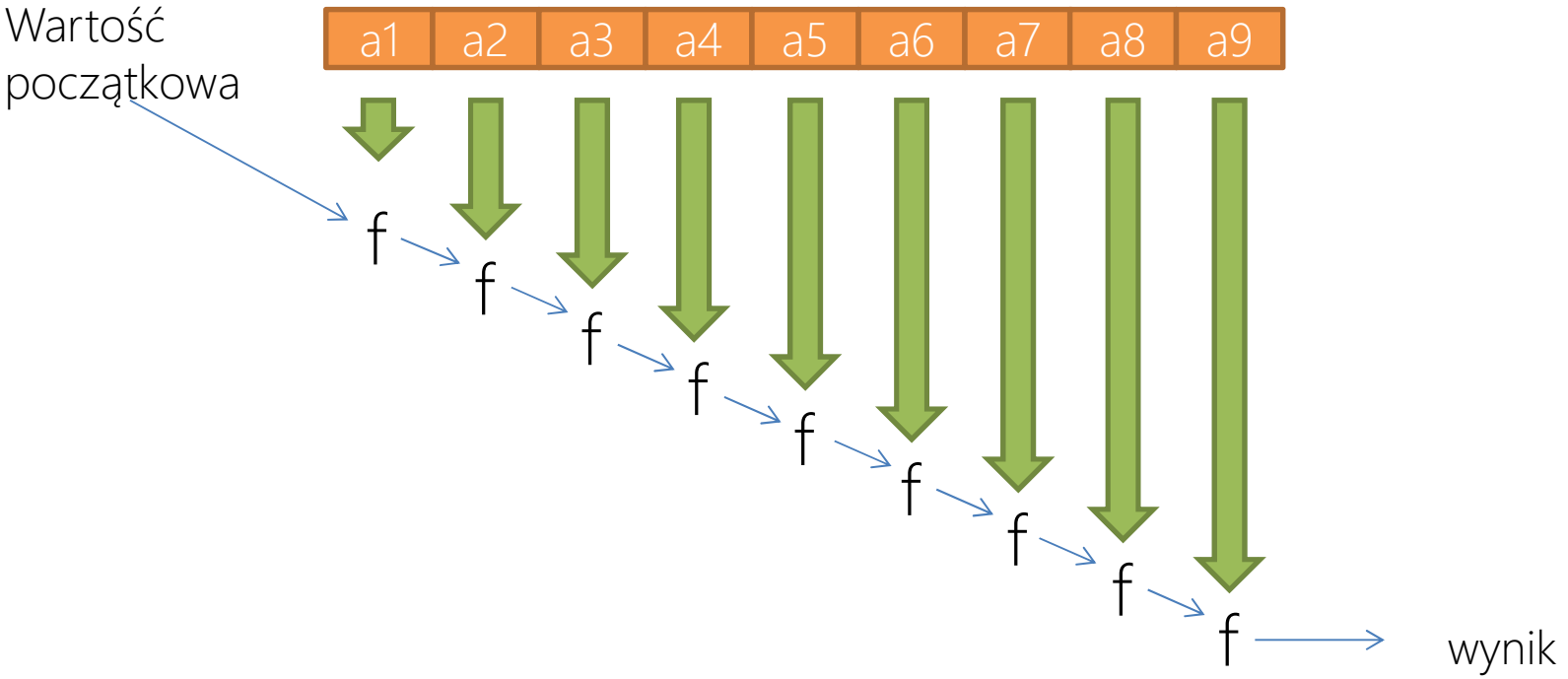
Operacja agregowania tworzy pojedyncza wartość z połączenia elementów listy

$$\mathbf{agreguj}(A[], R, (R, A) \rightarrow R) \Rightarrow R$$

Wartość początkowa

Funkcja agregująca

Operacja agregowania elementów (2)



Operacja agregowania elementów (3)

```
int sum = Enumerable.Range(0, 10)
          .Aggregate(0, (a, v)=>a+v);


listaLosowa
      .Aggregate(listaLosowa[0],
                 (a, v) => a > v ? v : a)
```

Zapytania

```
var rezultat = listaLosowa
       .Where(v => v > 10)
       .Select(v => v * v).ToList();
```

```
var rezultat =
     (from v in listaLosowa
      where v > 10
      select v * v
     ).ToList();
```

SelectMany (1)

Operator SelectMany pozwala spłaszczyć hierarchię sekwencji

```
var przedmioty = new List<string> {
   "Algebra",
   "Paradygmaty programowania",
   "Systemy Operacyjne"
};

var litery = przedmioty
             .SelectMany(p => p.ToCharArray())
             .ToList();
```

SelectMany (2)

```
var litery = przedmioty
      .SelectMany(p =>p.ToCharArray())
      .ToList();
```

```
static IEnumerable<R> SelectMany<A, R>(
          this IEnumerable<A> sequence,
          Func<A, IEnumerable<R>> function)
{
   foreach (A outerItem in sequence)
      foreach (R innerItem in function(outerItem))
          yield return innerItem;
}
```

SelectMany (3)

```
var litery = przedmioty
      .SelectMany(p =>p.ToCharArray())
      .ToList();
```

```
var litery  = (
               from przedmiot in przedmioty
               from litera in przedmiot.ToCharArray()
               select litera
              ).ToList();
```

SelectMany (4)

```
var a = new[] { 1, 4, 7 };
var b = new[] { 2, 5, 8 };
var wynik = a.SelectMany(i => b, (i, j) => i + j).ToList();
```

```
public static IEnumerable<C> SelectMany<A, B, C>(
      this IEnumerable<A> items,
      Func<A, IEnumerable<B>> function,
      Func<A, B, C> projection)
{
   foreach (A outer in items)
     foreach (B inner in function(outer))
       yield return projection(outer, inner);
}
```

SelectMany (5)

```
var a = new[] { 1, 4, 7 };
var b = new[] { 2, 5, 8 };
var wynik = a.SelectMany(i => b, (i, j) => i + j).ToList();
```

```
var wynik = (
             from v1 in a
             from v2 in b
             select v1 + v2
            ).ToList();
```

Linq to Objects - operatory

Operator	Grupa	Operator	Grupa
Aggregate	Agregacja	ElementAt	Elementy
All	Liczebność	ElementAtOrDefault	Elementy
Any	Liczebność	Empty	Tworzenie
AsEnumerable	Konwersja	Except	Zbiór
Average	Agregacja	First	Elementy
Cast	Konwersja	FirstOrDafault	Elementy
Concat	Zbiór	GroupBy	Grupowanie
Contains	Liczebność	GroupJoin	Połączenia
Count	Agregacja	Intersect	Zbiór
DefaultIfEmpty	Elementy	Join	Elementy
Distinct	Zbiór	Last	Elementy

Linq to Objects - operatory

Operator	Grupa	Operator	Grupa
LastOrDefault	Elementy	SelectMany	Projekcja
LongCount	Agregacja	SequenceEqual	Równość
Max	Agregacja	Single	Elementy
Min	Agregacja	SingleOrDefault	Elementy
OfType	Konwersja	Skip	Podział
OrderBy	Porządkowanie	SkipWhile	Podział
OrderByDescending	Porządkowanie	Sum	Agregacja
Range	Generacja	Take	Podział
Repeat	Generacja	TakeWhile	Podział
Reverse	Porządkowanie	ThenBy	Porządkowanie
Select	Projekcja	ThenByDescending	Porządkowanie

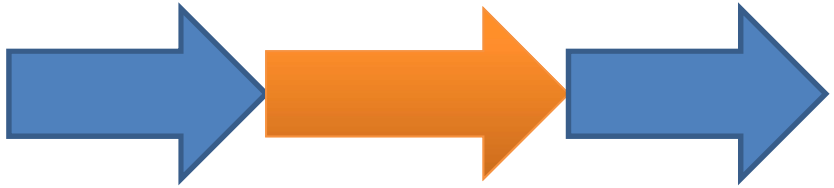
LINQ to Objects - operatory

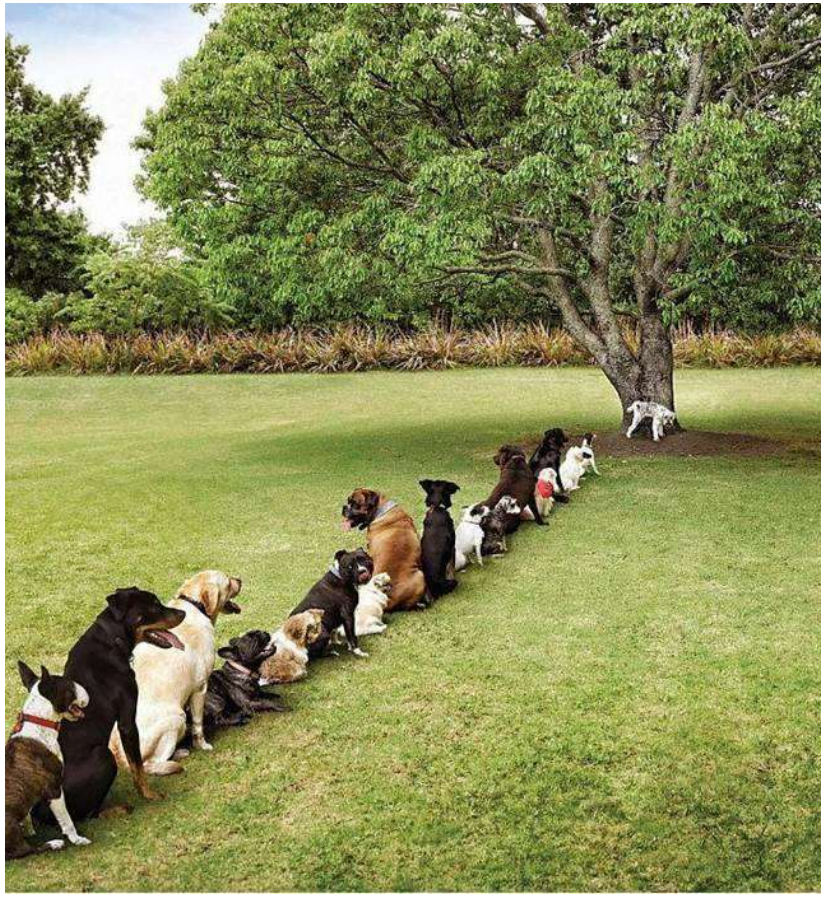
Operator	Grupa	Operator	Grupa
ToArray	Konwersja	ToLookup	Konwersja
ToDictionary	Konwersja	Union	Zbiór
ToList	Konwersja	Where	Ograniczenia

PROGRAMOWANIE ASYNCHRONICZNE

Programowanie synchroniczne

Programowanie synchroniczne polega na wykonywaniu poleceń w kolejności jedno po drugim





Programowanie synchroniczne

Przykład 1:

```
private void Komunikat(string wiadomosc) {
   MessageBox.Show($"{wiadomosc}");
}
private void button1_Click(object sender, EventArgs e)
{
    Stopwatch sw = new Stopwatch();
    Komunikat("Button1_click: Początek");
    sw.Start();
    var wynik = PobierzDaneOPogodzie("synchronicznie");
    sw.Stop();
    Komunikat($"Wynik: {wynik} po: {sw.ElapsedMilliseconds}");
    Komunikat("Button1_click: Koniec");
}
```

Programowanie synchroniczne

Przykład 1:

```
private string PobierzDaneOPogodzie(object argument)
{
    Thread.Sleep(5000);
    return "Slonecznie";
}
```

Programowanie synchroniczne

Przykład 2:

```
private string PobierzDaneOWalutach(object argument)
{
    Thread.Sleep(10000);
    return "Dolar bez zmian";
}

private string PobierzWiadomosci(object argument)
{
    Thread.Sleep(1000);
    return "Wiadomości są dobre";
}
```

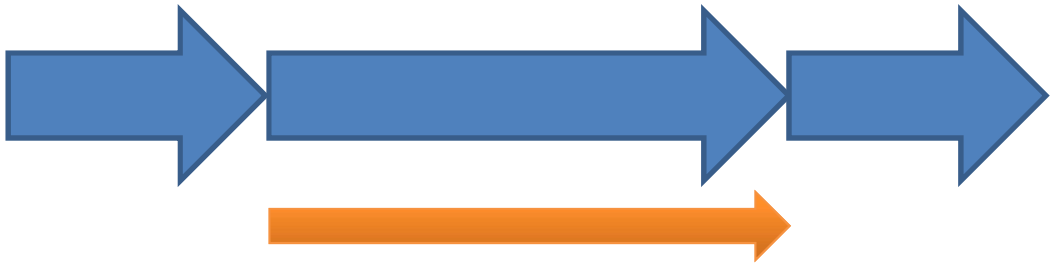
Programowanie synchroniczne

Przykład 2:

```
private void button2_Click(object sender, EventArgs e)
{
    Stopwatch sw = new Stopwatch();
    Komunikat("Button2_click: Początek");
    sw.Start();
    var wynik1 = PobierzDaneOPogodzie("synchronicznie");
    var wynik2 = PobierzDaneOWalutach("synchronicznie");
    var wynik3 = PobierzWiadomosci("synchronicznie");
    sw.Stop();
    Komunikat($"Wyniki po: {sw.ElapsedMilliseconds}");
    Komunikat("Button2_click: Koniec");
}
```

Programowanie asynchroniczne

Zadania są wykonywane niezależnie (całość nie jest blokowana), przy czym poszczególne zadania mogą być (choć nie muszą) wykonywane w różnych wątkach.



Programowanie asynchroniczne

W celu ułatwienia tworzenia aplikacji asynchronicznych w języku C# dostępne są dwie klasy

Task

Task<TResult>

Klasa Task

Klasa Task reprezentuje operację asynchroniczną, która nie zwraca wartości

Metoda	Opis
Start	Rozpoczyna wykonywanie zadania
Wait	Czeka aż wątek zakończy wykonywanie swojego działania
WaitAll(Task[])	Czeka na zakończenie wszystkich działań
WaitAny(Task[])	Czeka na zakończenie dowolnego działania
Delay(int32)	Tworzy zadanie, które zakończy się po określonym czasie

Klasa Task

Klasa Task reprezentuje operację asynchroniczną, która nie zwraca wartości

Metoda	Opis
ContinueWith(Action<Task>)	Tworzy kontynuację, która będzie uruchomiona w momencie zakończenia zadania
WhenAll(Task[])	Tworzy zadanie, które zakończy się jeżeli wszystkie zadania w tablicy się zakończą
WhenAny(Task[])	Tworzy zadanie, które zakończy się jeżeli jakiekolwiek zadanie z tablicy się zakończy

Klasa Task<Tresult>

Klasa Task<TResult> reprezentuje operację asynchroniczną, która zwraca wartość typu TResult

Posiada ona podobne metody, co klasa Task, a dodatkowo zawiera właściwość:

TResult Result

która pozwala odczytać rezultat operacji asynchronicznej

Programowanie asynchroniczne

Przykład 3:

```
private void Komunikat(string wiadomosc)
{

    string taskId = Task.CurrentId.HasValue ?
                    Task.CurrentId.ToString() : "UI";


    int threadId  = Thread.CurrentThread.ManagedThreadId;

    MessageBox.Show($"{wiadomosc} ({taskId}) {threadId}");
}
```

Programowanie asynchroniczne

Przykład 3:

```
private void button1_Click(object sender, EventArgs e) {
    Komunikat("Button1_click: Początek");

    var zadanie = new Task<string>(PobierzDaneOPogodzie, "zadanie");
    zadanie.Start();

    Komunikat("Akcja została uruchomiona");

    if (zadanie.Status != TaskStatus.Running &&
        zadanie.Status != TaskStatus.RanToCompletion)
        Komunikat("Zadanie nie zostało uruchomione");
    else
        Komunikat($"Wynik: " + zadanie.Result);
    Komunikat("Button1_click: koniec");
}
```

Programowanie asynchroniczne

Przykład 4:

```
private void button2_Click(object sender, EventArgs e) {
    Komunikat("Button2_click: Początek");
    Task<string> zadanie =
        new Task<string>(PobierzDaneOPogodzie, "zadanie");

    zadanie.ContinueWith(poprzednieZadanie => {
        Komunikat($"Wynik: " + poprzednieZadanie.Result);
    });

    zadanie.Start();
    Komunikat("Button3_click: koniec");
}
```

Programowanie synchroniczne

Przykład 5:

```
private void Kontynuacja(Task<string> zadanie) {
    Komunikat($"Wynik: " + zadanie.Result);
}

private void button3_Click(object sender, EventArgs e) {
    Komunikat("Button3_click: Początek");
    Task<string> zadanie =
          new Task<string>(PobierzDaneOPogodzie, "zadanie");

    zadanie.ContinueWith(Kontynuacja);
    zadanie.Start();
}
```

Programowanie asynchroniczne

Przykład 6:

```
private Task<string> OperacjaAsync() {
    Task<string> zadanie =
        new Task<string>(PobierzDaneOPogodzie, "zadanie");

    zadanie.Start();
    return zadanie;
}
private void button4_Click(object sender, EventArgs e) {
    Komunikat("Button4_click: Początek");
    var zadanie = OperacjaAsync();

    zadanie.ContinueWith(Kontynuacja);
}
```

async/await

async / await są operatorami języka C#, które ułatwiają tworzenie operacji asynchronicznych.

Dzięki nim możliwe jest tworzenie programów asynchronicznych w sposób jak najbardziej zbliżony do programów synchronicznych

async/await

Przykład 7:

```
private async void button5_Click(object sender, EventArgs e)
{
    Komunikat("Button5_click: Początek");
    string wynik = await OperacjaAsync();
    Komunikat($"Wynik: " + wynik);
    Komunikat("Button5_click: koniec");
}
```

programowanie współbieżne

W programowaniu współbieżnym kilka zadań jest wykonywanych równocześnie.

programowanie współbieżne

```
private Task<string> PobierzDaneOPogodzieAsync() {
  var zadanie = new Task<string>(PobierzDaneOPogodzie, "Pogoda");
  zadanie.Start();
  return zadanie;
}
private Task<string> PobierzDaneOWalutachAsync() {
  var zadanie = new Task<string>(PobierzDaneOWalutach, "Waluty");
  zadanie.Start();
  return zadanie;
}
private Task<string> PobierzDaneWiadomosciAsync() {
  var zadanie = new Task<string>(PobierzWiadomosci, "Wiadomości");
  zadanie.Start();
  return zadanie;
}
```

programowanie współbieżne

```
private void button6_Click(object sender, EventArgs e) {

  Komunikat("button6_click: Poczatek");
  Stopwatch sw = new Stopwatch();
  sw.Start();
  var zadanie1 = PobierzDaneOWalutachAsync();
  var zadanie2 = PobierzDaneOPogodzieAsync();
  var zadanie3 = PobierzDaneWiadomosciAsync();

  Task.WaitAll(zadanie1, zadanie2, zadanie3);
  sw.Stop();
  Komunikat("button6_click: Koniec " + sw.ElapsedMilliseconds);

}
```

programowanie współbieżne

```
private Task Zadanie()
{
    var zadanie =  new Task(() =>
    {
        var zadanie1 = PobierzDaneOWalutachAsync();
        var zadanie2 = PobierzDaneOPogodzieAsync();
        var zadanie3 = PobierzDaneWiadomosciAsync();

        Task.WaitAll(zadanie1, zadanie2, zadanie3);
    });

    zadanie.Start();
    return zadanie;
}
```

programowanie współbieżne

```
private Task Zadanie()
{
    var zadanie1 = PobierzDaneOWalutachAsync();
    var zadanie2 = PobierzDaneOPogodzieAsync();
    var zadanie3 = PobierzDaneWiadomosciAsync();
    return Task.WhenAll(zadanie1, zadanie2, zadanie3);
}
```

programowanie współbieżne

```
private async void button7_Click(object sender, EventArgs e)
{
    Komunikat("button7_click: Poczatek");
    Stopwatch sw = new Stopwatch();
    sw.Start();
    await Zadanie();
    sw.Stop();
    Komunikat("button7_click: Koniec "+sw.ElapsedMilliseconds);
}
```

programowanie współbieżne ze stanem

```
private void Wykonaj(int od, int @do) {
   for (int i = od; i <= @do; i++) wynik++;
   System.Threading.Thread.Sleep(1000);
}
private void button8_Click(object sender, EventArgs e) {
   wynik = 0;
   Stopwatch sw = new Stopwatch();
   sw.Start();
   Wykonaj(     1,  250000);
   Wykonaj(250001,  500000);
   Wykonaj(500001,  750000);
   Wykonaj(750001, 1000000);
   sw.Stop();
   WynikLabel.Text = wynik.ToString();
   CzasLabel.Text = sw.ElapsedMilliseconds.ToString(); }
```

programowanie współbieżne ze stanem

```
private int wynik;
private void Wykonaj(int od, int @do) {
   for (int i = od; i <= @do; i++) wynik++;
   System.Threading.Thread.Sleep(1000);
}
private void button8_Click(object sender, EventArgs e) {
   wynik = 0;
   Stopwatch sw = new Stopwatch();
   sw.Start();
   Wykonaj(     1,  250000);
   Wykonaj(250001,  500000);
   Wykonaj(500001,  750000);
   Wykonaj(750001, 1000000);
   sw.Stop();
   WynikLabel.Text = wynik.ToString();
   CzasLabel.Text = sw.ElapsedMilliseconds.ToString(); }
```

programowanie współbieżne ze stanem

```
private void button9_Click(object sender, EventArgs e) {
   wynik = 0;
   Stopwatch sw = new Stopwatch();
   Task[] t = new[] { new Task(()=>Wykonaj(     1,  250000)),
                      new Task(()=>Wykonaj(250001,  500000)),
                      new Task(()=>Wykonaj(500001,  750000)),
                      new Task(()=>Wykonaj(750001, 1000000))};
   sw.Start();
   for (int i = 0; i < t.Length; i++) t[i].Start();
   Task.WaitAll(t);
   sw.Stop();
   label1.Text = wynik.ToString();
   label2.Text = sw.ElapsedMilliseconds.ToString();
}
```

programowanie współbieżne ze stanem

```
object locker = new object();

void Wykonaj(int od, int @do)
{
   for (int i = od; i <= @do; i++)
     lock (locker)
     {
        wynik++;
     }
   System.Threading.Thread.Sleep(1000);
}
```

programowanie współbieżne bez stanu

```
int WykonajBezStanu(int od, int @do) {
    int wynik = 0;
    for (int i = od; i <= @do; i++) wynik++;
    System.Threading.Thread.Sleep(1000);
    return wynik;
}
```

programowanie współbieżne bez stanu

```
private void button11_Click(object sender, EventArgs e) {
  Stopwatch sw = new Stopwatch();
  var ts = new[] {new Task<int>(()=>WykonajBezStanu(     1,  250000)),
                  new Task<int>(()=>WykonajBezStanu(250001,  500000)),
                  new Task<int>(()=>WykonajBezStanu(500001,  750000)),
                  new Task<int>(()=>WykonajBezStanu(750001, 1000000))};
  sw.Start();
  for (int i = 0; i < ts.Length; i++) ts[i].Start();
  Task.WaitAll(ts);

  wynik = ts.Sum(t=>t.Result);

  sw.Stop();
  label1.Text = wynik.ToString();
  label2.Text = sw.ElapsedMilliseconds.ToString(); }
```

Paradykmioty, zbiur ścionk i innyh takich

1. **W pętli for(char i=0; i<256; i++) cout<<i<<endl; instrukcja cout wykona się :**
   więcej niż 257 razy (char z licznikiem daje over 9000)

2. **Funkcja rozszerzająca, jest to funkcja**
   statyczna

3. **Wartości grupowane są w n-tki w:**
   produktach kartezjańskich

4. **Które z poniższych wyrażeń poprawnie definiuje w języku C# wyrażenie**
   lambda (x,y) => (x y)

5. **Blok programu jest to konstrukcja**
   Ograniczająca zakres widoczności deklaracji wewnątrz niego

6. **Która, z poniższych instrukcji deklaruje zmienne sekwencyjnie**
   (let* ((x 2) (a 3)) (+x a))

7. **W językach o dynamicznym zakresie widoczności nazw**
   Funkcja jest wywoływana w kontekście środowiska w którym została wywołana

8. **Deklaracje są to instrukcje, które**
   tworzą nowe wiązania

9. **Instrukcja typedef języka C++ powoduje**
   przypisanie nowej nazwy do istniejącego typu

10. **C++ jest językiem programowania, w którym system typów:**
    jest statyczny

1. **W notacji prefiksowej (wyłącznie) polecenia zapisuje się w języku:**
   Scheme???

2. **Która deklaracja interfejsu w języku C# jest poprawna:**
   a. **interface MojInterfejs<in T> { void Funkcja(T x); }**

3. **Po wykonaniu programu Task t = new Task(() => {Console.Write("A");}); t.Start(); t.Wait();**
   Console.Write("B"); t.ContinueWith(c=>{Console.Write("C");}); na ekranie pojawi się:
   ABC

4. **W metodzie języka C#**
   async void Metoda() {
   int x = await Metoda2();
   Console.WriteLine("Komunikat");
   }
   napis "komunikat" wyświetli się po zakończeniu zadania zwróconego przez metodę Metoda2

5. **Która metoda języka C# jest funkcją wyższych rzędów:**
   a) **Func<double, double> F(int f) {}**

10. W pętli for(char i=0; i<256; i++) cout<<i<<endl;
instrukcja cout wykona się
więcej niż 257 razy (char => 255 => 100000000 przepełnienie!)

11. Funkcja rozszerzająca, jest to funkcja:
statyczna

12. Wartości grupowane są w n-tki w:
a) **produktach kartezjańskich**

13. Które z poniższych wyrażeń poprawnie definiuje w języku C# wyrażenie lambda λ x. λy.(x y)
(x,y) => (x y)

14. <u>Blok programu</u> jest to konstrukcja
Ograniczająca zakres widoczności deklaracji wewnątrz niego

15. Która, z poniższych instrukcji deklaruje zmienne sekwencyjnie
(let* ((x 2) (a 3)) (+x a))

16. W językach o <u>dynamicznym zakresie widoczności nazw:</u>
Funkcja jest wywoływana w kontekście środowiska w którym została wywołana

17. Deklaracje są to instrukcje, które

tworzą nowe wiązania

18. Instrukcja typedef języka C++ powoduje

a) przypisanie nowej nazwy do istniejącego typu

19. Wyrażenie języka Scheme: '((+ 1 2) (- 2 3) (-3 4))

Listę list symboli +, 1, 2, (); -, 2, 3, (); -, 3, 4, ()

20. C++ jest językiem programowania, w którym system typów:

jest statyczny

21. Które wyrażenie rachunku lambda (wzbogaconego instrukcję iloczynu) będzie miało to samo znaczenie, co instrukcja języka Scheme: (let ((x 2) (y 3)) (* x y)).

λx.λy.(* x y) 2 3

22. Wyrażenie ((lambda (z) (lambda (x y) (+ x y z))) 1) zwraca:

a) Funkcję (procedura w Scheme)

23. Zakresem widoczności zmiennych w wyrażeniu (let ((x 2) (y 3)) (+ x y)) jest:

wyrażenie (+ x y)

24. Smalltalk jest językiem należącym do paradygmatu:

obiektowego

25. Wyrażenie języka C#: x=>x+1 tworzy:

Funkcję anonimową (lambda)

26. Językiem o dynamicznym systemie typów nie jest:

C

27. Do zwrócenie pierwszego elementu pary w języku Scheme służy funkcja:

car

28. Który program języka PROLOG pozwoli poprawnie obliczyć wartość silni:

silnia(0,W) :- W is 1. silnia(N,W) :- X is N-1, silnia(X, W_N_1), W is N*W_N_1. (, ← koniunkcja (AND) :- ← implikacja ; ← alternatywa (OR))

29. Komenda mezczyzna(jan)? języka PROLOG definiuje

ma błąd składniowy

30. Wyrażenie let* języka Scheme definiuje zmienne

sekwencyjnie

31. Wyrażenie warunkowe w rachunku lambda zapisuje się jako:

λstx.stx

$$\begin{aligned}
&\underline{\textbf{true}} && \equiv \lambda xy.x \\
&\underline{\textbf{false}} && \equiv \lambda xy.y \\
&\underline{\textbf{if}} && \equiv \lambda bxy.bxy \\
&\underline{\textbf{not}} && \equiv \lambda b.\underline{\textbf{if}}\ b\ \underline{\textbf{false}}\ \underline{\textbf{true}} \\
&\underline{\textbf{and}} && \equiv \lambda b_1 b_2.\underline{\textbf{if}}\ b_1\ b_2\ \underline{\textbf{false}} \\
&\underline{\textbf{or}} && \equiv \lambda b_1 b_2.\underline{\textbf{if}}\ b_1\ \underline{\textbf{true}}\ b_2
\end{aligned}$$

32. Wyrażenie let języka Scheme definiuje zmienne

a) równolegle

33. Przyjmując, że typ int zajmuje w pamięci 4 bajty, a typ char 1 bajt to zmienna typu union Przyklad { int x; char y; }; będzie zajmować:

4 bajty

34. Wyrażenie języka C#: x=>x+2 nazywane jest:

wyrażeniem lambda

35. Fakty w PROLOGU:

są zawsze prawdziwe

36. Słówko kluczowe var w C# pozwala na

zdefiniowanie zmiennej dowolnego typu, który nie może być później zmieniany

37. Zakresem widoczności zmiennych w wyrażeniu (let* ((x 2) (y 3)) (+ x y)) jest

wyrażenie (y 3) i (+ x y)

38. Który z poniższych języków programowania jest językiem programowania obiektowego:

C#

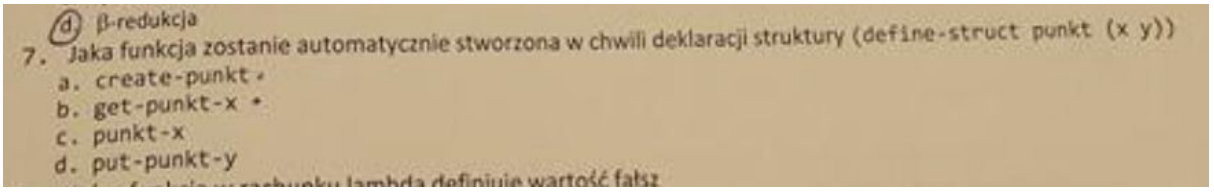
39. Postać normalna jest to

a) Wyrażenie, którego nie można dalej redukować

40. Alfa konwersja polega na zmianie

nazw zmiennych związanych

41. Jaka funkcja zostanie automatycznie stworzona w chwili deklaracji struktury (define-strukt punkt (x y)): (raczej put-punkt bo w ponizszym zadaniu nie ma nawet takiej odpowiedzi jak make-punkt)


d β-redukcja
7. Jaka funkcja zostanie automatycznie stworzona w chwili deklaracji struktury (define-struct punkt (x y))
a. create-punkt
b. get-punkt-x
c. punkt-x
d. put-punkt-y

make-punkt put-punkt???

42. Która z poniższych instrukcji poprawnie deklaruje funkcję rozszerzającą:
public static string Funkcja(this string str) {…} (Funkcja rozszerzająca ma this w argumencie – to co rozszerza!)

43. Która z poniższych instrukcji jest zawsze poprawna:

(letrec ((foo (lambda (x) (bar x))) (bar (lambda (x) (+ 1 x)))) (foo 5))

44. Wynikiem operacji (nfx.f(n f x) st.ssst) będzie

λfx.ffffx

45. W rekordzie aktywacji funkcji przechowywane są:

zmienne lokalne

46. Do zwrócenie drugiego elementu pary w języku Scheme służy funkcja

a) cdr

47. Przy przekazywaniu przez wartość argumentów "z" funkcji parametrem aktualnym może być:

dowolnie

48. Zakresem widoczności zmiennych w wyrażeniu (letrec ((x 2) (y 3)) (+ x y)) jest:

całe wyrażenie (let ((x 2) (y 3)) (+ x y))

49. Dowolna zmienna przechowywana na stercie jest (na stercie są wskaźniki!)

zawsze anonimowa

50. Wyrażenie letrec języka Scheme definiuje zmienne

rekurencyjnie

51. Zmienna związana jest to

zmienna, która występuje w ciele funkcji w której jest zdefiniowana

52. Która, z poniższych instrukcji występuje tylko w paradygmacie imperatywnym i pochodnych

a) pętli

53. Środowisko odniesienia jest to:

zbiór dostępnych wiązań w danym miejscu programu

54. Programowanie generyczne pozwala na

napisanie jednego kodu dla różnych typów danych

55. Który z poniższych języków programowania jest językiem programowania obiektowego?

C#

56. Z rachunku lambda zapis: (M N) oznacza:

aplikacje

57. Która z poniższych komend definiuje zmienną typu prostego w języku C++:

a) Obiekt *obj;

58. C++ jest językiem programowania w którym:
system typów jest statyczny

59. Środowisko odniesienia jest to:
zbiór wiązań dostępnych w danym miejscu programu

60. W typach będących typami rekurencyjnymi:
typ ten zawiera sam siebie

61. Rekord aktywacji funkcji przechowywany jest na:
stosie

62. Które wyrażenie rachunku lambda (wzbogaconego instrukcję iloczynu) będzie miało to samo znaczenie, co instrukcja języka Scheme: (let ((x 2) (y 3)) (* x y))
λx.λy.(* x y) 2 3

63. Wyrażenie ((lambda (z) (lambda (x y) (+ x y z))) 1) zwraca
Funkcję

64. Zakresem widoczności zmiennych w wyrażeniu (let ((x 2) (y 3)) (+ x y)) jest
wyrażenie (+ x y)

65. Smalltalk jest językiem należącym do paradygmatu
obiektowego

66. Wyrażenie języka C#: x=>x+1 tworzy:
Funkcję anonimową

67. Językiem o statycznym systemie typów jest:
C ,ADA, C++, C#, Java, Pascal, Visual Basic

68. Przyjmując, że typ int zajmuje w pamięci 4 bajty, a typ char 1 bajt to zmienna typu union Przykład { int x; char y; }; będzie zajmować
4 bajty

69. Wyrażenie języka C#: x=>x+2 nazywane jest
wyrażeniem lambda

70. Słówko kluczowe var w C# pozwala na
zdefiniowanie zmiennej dowolnego typu, która nie może być później zmieniona

71. Zakresem widoczności zmiennych w wyrażeniu (let* ((x 2) (y 3)) (+ x y)) jest

całe wyrażenie (let ((x 2) (y 3)) (+ x y))

72. Postać normalna jest to

Wyrażenie, którego nie można dalej redukować

73. Alfa konwersja polega na zmianie

nazw zmiennych związanych

74. Jaka funkcja zostanie automatycznie stworzona w chwili deklaracji struktury (define-strukt punkt (x y))

make-punkt

75. Która z poniższych instrukcji poprawnie deklaruje funkcję rozszerzającą:

public static string Funkcja(this string str) {…}

76. W rekordzie aktywacji funkcji przechowywane są

zmienne lokalne

77. Przy przekazywaniu przez wartość argumentów "z" funkcji parametrem aktualnym może być:

zmienna

78. Zmienna związana jest to

zmienna, która występuje w ciele funkcji w której jest zdefiniowana

79. Program

```
#include <iostream>
using namespace std;
template <typename T>T max(T x, T y) { return (x > y)? x : y;}
int main() {
cout << max(3, 7) << std::endl;
cout << max(3.0, 7.0) << std::endl;
cout << max(3, 7.0) << std::endl;
return 0;
}
```

da w rezultacie

Żadne z powyższych/Błąd kompilacji(niejednoznaczność 3 wywołania)

80. Która, z poniższych instrukcji występuje tylko w paradygmacie imperatywnym i pochodnych

pętli

81. Programowanie generyczne pozwala na

napisanie jednego kodu dla różnych typów danych

82. Środowisko odniesienia jest to:

zbiór dostępnych wiązań w danym miejscu programu

83. Deklaracje są to instrukcje, które

Tworzą nowe wiązania

84. Które zdanie jest prawdziwe:

Rachunek Lambda jest podstawą teoretyczną języków funkcyjnych

85. Tablice wielowymiarowe w języku C# (tworzone z pomocą instrukcji np.: var tab = new int[3,5]) są tablicami:

prostokątnymi

86. Które z poniższych wyrażeń jest poprawnym wyrażeniem Linq:

(from x in lista where x > 10 select x)

87. Które z poniższych instrukcji poprawnie deklaruje funkcje szablonowe w języku C++:

template <typename T> T dodaj (T a, T b);

88. Funkcje wyższych rzędów to funkcje które:

mają co najmniej jeden parametr, którego wartość musi być funkcją lub funkcje zwracają

89. Efekty uboczne w językach programowania:
utrudniają analizę działania programu

90. Czas życia wiązania:
a) jest zawsze krótszy niż czas życia elementu, na który pokazuje

91. Przykładem mapowania jest:
funkcja

92. W dynamicznych systemach typów typ jest przypisywany do:
a) wartości

93. Deklaracja class MojaKlasa<T> where T: new() {} oznacza, że:
Typ T musi mieć konstruktor domyślny

94. W języku C# wyrażenie x=>x+1:
nazywa się wyrażeniem lambda

95. Modyfikowalny stan obiektu w języku C#:
ułatwia tworzenie funkcji asynchronicznych

96. Która deklaracja jest inwariantna?

a) **interface MojInterfejs<T> { T Funkcja (T x); }**

97. Która metoda języka C# jest metodą rozszerzającą?

static class Klasa { public static void F(this, double x) {} }

98. Niech dana będzie funkcja int F(int a, int 0) { return a<b?a:b; }. W wyniku zastosowania na liście liczb całkowitych operacji: 1. Aggregate(l[0], F), gdzie l jest typu List<int> uzyskamy:

najmniejszą wartość na liście

99. Metoda rozszerzająca może być zdefiniowana w klasie:

statycznej

100. Wynikiem działania (λnfx.f(n f x) λfx.x) będzie wyrażenie:

a) **λfx.ffx**
b) **λfx.x**
c) **λfx.nfx**
d) **λfx.fx**

101. Możliwość modyfikowania wartości zmiennych w języku C#:

utrudnia tworzenie funkcji asynchronicznych???

102. Czyste funkcje są to funkcje, które:

nie zmieniają stanu aplikacji???

103. Jaki jest związek pomiędzy Fluent API, a Domain Specific Language?

można wykorzystać DSL do stworzenia Fluent API

104. Deklaracja języka C# interface IInterfejs<in T> {}:

ustanawia nowy interfejs kontrawariatny

105. Podstawowa operacja jaką można wykonać w rachunku lambda w celu wyznaczenia rozwiązania nazywa się

<beta>-redukcja???

106. Która funkcja w rachunku lambda definiuje wartość fałsz:

a) **λxy.y???**

107. Polecenie (define (f x y) (lambda (z) (+ x y z))) definiuje:

funkcję wyższych rzędów???

108. W języku C# inwariantne zmienne typu mogą być wykorzystane:

do określenia typu rezultatu funkcji i/lub typu parametrów

109. Jeżeli chcemy w języku C# zdefiniować kilka zadań asynchronicznych i wymagamy, aby program wykonywał się dalej po zakończeniu ich wszystkich to musimy skorzystać z funkcji:

Task.WaitAll

110. Które z wyrażeń języka C# tworzy wyrażenie lambda:

x=>x+1;

111. Ze wskaźników w języku C# można korzystać tylko w funkcji oznaczonej słówkiem kluczowym

unsafe

112. W celu zadeklarowania funkcji rekurencyjnej w języku F# nalezy zastosować instrukcję

let rec

113. Dopasowywanie wzorców w języku F# jest realizowane poprzez instrukcję

match … with

114. W języku C# instrukcja g(12,23)+f(12,23)==f(12,23)+g(12,23)

zwróci wartość true lub false w zależności od postaci funkcji f i g

115. Instrukcja języka F#

```
type Operator =
        | Plus
        | Minus
```

tworzy

Unię z dyskryminatorem

116. Funkcje czyste są to funkcje, które

Nie mają efektów ubocznych, a ich wartość zależy tylko od wartości ich parametrów

117. Złożenie funkcji w języku F# jest realizowane poprzez operator

>>

118. W rekurencji ogonowej
Wywołanie rekurencyjne musi być ostatnim wywołaniem funkcji

119. W celu stworzenia nowego iteratora w języku C# należy stworzyć klasę która będzie implementowała interfejsy
IEnumerator lub IEnumerator<out T>

120. Które polecenie języka F# tworzy funkcję wyższych rzędów
let f = fun x y -> fun a -> x+y+a;;

121. Listy w języku F# są strukturą
tylko do odczytu

122. Który z poniższych elementów nie jest obowiązkowy dla zmiennych
rozmiar

123. Wartościami niemodyfikowalnymi nazywamy takie wartości złożone, które
mogą być inicjowane tylko w konstruktorze

124. Która deklaracja interfejsu w języku C# jest poprawna
interface MojInterfejs<in T> {void Funkcja(T x);}

Gdy jest <OUT T> to funkcja w interfejsie może tylko zwracać wartość T, ale nie przyjmować jej jako parametru tak jak wyżej

125. Która instrukcja języka C# poprawnie deklaruje metodę rozszerzającą
static class Klasa {public static void F(this double) {} }

126. W celu zaimportowania metod statycznych z klasy Przykład tak aby nie trzeba było podawać nazwy klasy przy ich wywołaniach, w języku C# należy użyć instrukcji
using static Przyklad

127. W celu obsługi błędów w programach napisanych w stylu funkcyjnym można użyć
Monady Either

128. Funktorem nazywamy typ, który będzie wyposażony w funkcję
Map (M<T>,<T->R))->M<R>

129. Niejawne konwersje typów w języku F#
są niedozwolone

130. Które zdanie jest prawdziwe
modyfikowalny stan utrudnia tworzenie programów asynchronicznych

131. Monadą nazywamy typ, który będzie wyposażony w funkcje
Bind i Return

132. Która deklaracja klasy jest poprawna
class MojaKlasa<T> where T : new() {}

133. Która metoda języka C# jest poprawnie zdefiniowana

```
async void Metoda() {
        int x = await Metoda2();
        Console.WriteLine("Komunikat");
}
```

134. Metody rozszerzające języka C# pozwalają rozszerzyć
tylko typy wartościowe

135. Funkcje, które nie mają efektów ubocznych, a ich wartość zależy tylko od parametrów nazywamy
czystymi

136. Zmienną, której typ jest wywnioskowany przez kompilator w języku C# tworzymy za pomocą słowa kluczowego
var

137. Słowo kluczowe fixed pozwala
ustawić zmienną w stałym miejscu w pamięci

138. Odpowiednikiem funkcji wyższych rzędów w programowaniu obiektowym jest wzorzec
strategia

139. Operacja mapowania jest w Linq zdefiniowana jako funkcja
select??

140. Jeżeli chcesz w języku C# zdefiniować kilka zadań asynchronicznych i wymagasz, aby program wykonywał się dalej po zakończeniu pierwszej z nich wykorzystasz metodę
Task.WaitAny

141. Operator L1@L2

Tworzy nową listę bez modyfikacji istniejących

142. Które wyrażenie języka F# stworzy poprawnie listę 3 elementową
[1;2;3];;

143. Interfejs kontrawariantny jest definiowany instrukcją
interface IInterface <in T> {}

144. Instrukcja using static Przykład służy do
Zaimportowania do pliku wszystkich elementów statycznych klasy Przykład

145. Metody Either można użyć w programach napisanych w stylu funkcyjnym do
Obsługi błędów

146. Typ String w języku c# jest typem
niemodyfikowalnym

147. W interfejsach inwariantnych zmienna typu może pojawić się tylko jako
Jako rezultat i/lub parametr funkcji

148. Niech dana będzie lista L. Operacja mapowania zwróci nową listę, która
ma taką samą długość co lista L

149. Operacja sortowania w Linq jest zdefiniowana jako
OrderBy

150. Operator >> w języku F# realizuje operację
złożenia funkcji

151. Która z poniższych operacji jest niedozwolona w F#
niejawnej konwersji

152. Metoda Func<double, double> F(int f) {} jest
funkcją wyższych rzędów

153. Sumowanie wszystkich elementów na liście jest przykładem operacji
agregacji

154. Brak wartości może być obsłużony w F# za pomocą typu
Option

155. Wyrażenie let f = fun x y -> fun a -> x+y+a języka F# tworzy
funkcję wyższych rzędów

156. W klasie generycznej języka C# kod class MojaKlasa<T>(...) kod
musi być zawsze poprawny dla dowolnego typu T

157. Właściwość public int X{get}
Może być zainicjowana w konstruktorze klasy w której się znajduje

158. Podczas definiowania funkcji tworzącej leniwą kolekcję w C# korzysta się z instrukcji
return

159. Dana jest klasa języka C#
który fragment kodu jest poprawny class MojaKlasa<T> { public MojaKlasa(T x) { } }
var obj = new MojaKlasa<int>(12);

160. W celu ustawienia funkcji, która ma być wywołana po zakończeniu operacji asynchronicznej skorzystasz z funkcji
ContinueWith

161. Języki domenowe wewnętrzne
wykorzystują składnię języka w którym są zdefiniowane

162. W języku F# funkcja let dodaj x y = x+y ma typ
x:int -> y:int -> int

163. Generyki w C# pozwalają na napisanie jednego kodu dla
różnych typów danych

164. Odpowiednikiem instrukcji var wynik = kolekcja.Where(x=>x>0).ToList() w formie zapytania jest
var wynik = (from x in kolekcja where x>0 select x).ToList()

165. Operator >> w F# służy do
Stworzenia nowej funkcji poprzez odpowiednie połączenie dwóch istniejących

166. Jakie operacje są niedopuszczalne w języku F#
niejawnej konwersji

167. Która metoda języka C# jest funkcją wyższych rzędów

Action<double> F(int x) {}

168. Monada jest elementem programistycznym pochodzącym z paradygmatu
funkcyjnego

169. W C# inwariantne zmienne typu mogą być wykorzystane
jako wartość zwracana lub parametr funkcji

170. W metodach rozszerzających mamy dostęp
tylko do składowych publicznych klasy rozszerzanej

171. Instrukcja języka F# l1@l2 gdzie l1 i l2 to listy
tworzy nową listę

172. Program #include using namespace std; template T max(T x, T y)
{ return (x > y)? x : y;} int main() { cout << max(3, 7) << std::endl; cout
<< max(3.0, 7.0) << std::endl; cout << max(3, 7.0) << std::endl; return 0; }
da w rezultacie
Żadne z powyższych
173. Zakresem widoczności zmiennych w wyrażeniu (letrec ((x 2) (y 3))
(+ x y)) jest
całe wyrażenie (let ((x 2) (y 3)) (+ x y))
174. . Przy przekazywaniu przez wartość argumentów "z" funkcji
parametrem aktualnym może być:
zmienna

175. Operator koniunkcji jest w języku PROLOG oznaczany symbolem
,

176. Za pomocą którego wyrażenia języka Scheme uzyskano poniższą
strukturę
(cons 1 (cons (cons 2 (cons 3 empty)) (cons (cons 5 empty) (cons 4
empty))))

177. Wyrażeniu języka C#: (x,y)=>(x*2)+(y*3) w języku Scheme
odpowiada wyrażenie
c) (lambda (x y) (+ (* x 2) (* y 3)))

178. Dana jest metoda języka C# void Funkcja(Func<int,int>f) {}. Które z
poniższych wywołań jest poprawne
Funkcja(a=>a+1)

ZBIÓR TESTÓW ZE STRONY http://wazniak.mimuw.edu.pl/ (nawet gdyby miały się nie przydać to zawsze warto)

Programowanie imperatywne jest ściśle związane z budową sprzętu komputerowego o architekturze:

von Neumanna Dobrze

Abstrakcją komórek pamięci (w paradygmacie imperatywnym) są:

zmienne Dobrze

Dziedziczenie jest cechą charakterystyczną dla programowania:

obiektowego Dobrze

Obiekt to powiązanie danych z:

operacjami na tych danych Dobrze

W programowaniu funkcyjnym nie występują:

pętle Dobrze

Automatyczne dowodzenie twierdzeń (prostych...) jest możliwe w programowaniu:

obiektowym Dobrze

Język C++ reprezentuje paradygmat:

imperatywny i obiektowy Dobrze

Pierwszym językiem obiektowym był język:

Simula 67 Dobrze

Czy optymalizacja kodu wykonywana przez kompilator może poprawić asymptotyczną złożoność obliczeniową programu?

tak, ale rzadko Dobrze

Składnię języków programowania opisuje się za pomocą

bezkontekstowych Dobrze

Program może zawierać dwie różne zmienne o tej samej nazwie, gdy są to zmienne:

alokowane dynamicznie Źle

globalne Źle

lokalne w dwóch różnych blokach Dobrze

lokalne w tym samym bloku Źle

L-wartością nazywamy:

bieżący adres zmiennej Dobrze

wynik wyrażenia arytmetycznego Źle

indeks tablicy Źle

wartość zmiennej po dokonaniu podstawienia Źle

Wiązanie statyczne:

może zmienić się w trakcie wykonania programu Źle

następuje w trakcie wykonania programu Źle

następuje przed wykonaniem programu Dobrze

odnosi się tylko do zmiennych globalnych Źle

Wnioskowanie o typie zmiennej jest najczęstsze w językach:

funkcyjnych Dobrze

Okres życia zmiennej to:

czas pomiędzy alokacją zmiennej a jej dealokacją Dobrze

Obiekty w Javie są alokowane:

dynamicznie, na stercie Dobrze

Spośród wymienionych tu języków najbliższy silnemu typowaniu jest:

C# Dobrze

Silne typowanie bywa ,,osłabiane *przez:*

niejawne konwersje typów Dobrze

Podtyp to:

typ powstały przez ograniczenie zakresu istniejącego typu, zgodny z owym typem Dobrze

W języku C++ dostęp do przesłoniętej zmiennej nielokalnej można uzyskać za pomocą operatora:

:: (dwa dwukropki) Dobrze

Pojęcie typu w językach imperatywnych bliskie jest pojęciu:

zbioru skończonego Dobrze

Który z opisanych poniżej typów można uznać za typ

abstrakcyjny? Rzecz dzieje się w języku C:

wbudowany typ float Dobrze

W której sytuacji tablica asocjacyjna byłaby istotnie wygodniejsza niż zwykła tablica?

zapisujemy kolor przejeżdżających samochodów, identyfikując je numerami rejestracyjnymi Dobrze

Ewentualne luki między przechowywanymi w pamięci polami rekordu biorą się z:

konieczności umieszczania pól pod adresami, których 1 lub 2 najmniej znaczące bity są zerami Dobrze

Załóżmy, że w języku C sprawdzamy równość struktur (oczywiście tego samego typu). Dlaczego w ogólności nie można tego zrobić przez porównywanie bloków pamięci?

napisy mogą zawierać nieistotne znaki za znacznikiem końca Dobrze

Który operator języka C jest potrzebny, gdy wykorzystujemy wskaźniki do adresowania pośredniego?

& Dobrze

Załóżmy, że p jest zmienną wskaźnikową. W którym języku wyrażenie ++p jest poprawne?
C++ Dobrze

Które stwierdzenie jest fałszywe w odniesieniu do klas w języku C++?

konstruktor nie może być przeciążany Dobrze

W Javie obiekty są alokowane:

dynamicznie na stercie Dobrze

Sparametryzowane typy abstrakcyjne uzyskuje się w C++ za pomocą deklaracji z użyciem słowa kluczowego:

template Dobrze

Który język nie pozwala na użycie parametrów z wartością domyślną?

C Dobrze

Przekazanie funkcji jako parametru można w C# osiągnąć za pomocą mechanizmu:

delegatów Dobrze

Który język nie sprawdza zgodności typów parametrów?

PHP Dobrze

Przy której deklaracji procedury f wywołanie f(2*x + 3) jest poprawne?

void f(int n) w języku C Dobrze

Chcąc w języku C przekazać do funkcji tablicę przez wartość, trzeba:

"obudować" ją strukturą i przekazać tę strukturę Dobrze

Jaką dodatkową cechę mają parametry stałe deklarowane w C++ z użyciem const w stosunku do parametrów w trybie wejściowym w ogóle?

nie mogą być zmieniane nawet w obrębie podprogramu Dobrze

Załóżmy, że x jest parametrem w trybie out w procedurze w Adzie. Która instrukcja ma szansę być poprawna?

x := y + 1 Dobrze

Jawne przekazywanie przez referencję jest w C# możliwe, jeśli umieścimy słowo kluczowe ref:

i przy parametrze formalnym, i przy aktualnym Dobrze

W językach z zakresem widoczności zmiennych wiązanym statycznie jako środowiska wykonywania przekazanego przez parametr podprogramu najczęściej używa się:

środowiska definicji przekazanego podprogramu Dobrze

W implementacji podprogramów bez zagnieżdżeń, ale z rekurencją i z dynamicznymi zmiennymi lokalnymi na stosie potrzebne jest przechowywanie w rekordzie aktywacyjnym:

tylko łącza dynamicznego Dobrze

Której cechy język obiektowy nie musi posiadać?

podprogramy rodzajowe Dobrze

Jakie ograniczenie na przedefiniowywanie metod trzeba narzucić w języku silnie typowanym?

przedefiniowana metoda musi zachować taki sam protokół Dobrze

Rozstrzyganie odwołań do bytów o takiej samej nazwie mających definicje w dwóch klasach bazowych odbywa się w C++ za pomocą:

operatora :: (dwa dwukropki) Dobrze

W języku C++ obiekty zaalokowane na stosie dealokowane są:

niejawnie Dobrze

Językiem, w którym stosowane jest zawsze dynamiczne wiązanie wywołań z metodami, jest:

Smalltalk Dobrze

Językiem, w któym klasa może być samoistna (tzn. nie mieć nadklasy), jest:

C++ Dobrze

W języku C++ metody, które mają być wiązane dynamicznie, deklaruje się za pomocą:

słowa virtual Dobrze

Który nagłówek poprawnie deklaruje w C++ metodę abstrakcyjną?

virtual void p()=0; Dobrze

Klasy "lekkie", deklarowane jako struct, alokowane na stosie i nie pozwalające na dziedziczenie występują w:

C# Dobrze

Który element nie występuje wJavaScripcie?

klasy Dobrze

Czego z zasady nie ma w językach funkcyjnych?

pętli Dobrze

Która cecha jest typowa dla języków funkcyjnych, a rzadko występuje w językach imperatywnych i obiektowych?

możliwość używania funkcji wyższego rzędu Dobrze

Listy służą w Lispie do zapisywania:

i danych, i kodu Dobrze

Wywołanie ((LAMBDA (x) (* x x)) 2) w języku Scheme:

wyświetli 4 Dobrze

Funkcja DISPLAY w języku Scheme:

wyświetla swój argument na ekranie Dobrze

Wartością wyrażenia (CAR ‘(A B C)) w języku Scheme jest:

A Dobrze

Wartością wyrażenia (CONS ‘(A B) ‘(C D)) w języku Scheme jest:

((A B) C D) Dobrze

Jak w języku Scheme należy zapisać wywołanie złożenia funkcji f z samą sobą na argumencie x, czyli (f o f)(x)?

(f (f x)) Dobrze

Które stwierdzenie nie jest prawdziwe w odniesieniu do języka ML?

lista może zawierać elementy różnych typów Dobrze

Do łączenia list w Haskellu służy:

operator ++ Dobrze

Sekwencja poleceń zmieniających krok po kroku stan maszyny jest charakterystyczna dla paradygmatu:

imperatywnego Dobrze

Powiązanie danych z operacjami na nich w całość, stanowiącą odrębną jednostkę, jest charakterystyczne dla paradygmatu:

obiektowego Dobrze

Składanie i obliczanie funkcji, bez stanu maszyny i bez efektów ubocznych, jest charakterystyczne dla paradygmatu:

funkcyjnego Dobrze

Dowodzenie wpisanego celu w oparciu o podane przesłanki za pomocą rezolucji jest charakterystyczne dla paradygmatu:

logicznegoDobrze

Który element nie jest charakterystyczny dla programowania strukturalnego?

skok Dobrze

Przykładem programowania sterowanego przepływem danych nie jest:

przetwarzanie kolejnych elementów tablicy w pętli for Dobrze

W programowaniu sterowanym zdarzeniami pojęcie zdarzenia raczej nie odnosi się do:

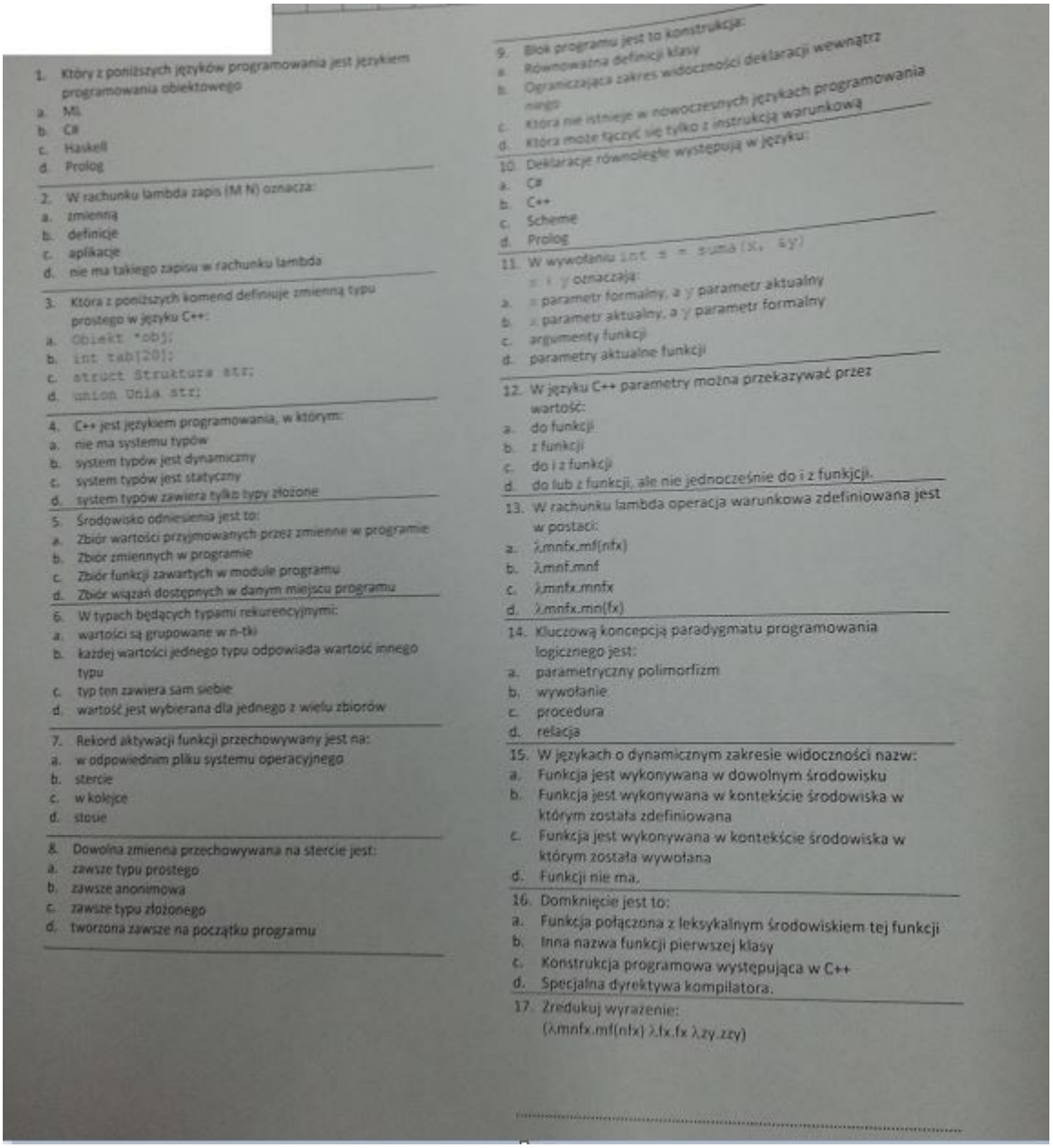
rekurencyjnego wywołania funkcji Dobrze

Które pojęcie jest najogólniejsze?

programowanie współbieżne Dobrze

Który język nie zawiera żadnych wbudowanych mechanizmów obsługi współbieżności?

C++ Dobrze



1. Który z poniższych języków programowania jest językiem programowania obiektowego
a. ML
b. C#
c. Haskell
d. Prolog

2. W rachunku lambda zapis (M N) oznacza:
a. zmienną
b. definicje
c. aplikacje
d. nie ma takiego zapisu w rachunku lambda

3. Która z poniższych komend definiuje zmienną typu prostego w języku C++:
a. `Obiekt *obj;`
b. `int tab[20];`
c. `struct Struktura str;`
d. `union Unia str;`

4. C++ jest językiem programowania, w którym:
a. nie ma systemu typów
b. system typów jest dynamiczny
c. system typów jest statyczny
d. system typów zawiera tylko typy złożone

5. Środowisko odniesienia jest to:
a. Zbiór wartości przyjmowanych przez zmienne w programie
b. Zbiór zmiennych w programie
c. Zbiór funkcji zawartych w module programu
d. Zbiór wiązań dostępnych w danym miejscu programu

6. W typach będących typami rekurencyjnymi:
a. wartości są grupowane w n-tki
b. każdej wartości jednego typu odpowiada wartość innego typu
c. typ ten zawiera sam siebie
d. wartość jest wybierana dla jednego z wielu zbiorów

7. Rekord aktywacji funkcji przechowywany jest na:
a. w odpowiednim pliku systemu operacyjnego
b. stercie
c. w kolejce
d. stosie

8. Dowolna zmienna przechowywana na stercie jest:
a. zawsze typu prostego
b. zawsze anonimowa
c. zawsze typu złożonego
d. tworzona zawsze na początku programu

9. Blok programu jest to konstrukcja:
a. Równoważna definicji klasy
b. Ograniczająca zakres widoczności deklaracji wewnątrz niego
c. Która nie istnieje w nowoczesnych językach programowania
d. Która może łączyć się tylko z instrukcją warunkową

10. Deklaracje równoległe występują w języku:
a. C#
b. C++
c. Scheme
d. Prolog

11. W wywołaniu `int s = suma(x, &y)` `x` i `y` oznaczają:
a. `x` parametr formalny, a `y` parametr aktualny
b. `x` parametr aktualny, a `y` parametr formalny
c. argumenty funkcji
d. parametry aktualne funkcji

12. W języku C++ parametry można przekazywać przez wartość:
a. do funkcji
b. z funkcji
c. do i z funkcji
d. do lub z funkcji, ale nie jednocześnie do i z funkcji.

13. W rachunku lambda operacja warunkowa zdefiniowana jest w postaci:
a. λmnfx.mf(nfx)
b. λmnf.mnf
c. λmnfx.mnfx
d. λmnfx.mn(fx)

14. Kluczową koncepcją paradygmatu programowania logicznego jest:
a. parametryczny polimorfizm
b. wywołanie
c. procedura
d. relacja

15. W językach o dynamicznym zakresie widoczności nazw:
a. Funkcja jest wykonywana w dowolnym środowisku
b. Funkcja jest wykonywana w kontekście środowiska w którym została zdefiniowana
c. Funkcja jest wykonywana w kontekście środowiska w którym została wywołana
d. Funkcji nie ma.

16. Domknięcie jest to:
a. Funkcja połączona z leksykalnym środowiskiem tej funkcji
b. Inna nazwa funkcji pierwszej klasy
c. Konstrukcja programowa występująca w C++
d. Specjalna dyrektywa kompilatora.

17. Zredukuj wyrażenie:
(λmnfx.mf(nfx) λfx.fx λzy.zzy)

.....

1. Który z poniższych języków programowania jest językiem programowania obiektowego?
a) ML
b) C#
c) Haskell
d) Prolog

2. Z rachunku lambda zapis: (M N) oznacza:
a) zmienną
b) definicje
c) aplikacje
d) nie ma takiego zapisu w rachunku lambda

3. Która z poniższych komend definiuje zmienną typu prostego w języku C++:
a) Obiekt *obj;
b) int tab[20];
c) struct Struktura str;
d) union Unia str;

4. C++ jest językiem programowania w którym:
a) nie ma systemów typów
b) system typów jest dynamiczny
c) system typów jest statyczny
d) system typów zawiera tylko typy złożonego

5. Środowisko odniesienia jest to:
a) zbiór wartości przyjmowanych przez zmienne w programie
b) zbiór zmiennych w programie
c) zbiór funkcji zawartych w module programu
d) zbiór wiązań dostępnych w danym miejscu programu

6. W typach będących typami rekurencyjnymi:
a) wartości grupowane są w n-tki
b) każdej wartości jednego typu odpowiada wartość innego typu
c) typ ten zawiera sam siebie
d) wartość jest wybierana dla jednego z wielu zbiorów

7. Rekord aktywacji funkcji przechowywany jest na:
a) w odpowiednim pliku systemu operacyjnego
b) stercie
c) kolejce
d) stosie

8. Dowolna zmienna przechowywana na stercie jest:
a) zawsze typu prostego
b) zawsze anonimowa
c) zawsze typu złożonego
d) tworzona zawsze na początku programu

9. Blok programu jest to konstrukcja:
a) równoważna definicji klasy
b) ograniczająca zakres widoczności deklaracji wewnątrz niego
c) która nie istnieje w nowoczesnych językach programowania
d) która może łączyć się tylko z instrukcją warunkową

10. Deklaracje równoległe występują w języku:
a) C#
b) C++
c) Scheme
d) Prolog

11. W wywołaniu: int s=suma(x,&y) x i y oznaczają:
a) x parametr formalny, &y parametr aktualny
b) x parametr aktualny, &y parametr formalny
c) argumenty funkcji
d) parametry aktualne funkcji

12. W języku C++ parametry można przekazać przez wartość:
a) do funkcji
b) z funkcji
c) do i z funkcji
d) do lub z funkcji ale nie jednocześnie do i z funkcji

13. W rachunku lambda operacja warunkowa zdefiniowana jest w postaci:
a) λmnfx.mf(nfx)
b) λmnf.mnf
c) λmnfx.mnfx
d) λmnfx.mn(fx)

14. Kluczową koncepcją paradygmatu programowania logicznego jest:
a) parametryczny polimorfizm
b) wywołanie
c) procedura
d) relacja

15. W językach o dynamicznym zakresie widoczności nazw
a) Funkcja jest wywoływana w kontekście środowiska w którym została zdefiniowana
b) Funkcja jest wywoływana w kontekście środowiska w którym została wywołana
c) Funkcji nie ma
d) Funkcja jest wykonywana w dowolnym środowisku

16. Domknięcie jest to:
a) funkcja połączona z leksykalnym środowiskiem tej funkcji
b) inna nazwa funkcji pierwszej klasy
c) konstrukcja programowa występująca w C++
d) specjalna dyrektywa kompilatora

17. Zredukuj wyrażenie:
(λmnfx.mf(nfx) λfx.fx λzy.zzy)

\+ 1 2
λm.(λn.(λfx.mf(nfx))1)2 m → 2
λn.(λfx.2f(nfx))1) n → 1
λfx.2f(1fx) n → 1
λfx.f(f(f x)) → 3

Rozpoczęto: [redacted]
Stan: [redacted]
Ukończono: [redacted]
Wykorzystany czas: [redacted]
Punkty: [redacted]
Ocena: [redacted]

Nawigacja po teście

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40		

Pokaż wersję drukowaną

Zakończ przegląd

Pytanie 1
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Niektóre rozwiązania mogą być uzależnione tylko

Wybierz jedną odpowiedź:
- a. od wersji kompilatora
- b. od wersji systemu operacyjnego
- c. o dobrym algorytmie ✓
- d. od wersji interpretera ✓

Twoja odpowiedź jest poprawna.

Pytanie 2
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

```
Dany jest kod:
class Program
{
    static IEnumerable<int> Go()
    {
        yield return 5;
        yield return 6;
        yield return 7;
    }
    static void Main(string[] args)
    {
        var xs = Go();
        foreach(var x in xs){}
        foreach(var x in xs){}
    }
}
```

Sposobuje się wyświetlenie na ekranie:

Wybierz jedną odpowiedź:
- a. Błędy przy kompilacji
- b. 5 6 7
- c. 5 7
- d. 5 6 7 ✓

Twoja odpowiedź jest poprawna.

Pytanie 3
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Podana metoda

```
int Go(int n)
{
    if (n<=1) return 1;
    int sum=0;
    for(int i=0;i<n;i++)
        sum+=Go(i);
    return sum;
}
```

wykonuje się dla wywołania:

Wybierz jedną odpowiedź:
- a. 6
- b. Inny ✓
- c. Pusty
- d. 5 10

Twoja odpowiedź jest poprawna.

Pytanie 4
Niepoprawna
Uzyskano ze 1,00
0,00 z 1,00
Flaga pytania

Do wykonania poniższego kodu

```
(print (loop n (mapcar (lambda (x) (* x x)) n)) '(1 2 3 4 5) '(1 2 3 4 5))
```

zarówno w Clisp jak i w Gnu Common Lisp potrzebna jest:

Wybierz jedną odpowiedź:
- a. brak 2 dodatkowych nawiasów
- b. brak 1 nawiasu ✗
- c. brak 2 nawiasów (1 lub 2)
- d. brak 3 dodatkowych nawiasów

Twoja odpowiedź jest niepoprawna.

Pytanie 5
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Wynik wersu kodowego języka M dla 5 x y =>y instrukcji jest wyrażeniem:

Wybierz jedną odpowiedź:
- a. Zamiana x na wartość y
- b. Przypisanie do zmiennej wyniku wartości 10
- c. Ogólnej błędnej postaci funkcji, bledne zastosowanie, w wyrażeniu jest 2 wyniki
- d. Przypisanie do zmiennej wyniku wartości y, wartości x nie zmienia się ✓

Twoja odpowiedź jest poprawna.

Pytanie 6
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Dany jest kod C#:

```
class Foo<T>
{
    public T[] a = new T[10];
    public T this[int i]
    {
        get { return a[i]; }
        set { a[i] = value; }
    }
}
```

Jest to?:

Wybierz jedną odpowiedź:
- a. Szablon klasy kontenerowej, wersja Array (Tablica)dla dowolnego typu
- b. Zwykła tablica typu generyczna, przeniesie w C# nie można tworzyć tablic generycznych.
- c. Indeksyt wyjątkowe generowane na stos w C#
- d. Szablon/kontenerowi w wersji Stos obsługującym jeden typ danych Comparable

Twoja odpowiedź jest poprawna.

Pytanie 7
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Zamiana kodowej wartości na poprawną wersję: skrobiarą jako:

Wybierz jedną odpowiedź:
- a. Macierzenie
- b. Hierowanie
- c. Parsowie
- d. Agregacja ✓

Twoja odpowiedź jest poprawna.

Pytanie 8
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Która instrukcja języka C# poprawnie definiuje pr metody kontenerującej

Wybierz jedną odpowiedź:
- a. class Klasa : IEnumerator { public static void CFun(Include, double d) {} }
- b. static class Klasa : IEnumerator { public static void CFun(int Include, double d) {}}
- c. class Klasa { public void CFun(double d) {} }
- d. static class Klasa { public static void CFun(double d) {} } ✓

Twoja odpowiedź jest poprawna.

Pytanie 9
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Operacja (3, 254, true, "ala ma kota"), języka F# stworzy

Wybierz jedną odpowiedź:
- a. Lista z trzyma krotkami
- b. Krotka
- c. Tu nie ✓
- d. Lista 4

Twoja odpowiedź jest poprawna.

Pytanie 10
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Funkcja rekurencyjna w F# definiowana za pomocą instrukcji:

Wybierz jedną odpowiedź:
- a. let rec=while
- b. let ... and ✓
- c. let recursive
- d. rec

Twoja odpowiedź jest poprawna.

Pytanie 11
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Wyrażenie języka M

```
let f = function
        | 0 -> 1
        | n -> n * (f (n-1))
        | _ -> -1
```

tworzy:

Wybierz jedną odpowiedź:
- a. Funkcję potęgi wzniesienia x do kwadratu
- b. Funkcję trzyargumentową
- c. Funkcję potęgi generyczną
- d. Funkcję silnia ✓ parametrów

Twoja odpowiedź jest poprawna.

Pytanie 12
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Która z podanych wyżej opcji języka C# reprezentuje wyrażeniem kontekst

Wybierz jedną odpowiedź:
- a. using ✓
- b. yield
- c. volatile
- d. p => p++

Twoja odpowiedź jest poprawna.

Pytanie 13
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Jaką cechę ogólne generycznych w C# posiada np.

Wybierz jedną odpowiedź:
- a. Jawnie zrzutowanych
- b. Jawnie zrzutowanych, które mogą być dodane do różnych elementach.
- c. Implicite zrzutowanych, które mogą być dodane do różnych elementach ✓
- d. Implicite zrzutowanych, które mogą być dodane do różnych typach danych ✓

Twoja odpowiedź jest poprawna.

Pytanie 14
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Funkcja mapgo w stałej akcesorze:

Wybierz jedną odpowiedź:
- a. Nie ma, mapGenerujące, należy nie korzystania programistów asynchronicznych w C# ✓
- b. Umożliwiają tworzenie programów asynchronicznych w C#
- c. Ułatwiają tworzenie programów asynchronicznych w C#
- d. Uniemożliwiają tworzenie programów asynchronicznych w C#

Twoja odpowiedź jest niepoprawna.

Pytanie 15
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Klasy anonimowe (nienazwane) w języku C# wykorzystuje się konstrukcję typu zwaną Method Chaining

Wybierz jedną odpowiedź:
- a. Nie ma takiej konstrukcji, i konstrukcja(int) i nie(int) ✓
- b. Nie ma takiej konstrukcji, i(zadnaint) i nie(int)
- c. Nie ma takiej konstrukcji, i(zadnaint) i nie(int)
- d. Nie ma takiej konstrukcji, i(int) i nie(int)

Twoja odpowiedź jest poprawna.

Pytanie 16
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Do prawda w C#, istnieją deklaratywy, dla kompilatora, w celu zasymulowania wartości, by? Są to przykładowo: wskazówki kompilatora, mapowy poziom zoptymalizować i wpisu pisarzom: kluczowy.

Wybierz jedną odpowiedź:
- a. Enumeratorów
- b. Chwilę
- c. Atrybutów ✓
- d. Wszystkie ✓

Twoja odpowiedź jest poprawna.

Pytanie 17
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Zdaje stanowisty może zawierać klasa C# pod jaki zawiadowania jako static

Wybierz jedną odpowiedź:
- a. tylko elementy statyczne ✓
- b. elementy statyczne i prywatnymi metod instancji
- c. elementy statyczne i metodami
- d. tylko statyczne zdaje i wszystkimi, dla nie statycznie metody

Twoja odpowiedź jest poprawna.

Pytanie 18
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Klasy anonimowe mogą zostać utworzone za pomocą konstrukcji wzorca Method Chaining

Wybierz jedną odpowiedź:
- a. class Przykład { public int Metoda(){ throw new NotImplementedException(); } }
- b. class Przykład { public void Metoda(){ throw new NotImplementedException(); } }
- c. class Przykład { public void Metoda(){ throw new NotImplementedException(); } } ✓
- d. class Przykład { public Przykład Metoda(){ throw new NotImplementedException(); } }

Twoja odpowiedź jest poprawna.

Pytanie 19
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Dana jest funkcja kod 5 x y =>y x

Która z poniższych odpowiedzi stanowią tylko wynik funkcja:

Wybierz jedną odpowiedź:
- a. 5 5 y =>5 x = 5 y
- b. let f x1 x2 x3 = fun y -> x1 x2 x3
- c. let x = fun y -> y
- d. let f (x,y) = x+y

Twoja odpowiedź jest poprawna.

Pytanie 20
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Dana jest funkcja:

```
let rec Foo xs =
    match (xs:'a list) with
    | [] -> []
    | [x] -> [x]
    | x::y::ys -> x::y::(Foo ys)
```

Aby móc do niej przekazać obiekt klasy Point3d, Point3d(1,2,3) gdzie klasa Point3d dziedziczy po Cos, znaczyło ten musi być:

Wybierz jedną odpowiedź:
- a. Kowariantny ✓
- b. Kontrawariantny
- c. Niezmienny
- d. Kowariantny

Twoja odpowiedź jest poprawna.

Pytanie 21
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Zdyskwalifikuj dlatch funkcje wyżej/ych napisów w programowaniu obiektowym:

Wybierz jedną odpowiedź:
- a. jest zawsze: Wyniku
- b. nie mogą wykorzystywać
- c. jest zawsze: Strategii ✓
- d. jest zawsze: Rozkazowy

Twoja odpowiedź jest poprawna.

Pytanie 22
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Funkcja kontrolowana w C# musi zawsze? wartość typu:

Wybierz jedną odpowiedź:
- a. Dowiedzieć
- b. Kompatybilny
- c. Kowariant
- d. Kontrawariant ✓

Twoja odpowiedź jest poprawna.

Pytanie 23
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Dany jest fragment programu:

```
let rec fib = function
    | 0 | 1 -> 1
    | n -> fib(n-1) + fib(n-2)
```

Która z poniższych fragmentów może przedstawić test funkcji fibonacciego

Wybierz jedną odpowiedź:
- a. let fibonacciTest() = printfn "%A" [for i in 0..10 -> fib i]
- b. Funkcja, bazuje fibonacciego, z testowaniem przed [return printf()]
- c. Funkcja, bazuje fibonacciego[return] [return i]/[wewnętrzna i]
- d. Funkcja, bazuje fibonacciego[return] [return wartosc]

Twoja odpowiedź jest niepoprawna.

Pytanie 24
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Który kurs bazuje, kurs z, Operacja fibonacciego wartości rzecz klasy, stanu

Wybierz jedną odpowiedź:
- a. Nie ma takiego, nie dotyczy, wyniku ✓
- b. nie jest zbudowa od klasy ✗
- c. klasy, kurs wartości, do stanu w zależności od funkcji. Które bazuje do nie zależnosmo
- d. jest zbudowa od klasy ✗

Twoja odpowiedź jest poprawna.

Pytanie 25
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

W mnożnikach rozszerzających mamy dostęp:

Wybierz jedną odpowiedź:
- a. do składowych prywatnych tych przeprzed klasy rozszerzonej
- b. tylko do składowych publicznych klasy rozszerzonej
- c. tylko do składowych prywatnych klasy rozszerzonej ✓
- d. do składowych chronionych klasy rozszerzonej

Twoja odpowiedź jest poprawna.

Pytanie 26
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Jaka nazwa klasy/funkcji jest wyznaczeniem przekazywanych nazwiązujących do parzystego/nie:

Wybierz jedną odpowiedź:
- a. funkcyjnego ✓
- b. Niekreślonego
- c. Przeciwstawnego
- d. Wielopostaciowego

Twoja odpowiedź jest poprawna.

Pytanie 27
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Która z poniższych fragmentów kodu pozwoli na objekt. Shuffle()

Wybierz jedną odpowiedź:
- a. class Generator{ public Throw(Type t) => throw new NotImplementedException(); }
- b. class Generator<T>{ public Throw(Type t) => throw new NotImplementedException(); } ✓
- c. static class Generator{ public Throw(Type t) => throw new NotImplementedException(); }
- d. static class Generator<T>{ public Throw(Type t) => throw new NotImplementedException(); }

Twoja odpowiedź jest poprawna.

Pytanie 28
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Dana jest metoda C# o nazwie Funkcja(Func<int, int> f) , która z poniższych wywołań tej funkcji jest poprawne:

Wybierz jedną odpowiedź:
- a. Funkcja(Console.ToString())
- b. Funkcja(Func)
- c. Funkcja(() => true)
- d. Funkcja(0) ✓

Twoja odpowiedź jest poprawna.

Pytanie 29
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Na przykład metoda F# map'ID mocno powiedział, że

Wybierz jedną odpowiedź:
- a. to jeden z powodów tego? funkcja na wyższej klasach parametrów
- b. ani słuszna
- c. to jeden z powodów tego funkcja potrzeby parametru ✓
- d. Żadne w wymienionych

Twoja odpowiedź jest poprawna.

Pytanie 30
Poprawna
Uzyskano ze 1,00
1,00 z 1,00
Flaga pytania

Która z poniższych instrukcji C# pozwoli na wyrzucenie wyjątkowego elementu w tablicy

Wybierz jedną odpowiedź:
- a. throw new ArgumentException();
- b. throw new ArgumentNullException();
- c. throw new IndexOutOfRangeException(); ✓
- d. throw new InvalidOperationException();

Twoja odpowiedź jest poprawna.

Zakończ przegląd

Porozstrzymy w kontakcie
Kontakt
email
Filtrowanie learningpoz.pl

Powered by Moodle
Made with ♥ by moodle.com

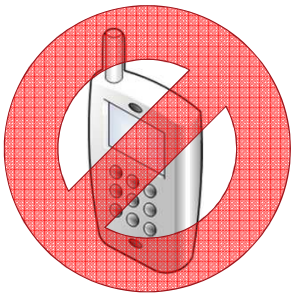
Paradygmaty programowania

dr inż. Łukasz Bartczuk

Katedra Inżynierii Komputerowej
p. 517

Lukasz.Bartczuk@kik.pcz.pl

Wprowadzenie



Agenda wykładu

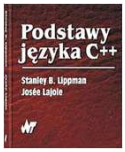
1. Wprowadzenie
2. Typy danych
3. Zmienne
4. Nazwy i ich wiązanie
5. Abstrakcje programistyczne
6. Programowanie obiektowe inaczej
7. Wstęp do programowania funkcyjnego
8. Język Scheme
9. Rachunek Lambda
10. Wstęp do programowania w logice
11. Język Prolog
12. Domain Specific Languages (DSL)

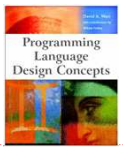
Z czego się uczyć

- Wykład + laboratoria
- Literatura + czasopisma
- Dokumentacja oprogramowania
- Internet

Własne próby pisania programów

Przykładowa literatura





Ewolucja języków programowania

Początkowo programy wyglądały tak:

```
01010101100010011110010110000011
11101100000100001100011101000101
11110100000000101000000000000000
00000000110001110100010111111000
00000110000000000000000000000000
10000110010001011111100010000110
01010101111101001000110100000100
00000010111101000100010111111100
01101000000000000000000000000000
00000000110010011100001110010000
10010000100100001001000010010000
```


Ewolucja języków programowania

Ta sama funkcja w assemblerze ...

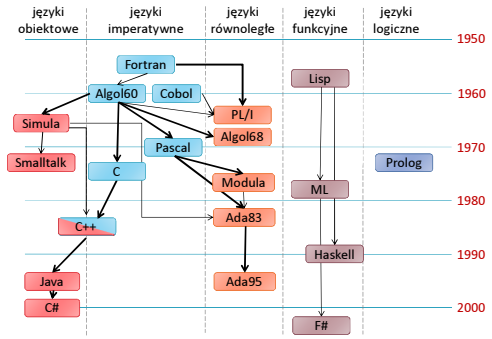
```
push ebp
mov ebp, esp
sub esp, 0x10
mov DWORD PTR [ebp-0xc], 0x5
mov DWORD PTR [ebp-0x8], 0x6
mov eax, DWORD PTR [ebp-0x8]
mov edx, DWORD PTR [ebp-0xc]
lea eax, [edx+eax*1]
mov DWORD PTR [ebp-0x4], eax
mov eax, 0x0
leave
ret
```

Ewolucja języków programowania

... i w C

```
int main()
{
  int a = 5;
  int b = 6;
  int c = a+b;
  return 0;
}
```


Ewolucja języków programowania



Co to jest paradygmat?

„Paradygmat to przyjęty sposób widzenia rzeczywistości w danej dziedzinie, doktrynie itp."

„zespół form fleksyjnych (deklinacyjnych lub koniugacyjnych) właściwy danemu typowi wyrazów; wzorzec; model deklinacyjny lub koniugacyjny"

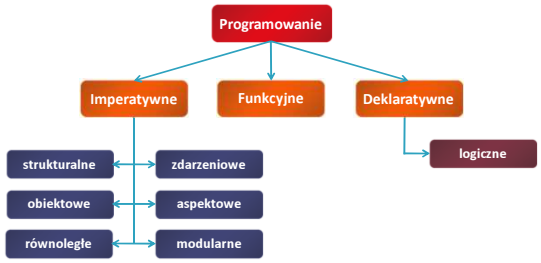
Słownik Języka Polskiego PWN

Paradygmat programowania

Paradygmat programowania jest wyróżniającym się stylem programowania. Każdy paradygmat jest scharakteryzowany przez dominację pewnych kluczowych koncepcji.

- zbiór koncepcji reprezentujących podejście do implementacji algorytmów
- zbiór mechanizmów używanych przez programistę do pisania programów i określających jak te programy będą następnie wykonywane przez komputer

Podstawowe rodzaje paradygmatów



Czym się różnią?

- program = kod + dane
- kod = algorytm + język programowania
- dane = struktury danych + język programowania

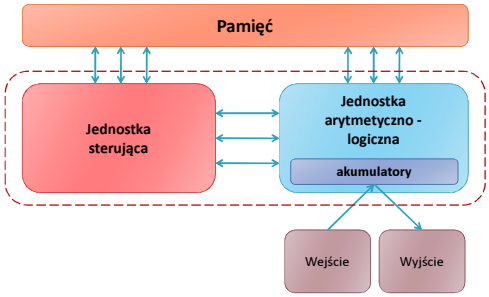
- sposób patrzenia na dane
- sposób patrzenia na kod
- sposób wiązania danych z kodem
- sposób patrzenia na przepływ sterowania

Programowanie imperatywne

Programowanie imperatywne polega na sekwencyjnym wykonywaniu kodu, czyli instrukcji modyfikujących wartości zmiennych

- Lat. *Imperare – wykonuj komendy*
- Kluczowe koncepcje: ***zmienne, komendy, procedury, dane abstrakcyjne***
- Jest bezpośrednim odzwierciedleniem sposobu działania maszyny von Neumanna

Maszyna von Neumanna



Programowanie imperatywne - przykład

```
int suma(int dane[], int ilosc)
{
   int wynik = 0;
   for(int i=0; i<ilosc; i++)
      wynik = wynik + dane[i];
      // wynik += dane[i];
   return wynik;
}
```

Programowanie obiektowe

Programowanie obiektowe polega na operowaniu obiektami, czyli połączenia danych i operacji jakie można na nich wykonywać

- Jest rozszerzeniem paradygmatu programowania imperatywnego
- Obiekty pozwalają na modelowanie elementów zarówno świata rzeczywistego jak i cybernetycznego.
- Kluczowe koncepcje: ***obiekty, klasy i podklasy, dziedziczenie polimorfizm***

Programowanie obiektowe - przykład

```
class Tablica {
   private:
     int* dane;
     int n;

   public:
     void Wypelnij() {
       for(int i=0; i<n; i++)
         dane[i] = rand();
     }

     int suma() {
       int wynik = 0;
       for(int i=0; i<n; i++)
         wynik += dane[i];
       return wynik
     }
};
```


Programowanie funkcyjne

Programowanie funkcyjne operuje nie na zmiennych lecz na wartościach

- Każdy program jest wyrażeniem
- Każde wyrażenie wyznacza wartość
- Jedyną akcją jest wywołanie funkcji
- Jedyną regułą kompozycji jest składanie funkcji
- Kluczowe koncepcje: ***wyrażenia, funkcje, parametryczny polimorfizm,*** a także ***dane abstrakcyjne i leniwe obliczenia***

Programowanie funkcyjne - przykład

```
(define (SumaWszystkichElementowListy lista)
  (cond (
      (empty? Lista) 0)
      (else (+ (first lista)
        (SumaWszystkichElementowListy (rest lista))))))
```


Programowanie w logice

Programowanie w logice polega na opisaniu znanych faktów dotyczących analizowanego problemu i relacji zachodzących pomiędzy tymi faktami.

- Implementują relację pomiędzy zbiorami wartości S i T
- Kluczowe koncepcje: ***fakty, relacje, zapytania.***

Programowanie w logice - przykład

```
dziecko (Jan, Adam).
dziecko (Jan, Ewelina).
dziecko (Tadeusz, Jan).
dziecko (Beata, Jan).

wnuk(X, Y) :- dziecko(X,Z),
dziecko(Z,Y).

?- wnuk(Jan, Adam).
?- wnuk(Tadeusz, Ewelina).
?- wnuk(Beata, X).
```

Wartości i typy danych

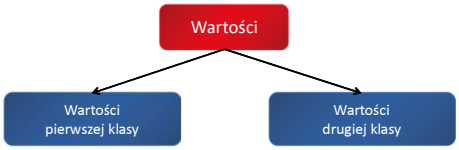
Wartości

Wartość jest obiektem, który może być modyfikowany przez program

Operacje jakie mogą być wykonywane na wartościach:

- Obliczane ich
- Przechowywanie ich w pamięci (odczytywanie i zapisywanie)
- Przekazywanie do funkcji
- Zwracanie z funkcji

Wartości

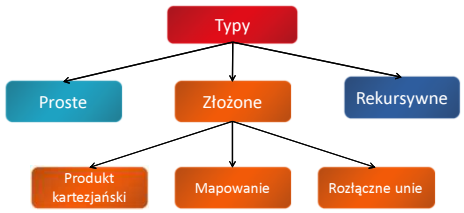


Typ danych

Typ danych jest zbiorem wartości wraz z co najmniej jedną operacją jaką można wykonywać na wszystkich tych wartościach

```
{ „Ala”, -1, 3.14, [12, 84] }

{ …, -2, -1, 0, 1, 2, … }
```


Typy danych



Proste typy danych

Typy proste – mogą przechowywać wartości proste, czyli wartości, które nie mogą być podzielone na mniejsze części

Przykłady:

`int, float, double, char, bool, wskaźniki`

Złożone typy danych

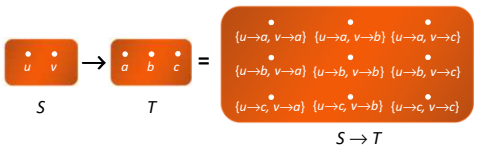
Typy złożone – mogą przechowywać wartości złożone, czyli wartości, które mogą być podzielone na mniejsze części

- Tablice
- Struktury danych (listy, stos itd.)
- Struktury / rekordy
- Unie
- Klasy

Mapowania

W ***mapowaniu*** każdej wartości typu S odpowiada wartość typu T.

$$m: S \to T = \{m \mid x \in S \Rightarrow m(x) \in T\}$$



$$\#(S \to T) = (\#T)^{\#S}$$

Mapowania

W ***mapowaniu*** każdej wartości typu S odpowiada wartość typu T.

Typowymi przedstawicielami mapowań są: ***tablice, słowniki i funkcje mapujące (funkcje)***.

Podstawowe operacje jakie można wykonywać na mapowaniu to:

- ***stworzenie*** mapowania z n elementów
- ***indeksacja*** wybranie jednego elementu mapowania na podstawie jego indeksu

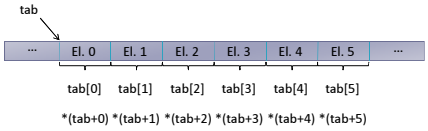
Mapowania - tablice

```
#include<iostream>

int main()
{
  bool tab1[3];
  int tab2[] = {0,23,53};

  tab1[0] = 4;
  for(int i=0; i<3; i++)
  {
     std::cout<<tab2[i]<<std::endl;
  }
  return 0;
}
```


Mapowania - tablice

W języku C/C++ nie istnieje specjalny typ tablicowy, a tablice generowane jako ciągły blok pamięci o określonym rozmiarze.

```
int tab[6]
```



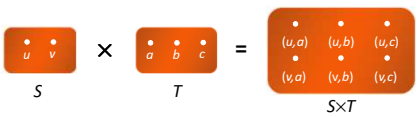
tab[0] tab[1] tab[2] tab[3] tab[4] tab[5]

*(tab+0) *(tab+1) *(tab+2) *(tab+3) *(tab+4) *(tab+5)

Produkt kartezjański

W typach będących ***produktem kartezjańskim*** wartości kilku (być może różnych) typów są grupowane w n-tki.

$S \times T = \{(x,y) \mid x \in S; y \in T\}$



S T $S \times T$

$\#(S \times T) = \#S \cdot \#T$

Produkt kartezjański

W typach będących ***produktem kartezjańskim*** wartości kilku (być może różnych) typów są grupowane w n-tki.

Podstawowe operacje jakie można wykonywać na n-tkach to:

- ***stworzenie*** n-tki z n elementów
- ***wybranie*** jednego z elementów n-tki

Produkt kartezjański - struktury

```
#include<iostream>

enum Miesiac {sty, lut, mar, kwi, maj, cze, lip, sie,
              wrz, paz, lis, gru};
struct Data {
  Miesiac miesiac;
  byte dzien;
};

int main()
{
  struct Data wyklad = {lut, 21};
  std::cout<<wyklad.miesiac<<" "<<wyklad.Dzien
           <<std::endl;
  return 0;
}
```


Produkt kartezjański - struktury

Struktury w języku C++ są generowane w ciągłym obszarze pamięci.

```
struct Data
{
  int i;
  char c;
};
```

Jaki będzie wynik działania funkcji **sizeof(struct Data)?**

Unie

W ***uniach*** wartość jest wybierana dla jednego z wielu (zazwyczaj różnych) zbiorów

Podstawowe operacje jakie można wykonywać na uniach to:

- ***stworzenie*** unii poprzez ustawienie wartości z T lub z S i odpowiednie jej oznaczenie
- ***sprawdzenie etykiety*** - określenie czy wartość pochodzi ze zbioru S czy z T
- ***projekcja*** odzyskanie wartości zbioru S lub zbioru T

Unie (C++)

```
#include<iostream>

int main()
{
  union Przyklad
  {
    char znaki[4];
    int calkowita;
  };

  Przyklad p;
  p.znaki[0] = 'A'; p.znaki[1] = 'B';
  p.znaki[2] = 'C'; p.znaki[3] = '\n';

  std::cout<<p.znaki<<" "<<p.calkowita<<std::endl;
}
```


Unie (Ada)

```
type Dokladnosc is (dokladna, niedokladna);
Type Liczba (dok:Dokladnosc := dokladna) is
    record
        case dok of
            when dokladna => calk: Integer;
            when niedokladna => rzecz: Float;
        end case;
    end record;

pi: constant Liczba := (dok=> niedokladna, rzecz=>
3.14);

function zaokr(licz: Liczba) return Float is
    case licz.dok is
        when dokladna => return licz.calk;
        when niedokladna => return licz.rzecz;
    end case;
end;
```

Typy rekursywne

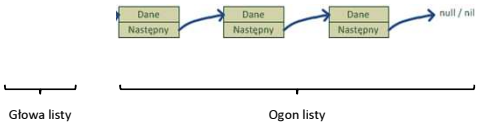
Typ rekursywny to taki, który w swojej definicji zawiera sam siebie.

Przykładowe typy rekursywne:

- Listy:

  $ListaInt = \{pusta_lista\} \cup \{Elem(i, l) \mid i \in int; l \in ListaInt\}$
- Łańcuchy znaków (string) (Haskell, Prolog)
- Drzewa

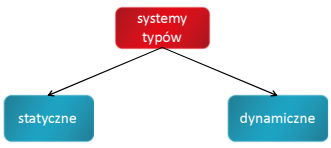
Listy jako struktura rekurencyjna



Głowa listy Ogon listy

Systemy typów

System typów może być zdefiniowany jako zbiór reguł do klasyfikowania wyrażeń zgodnie z rodzajem wartości, które wyznaczają.



Statyczne systemy plików

W ***statycznych systemach typów*** każda zmienna i wyrażenie posiada ustalony typ (ustawiony przez programistę lub wywnioskowany przez kompilator)

Właściwości statycznych systemów typów:

- **są bardzie efektywne niż systemy dynamiczne**
- **wymagają sprawdzenia już na etapie kompilacji programu**
- **wymagają mniej pamięci**
- **są bardziej bezpieczne**

Przykłady języków o statycznych systemach typów:
Ada, C, C++, C#, Java, Pascal, Visual Basic

Dynamiczne systemy typów

W ***dynamicznych systemach typów*** wartości posiadają ustalony typ, ale zmienne i wyrażenia nie

Właściwości dynamicznych systemów typów:

- są mniej efektywne niż systemy statyczne
- wymagają sprawdzenia podczas wykonywania programu
- wymagają więcej pamięci
- są mniej bezpieczne

Przykłady języków o dynamicznych systemach typów:
Lisp, JavaScript, Perl, PHP, Prolog, Python, Smalltalk

Zmienne

Zmienne

Zmienną można sobie wyobrazić jako ***„pudełko”*** służące do przechowywania wartości.

Operacje jakie można wykonać na zmiennych:

- Stworzenie zmiennej
- Wyciągnięcie wartości z „pudełka” – ***operacja odczytu***
- Włożenie wartości do „pudełka” – ***operacja zapisu***
- Usunięcie zmiennej

Zmienne

W rzeczywistości zmienne są segmentem w pamięci, który posiada:

- ***unikalny adres***
- ***wartość***
- ***rozmiar***
- ***stan***

Zmienne

Odczytanie wartości zmiennej

- odczytanie ciągu bitów
- zinterpretowanie ciągu bitów jako wartości

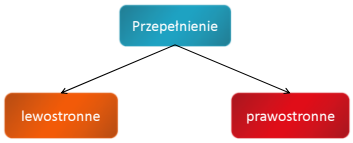
Zachowanie wartości zmiennej

- konwersja wartości na ciąg bitów
- zapisanie ciągu bitów w pamięci

Zmienne – przepełnienie

Ponieważ wartość zmiennej jest ciągiem bitów o określonej długości może wystąpić ***efekt przepełnienia***.

Z przepełnieniem mamy do czynienia w momencie, gdy do zmiennej o określonej wielkości chcemy wpisać dane o większej liczbie bitów.



Zmienne – przepełnienie

Aby przeciwdziałać przepełnieniu wykorzystane mogą być różne techniki:

1. ***obcinanie wartości***
2. ***zwracanie wyjątku***
3. ***nie zwracanie uwagi na przepełnienie***

Język C++, wykorzystuje strategię 1 i 3 w zależności czy przepełnienie występuje po prawej czy lewej stronie wartości

Zmienne – przepełnienie lewostronne

Problem 1:

```
for(char i=0;i<256; i++)
   cout<<i<<endl;
```

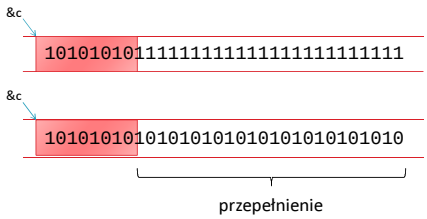
```
1.  i = 255;   //   11111111
2.  i = 256;   //  100000000
3.  i = 0;     //   00000000
```

Problem 2:

```
char x=255;
x++;
int wynik = 1000/x;
```

Zmienne – przepełnienie prawostronne

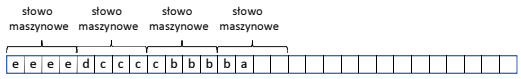
```
char c = 170;
int *wskaznik = (int*)&c;
*wskaznik = 2863311530;
```



Zmienne – położenie zmiennych

```
int  main()
{
  char a = 1;
  int  b = 2;
  int  c = 3;
  char d = 4;
  int  e;
  e = a+b+c+d;
  return 0;
}
```

```
int  main()
{
  char a = 1;
  char d = 4;
  int  c = 3;
  int  b = 2;
  int  e;
  e = a+b+c+d;
  return 0;
}
```

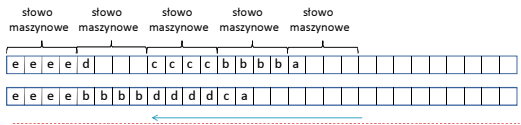


Na szczęście pamięć nie jest przydzielana w ten sposób !!

Zmienne – położenie zmiennych

```
int  main()
{
  char a = 1;
  int  b = 2;
  int  c = 3;
  char d = 4;
  int  e;
  e = a+b+c+d;
  return 0;
}
```

```
int  main()
{
  char a = 1;
  char c = 3;
  int  d = 4;
  int  b = 2;
  int  e;
  e = a+b+c+d;
  return 0;
}
```



Zmienne niezainicjalizowane

Pamięć nie może być pusta!!!!

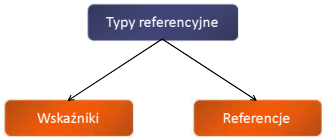
```
#include <stdio.h>

int a;

int main()
{
   int b;
   printf("%d", a+b);
   return 0;
}
```

Zmienna niezainicjalizowana

A co w przypadku zmiennej „a"?

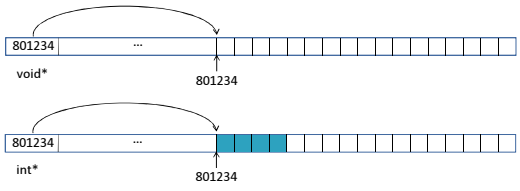
Typy referencyjne

Referencje jako zmienne pozwalają na przechowywanie odniesienia (adresu) do danych



Wskaźniki

Wskaźnik jako wartość jest adresem w pamięci, a jako zmienna jest pojemnikiem do przechowywania tego adresu



Referencja i dereferencja

Referencja – przypisanie adresu do wskaźnika

W języku C++ operator &

```
char ch = 'a';
char* wsk;
wsk = &ch
```

Dereferencja – pobranie wartości z danego adresu

W języku C++ operator *

```
printf("%c", *wsk);
```

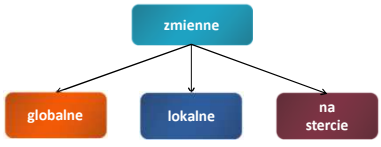

Referencje

Referencje w C++ są „uproszczoną" i „zubożoną" wersją wskaźników

- nie ma możliwości odniesienia się do samej referencji (do adresu, który jest do niej przypisany)
- referencja może być ustawiona tylko w momencie tworzenia i nie może być zmieniona później
- referencje nie mogą być puste
- referencje nie mogą być nie zainicjalizowane

Czas życia zmiennej

Czas życia zmiennej jest przedziałem czasu pomiędzy stworzeniem, a zniszczeniem zmiennej



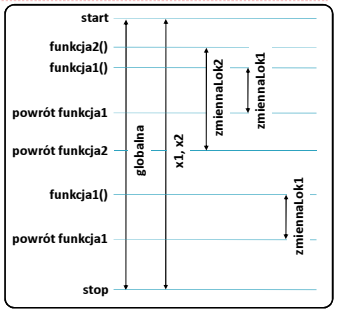
Zmienne lokalne i globalne

Zmienna globalna – zmienna dostępna w dowolnym miejscu w programie. Czas jej życia jest równy czasowi działania aplikacji

Zmienna lokalna – zmienna zadeklarowana wewnątrz procedury lub funkcji i dostępna tylko w wewnątrz tej procedury lub funkcji. Czas jej życia jest równy czasowi aktywacji procedury lub funkcji

Zmienne lokalne i globalne

```
int globalna;

void funkcja1() {
  int zmiennaLok1;
  …
}

void funkcja2() {
  int zmiennaLok2;
  …
  funkcja1();
  …
}

int main() {
  int x1, x2;
  funkcja2();
  funkcja1();
  return 0;
}
```



Zmienne na stercie

Zmienna na stercie – zmienna anonimowa, tworzona i niszczona w dowolnym momencie działania aplikacji. Dostęp do zmiennej na stercie jest możliwy za pomocą wskaźników

- tworzone są za pomocą alokatorów:
  - C: **malloc()**
  - C++, C#, Java, ADA: **new**
- niszczone są za pomocą delokatorów
  - C: **free()**
  - C++: **delete**
  - C#, Java: **garbage collector**

Nazwy i ich wiązanie

Czym jest nazwa?

Nazwa (identyfikator) jest mnemonicznym ciągiem znaków wykorzystywanym do reprezentowania czegoś innego.

Z nazwami łączą się następujące pojęcia:

- Wiązanie i czas jego wystąpienia
- Środowisko odniesienia
- Zakres ważności wiązania
- Czas życia wiązania

Wiązanie

Wiązanie jest skojarzeniem dwóch elementów np. nazwy i obiektu na który wskazuje.

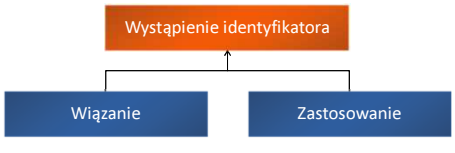
Czas wystąpienia wiązania jest to moment, w którym następuje stworzenie wiązania

Obiekty „bindable”

Obiekty „bindable” są to obiekty, które mogą być związane z identyfikatorem.

- ***C***: typy, zmienne, funkcje
- ***Java***: wartości, zmienne lokalne, zmienne klasy i instancji, metody, klasy i pakiety
- ***Ada***: typy, wartości, zmienne, procedury, wyjątki, pakiety, zadania

Bloki, a widoczność nazwy




```
int main()
{
   int n = 10;
   n = n+1;
   return 0;
}
```


Deklaracje

Deklaracja jest konstrukcją służącą do tworzenia wiązań.

- Deklaracje typów
- Deklaracje stałych
- Deklaracje zmiennych
- Deklaracje procedur i funkcji
- Deklaracje wyjątków
- Deklaracje klas
- Deklaracje pakietów
- ...

Deklaracje typów

Deklaracja typu - powiązanie identyfikatora do istniejącego typu danych

```
typedef char* LancuchZnakow
```

Definicja nowego typu – przypisanie nazwy do nowego typu danych, który nie jest odpowiednikiem typu już istniejącego

```
struct DaneOsobowe
{
    LancuchZnakow Imie;  LancuchZnakow Nazwisko;
};
```


Deklaracja procedur/funkcji

Deklaracja procedury lub funkcji wiąże identyfikator z procedurą lub funkcją

```
#include <iostream>
using namespace std;

double dodaj(double, double);

int main() {
    cout<<dodaj(1,2)<<endl;
    system("pause");
}

double dodaj(double x, double y) {
  return x+y;
}
```

```
#include <iostream>
using namespace std;

double dodaj(double x, double y) {
  return x+y;
}

int main() {
    cout<<dodaj(1,2)<<endl;
    system("pause");
}
```


Deklaracje złożone

Deklaracje równoległe

Scheme:

```
(define dodaj (let ((a 2) (b 3)) (+ a b)))

(define dodaj (let ((a 2) (b a)) (+ a b)))
```


Deklaracje sekwencyjne

Scheme:

```
(define dodaj (let* ((a 2) (b a)) (+ a b)))
```

C++:

```
int main()
{
   int x=2, y=x+2;
   …
}
```


Deklaracje rekurencyjne

Scheme:

```
(letrec ((even?
           (lambda (n)
             (if (zero? n) #t (odd? (- n 1)))))
         (odd?
           (lambda (n)
             (if (zero? n) #f (even? (- n 1))))))
(even? 88))
```

C++:

```
struct Element
{
   int dane;
   Element* nastepny;
}
```

Środowisko odniesienia

```
x = f(y) + 12;
```

Środowisko odniesienia – jest to zbiór dostępnych wiązań występujących w określonym miejscu w programie.

Środowisko odniesienia

```
int globalna;

void f(double x) {
  float x1 = 3.14;
  x1 += x;
}

int main() {
  double x, y;
  x = f(y) + 12;
  return 0;
}
```

globalna → zmienna typu int
f → funkcja
x → parametr funkcji typu double
x1 → liczba rzeczywista 3.14

globalna → zmienna typu int
f → funkcja
main → funkcja
x → zmienna typu double
y → zmienna typu double

Zakres widoczności nazwy

Zakres widoczności deklaracji – fragment programu, w którym deklaracja jest dostępna.

Blok – jest to konstrukcja programu, ograniczająca zakres widoczności dowolnej deklaracji wewnątrz niego

Bloki, a widoczność nazwy

```
void funkcja1() {
  int zmiennaLok1 = 0;
  printf("%d", zmiennaLok1);
  for(int i=0; i<10; i++)
  {
     float zmiennaLok1 = 3.14;
     printf("%f", zmiennaLok1);
  }
  printf("%d", zmiennaLok1);
}
```


Zakres widoczności nazwy

Język programowania jest językiem o ***zakresie statycznym***, jeżeli procedura lub funkcja jest wykonywana w kontekście środowiska w którym została zdefiniowana.

Język programowania jest językiem o ***zakresie dynamicznym***, jeżeli procedura lub funkcja jest wykonywana w kontekście środowiska w którym została wywołana.

Zakres widoczności nazwy – przykład

```
const int s = 2;

int f(int x) {
  return s*x;
}

void p(int y) {
  printf("p: %d", f(y));
}

void q(int z) {
  const int s = 3;
  printf("q: %d", f(z));
}

void r(int z) {
  const char s[] = "ALA";
  printf("r: %d", f(z));
}
```

Czas życia wiązania, a czas życia obiektu

Czas życia wiązania jest przedziałem czasu pomiędzy stworzeniem wiązania, a jego usunięciem.

Czas ↓

- Stworzenie obiektu
- Stworzenie wiązania
- Odwołania do obiektów za pomocą identyfikatora
- Tymczasowa dezaktywacja wiązania
- Wznowienie wiązania
- Usunięcie wiązania
- Usunięcie obiektu

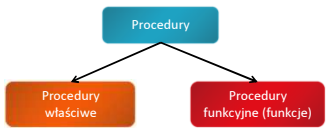
Kolejność zależna od języka i zastosowania

Funkcje

Abstrakcja

W języku programowania abstrakcja pozwala na rozróżnienie **co** jednostka programu robi i **jak** jednostka programu pracuje.

Procedura – jednostka obejmująca instrukcje programu.



Procedura właściwa

Procedura właściwa obejmuje komendy do wykonania i podczas działania modyfikuje wartości zmiennych.

C++:

```
void I(P1,…,PN)
{
  B
}
```

Pascal:

```
procedure I(P1,…,PN)
var
  D
begin
  B
end;
```

Procedura funkcyjna

Procedury funkcyjne (funkcje) obejmują wyrażenia, których wartość powinna zostać obliczona. Kiedy funkcja jest wywoływana zwraca wyznaczoną wartość zawartego w niej wyrażenia.

C++:

```
T I(P1,…,PN)
{
  B
}
```

Pascal:

```
function I(P1,…,PN) : T
var
  D
begin
  B
end;
```

Parametry i argumenty

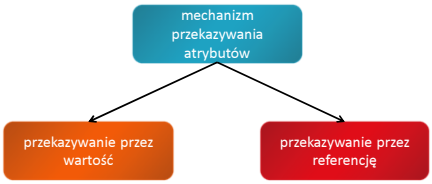
Parametr formalny jest identyfikatorem, za pomocą którego może mieć dostęp do argumentu.

Parametr aktualny jest wyrażeniem, które zwraca nam argument.

Argument jest wartością lub jednostką przekazywaną do procedury.

Mechanizm przekazywania parametrów

Machanizm przekazywania parametrów - forma skojarzenia wartości atrybutu z odpowiednim parametrem formalnym.



Przekazywanie parametrów przez wartość

Przekazywanie parametrów przez wartość - pozwala aby wartość była kopiowana do i/lub z procedury.

Przekazywanie „do”

Podczas wywoływania procedury tworzona jest zmienna lokalna i inicjalizowana wartością odpowiadającego jej argumentu. Wszelkie modyfikacje tej zmiennej lokalnej **nie są widoczne** poza procedury.

ADA:

```
procedure Dodaj(a, b: in Integer)
is
  suma : Integer := 0;
begin
  a := a + 5;
  suma := a + b;
end;
…
x := 5;
Dodaj(x, 5);
```

C++:

```
void Dodaj(int a, int b)
{
  int suma = 0;
  a += 5;
  suma = a + b;
}
…
x = 5;
Dodaj(x, 5);
```

Przekazywanie „z”

Podczas wywoływania funkcji tworzona jest zmienna lokalna ale **nie inicjalizowana**. **Kiedy następuje powrót z procedury wartość tej zmiennej jest przypisana do argumentu**.

```
ADA:

procedure Dodaj(a, b: in Integer, suma : out Integer) is
begin
  a := a + 5;
  suma := a + b;
end;
…
x: Integer;
…
Dodaj(3, 4, x);
```

Przekazywanie „do i z”

Jest to połączenie dwóch poprzednich metod.
Podczas wywoływania funkcji tworzona jest zmienna lokalna i **inicjalizowana** wartością odpowiadającego jej argumentu.
Kiedy następuje powrót z procedury wartość tej zmiennej jest przypisana do argumentu.

```
ADA:

procedure PodzielPrzez2(a: in out Float) is
begin
  a := a / 2;
end;
…
x : Float := 5.0;
…
PodzielPrzez2(x);
```

Przekazywanie przez referencję

Przekazywanie parametrów przez referencję - pozwala aby parametr formalny był przypisany bezpośrednio do argumentu

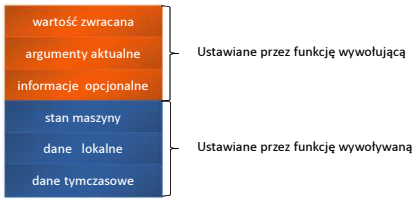
aliasing

Aliasing – występuje przy przekazywaniu parametrów przez referencję gdy, dwa lub więcej parametrów formalnych jest bezpośrednio przypisanych do tej samej zmiennej.

```
void funkcja(int &v1, int &v2, int &v3)
{
  v1 -= v3;
  v2 += v3;
}

…

  funkcja(x,y,z);
  funkcja(x,y,x);
```

Środowisko wykonania funkcji

Środowisko wykonania funkcji (rekord aktywacji/ramka aktywacji) jest to struktura danych zawierająca informacje o tym środowisku.



Implementacja funkcji w C/C++

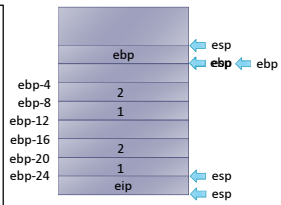
Proces wywołania funkcji można podzielić na trzy etapy:

1. Przygotowanie do wywołania funkcji (przygotowanie środowiska funkcji)
2. Wywołanie i wykonanie funkcji
3. Zakończenie wykonywania funkcji (usunięcie środowiska funkcji)

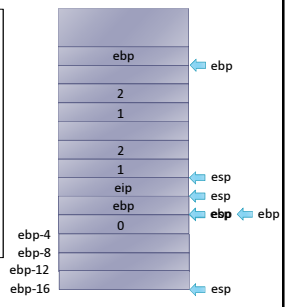
Środowisko wywołania funkcji - przykład

```
int dodaj(int x, int y)
{
    int z = 0;
    return x+y+z;
}

int main  ()
{
    int a = 1;
    int b = 2
    int c =  dodaj(a,b);
    return 0;
}
```

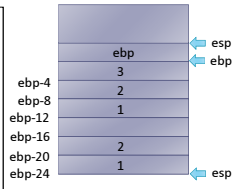
```
int main  ()
{
    int a = 1;
    int b = 2
    int c =  dodaj(a,b);
    return 0;
}

push   ebp
mov    ebp,esp
sub    esp,0x18
mov    DWORD PTR [ebp-0xc],0x1
mov    DWORD PTR [ebp-0x8],0x2
mov    eax,DWORD PTR [ebp-0x8]
mov    DWORD PTR [esp+0x4],eax
mov    eax,DWORD PTR [ebp-0xc]
mov    DWORD PTR [esp],eax
call   0x80483b4 <dodaj>
mov    DWORD PTR [ebp-0x4],eax
mov    eax,0x0
leave
ret
```




```
int dodaj(int x, int y)
{
    int z = 0;
    return x+y+z;
}

push   ebp
mov    ebp,esp
sub    esp,0x10
mov    DWORD PTR [ebp-0x4],0x0
mov    eax,DWORD PTR [ebp+0xc]
mov    edx,DWORD PTR [ebp+0x8]
lea    eax,[edx+eax*1]
add    eax,DWORD PTR [ebp-0x4]
leave
ret
```



```
int main  () {
    int a = 1;
    int b = 2
    int c =  dodaj(a,b);
    return 0;
}

push   ebp
mov    ebp,esp
sub    esp,0x18
mov    DWORD PTR [ebp-0xc],0x1
mov    DWORD PTR [ebp-0x8],0x2
mov    eax,DWORD PTR [ebp-0x8]
mov    DWORD PTR [esp+0x4],eax
mov    eax,DWORD PTR [ebp-0xc]
mov    DWORD PTR [esp],eax
call   0x80483b4 <dodaj>
mov    DWORD PTR [ebp-0x4],eax
mov    eax,0x0
leave
ret
```



Przekazywanie przez referencje i wskaźnik

Przez referencje

```
int funkcja(int &a, int &b)
{
   return a+b;
}

int main()
{
   int x = 5;
   int y = 6;
   int wynik = funkcja(x,y);
   return 0;
}
```

Przez wskaźnik

```
int funkcja(int *a, int *b)
{
   return *a+*b;
}

int main()
{
   int x = 5;
   int y = 6;
   int wynik = funkcja(&x,&y);
   return 0;
}
```

Przekazywanie przez referencje I wskaźnik

Funkcja `main`

Przez referencje

```
080484a6 <main>:
push   ebp
mov    ebp,esp
sub    esp,0x18
mov    DWORD PTR [ebp-0x8],0x1
mov    DWORD PTR [ebp-0xc],0x2
lea    eax,[ebp-0xc]
mov    DWORD PTR [esp+0x4],eax
lea    eax,[ebp-0x8]
mov    DWORD PTR [esp],eax
call   8048494 <dodaj>
mov    DWORD PTR [ebp-0x4],eax
mov    eax,0x0
leave
ret
```

Przez wskaźnik

```
080484a6 <main>:
push   ebp
mov    ebp,esp
sub    esp,0x18
mov    DWORD PTR [ebp-0x8],0x1
mov    DWORD PTR [ebp-0xc],0x2
lea    eax,[ebp-0xc]
mov    DWORD PTR [esp+0x4],eax
lea    eax,[ebp-0x8]
mov    DWORD PTR [esp],eax
call   8048494 <_Z5dodajPiS_>
mov    DWORD PTR [ebp-0x4],eax
mov    eax,0x0
leave
ret
```

Przekazywanie przez referencje i wskaźnik

Funkcja `main`

Przez referencje

```
08048494 <dodaj>:
push   ebp
mov    ebp,esp
mov    eax, DWORD PTR [ebp+0x8]
mov    edx, DWORD PTR [eax]
mov    eax, DWORD PTR [ebp+0xc]
mov    eax, DWORD PTR [eax]
lea    eax, [edx+eax*1]
pop    ebp
ret
```

Przez wskaźnik

```
08048494 <_Z5dodajPiS_>:
push   ebp
mov    ebp,esp
mov    eax,DWORD PTR [ebp+0x8]
mov    edx,DWORD PTR [eax]
mov    eax,DWORD PTR [ebp+0xc]
mov    eax,DWORD PTR [eax]
lea    eax,[edx+eax*1]
pop    ebp
ret
```

Zmienna liczba parametrów funkcji

C++

```
#include<stdarg.h> //

int funkcja(int zmiennaNazwana1, char zmiennaNazwana2, …) {
   va_list args;
   va_start(args, zmiennaNazwana2);
   …
   char ch = va_arg(args, char);
   …
   char dp = va_arg(args, char);
   …
   va_end(args);
}
```

C#

```
void funkcja(params string[] vars) //
{
   for (int i = 0; i < vars.Length; i++)
     Console.WriteLine(vars[i].ToString());
}
```

Wstęp do programowania funkcyjnego

Programowanie imperatywne

Programy imperatywne składają się z ciągu instrukcji, które modyfikują stan programu, przechowywany pod postacią zmiennych.

Typowe konstrukcje występujące w programowaniu imperatywnym:

- instrukcje m.in.
  - przypisania
  - pętli
  - skoku
- procedury

Efekt uboczny

Efekt uboczny – podczas obliczania wartości wyrażenia lub funkcji zmienia się stan programu.

3 + 4

i = 3 + 4

Efekt uboczny

```
#include<iostream>
using namespace std;

int b = 4;

int main()
{
  int x = funkcja();
  int y = b+x;
  system("pause");
}
```

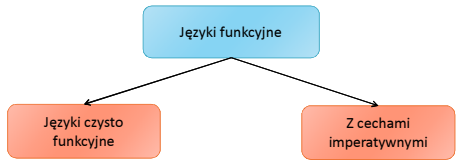
```
int funkcja()
{
  int a = 3;
  int b = 6;
  return a+b;
}
```

```
int funkcja()
{
  int a = 3;
  b = 6;
  return a+b;
}
```

```
#include<iostream>
using namespace std;

int b = 4;

int funkcja() {
  int a = 3;
  b += 2;
  return a+b;
}

int funkcja2() {
  int a = 3;
  return a+b;
}

int main()
{
  cout<<funkcja()+funkcja2()<<endl;
  return 0;
}
```


Programowanie funkcyjne

Języki funkcyjne definiują wynik działania programu, jako funkcję matematyczną parametrów wejściowych, bez stanu wewnętrznego, a więc również bez efektów ubocznych.



Miranda, Haskell, Sisal, pH

Lisp, Scheme, ML

Programowanie funkcyjne

Programowanie w języku funkcyjnym polega tworzeniu coraz bardziej złożonych wyrażeń i definiowaniu stałych, których wartość musi być obliczona.

- bazuje na matematycznej koncepcji funkcji
- podstawową operacją jest podstawienie funkcji
- wykonanie programu polega na obliczaniu wartości kolejnych funkcji.
- w językach tych nie istnieją zmienne
- funkcje są traktowane jako wartości pierwszej klasy, czyli mogą być przekazywane do i z innych funkcji.
- podstawowymi typami złożonym są para i lista
- Garbage Collection

Funkcje w Scheme

```
(define (stopnie->radiany stopnie)
    (* 3.14 (/ stopnie 180)))
```

```
>(stopnie->radiany 180)
3.14
```


Funkcje jako wartości pierwszej klasy

```
int funkcja(int (*wsk)(int, int))
{
  …
  int wynik = wsk(x,y);
  …
}

int dodawanie(int x, int y)
{
  return x+y;
}

int main()
{
   funkcja(dodawanie);
}
```


Funkcje jako wartości pierwszej klasy

```
int dodawanie(int x, int y, int z)
{
   return x+y+z;
}
```

```
int dodawanie(int x, int y)
{
   int z;
   cin >> z;
   return x+y+z;
}
```

Funkcje jako wartości pierwszej klasy

```
(define (funkcja fun)
  (fun 2 3))

>(funkcja +)
5
>(funkcja *)
6
```

```
(define (funkcja1 z)
  (lambda(x y) (+ x y z))

>(funkcja (funkcja1 3))
```


Domknięcie (closure)

Rozważmy następujący fragment kodu JavaScript

```
var zG = 1;

function FunkcjaZewnetrzna(pZ)
{
    var zlZ = 2;

    return function(pW)
    {
      var zlW = 3;
      return zG + zlZ + zlW + pZ + pW;
    }

}
```


Domknięcie (closure)

Wywołanie tych funkcji można przedstawić np. w sposób

```
var f = FunkcjaZewnetrzna(4);
alert(f(5));
```

```
var zG = 1;

function FunkcjaZewnetrzna(pZ)
{
    var zlZ = 2;

    return function(pW)
    {
      var zlW = 3;
      return zG + zlZ + zlW + pZ + pW;
    }

}
```

Domknięcie (closure)

Domknięciem (closure) w językach programowania nazywa się funkcję pierwszej klasy połączoną z leksykalnym środowiskiem tej funkcji przy czym środowisko to zawiera odniesienia do wszystkich symboli niezdefiniowanych w tej funkcji.

Z punktu widzenia implementacji języka wykorzystującego domknięcia, domknięciem może to być struktura, która zawiera wskaźnik do kodu funkcji oraz reprezentację środowiska (zbiór zmiennych i ich wartości) w czasie, utworzenia dopełnienia.

Operacje we/wy w programowaniu funkcyjnym

Operacje wejścia/wyjścia są traktowane jako typowy przykład operacji ze skutkiem ubocznym.

W językach czysto funkcyjnych operacje we/wy są najczęściej izolowane od reszty programu.

Nie modyfikują one stanu zmiennych, a jedynie wewnętrzny stan strumienia znaków, z którego program pobiera lub zapisuje dane.

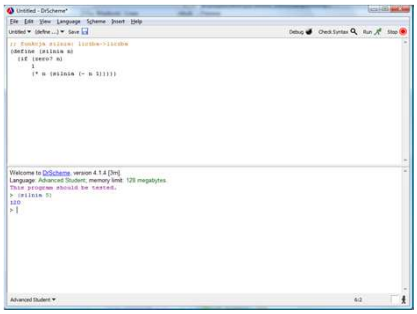
Język Scheme

Środowisko DrRacket

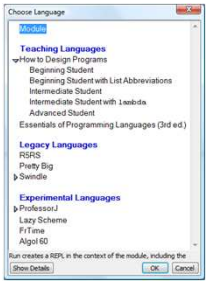
Środowisko DrRacket jest środowiskiem programistycznym wykorzystującym różne dialekty języka Scheme.

- środowisko darmowe (można je pobrać ze strony http://racket-lang.org/)
- występuje w wersjach dla windows, linux, mac
- programowanie w Scheme jest zabawne choć nie zawsze proste
- środowisko ma duże możliwości

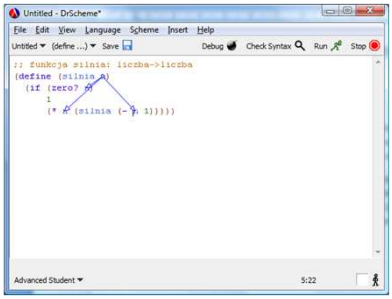
Środowisko DrRacket



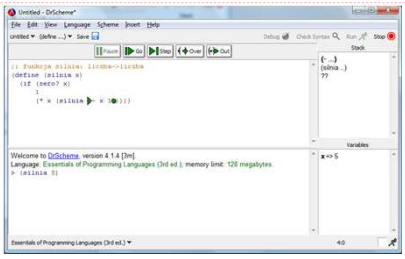
Środowisko DrRacket



Środowisko DrRacket

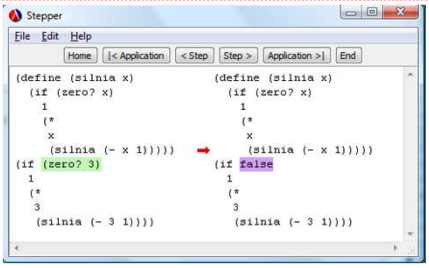


Środowisko DrRacket



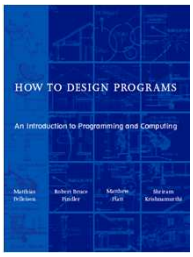
Opcja **niedostępna** w następujących wersjach Scheme:
- Beginning student …
- Intermediate student …

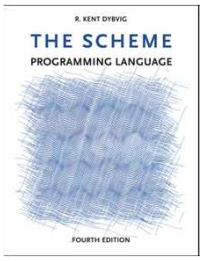
Środowisko DrRacket



Opcja **dostępna tylko** w następujących wersjach Scheme:
- Beginning student …
- Intermediate student …

Literatura





http://htdp.org http://www.scheme.com/tspl4/

Język Scheme

- Jest odmianą języka Lisp (List Processor)
- Podstawową cechą jest minimalizm
- Nie jest językiem czysto funkcyjnym – pozwala na efekty uboczne obliczeń, jak również na tworzenie programów w stylu proceduralnym i obiektowym

Proste typy danych w Scheme

- logiczny: #t, #f
- numeryczne:
  - liczby złożone : 2+3i
  - całkowity: 1
  - rzeczywisty: 1.1
  - ułamkowy: 1/2

  - dokładne
  - niedokładne
- znakowy (unicode)
- łańcuchy znaków (przedstawiane jako tablica o ustalonej długości

Predykaty typów

- string?
- char?
- boolean?
- integer?
- real?
- complex?
- rational?

```
>(string? ”Ala”)
#t
>(complex? 1)
#t
>(real? 1)
#t
>(integer? 2.2)
#f
```


Składnia Scheme

Praktycznie wszystkie wyrażenia w języku Scheme przedstawiane są w postaci:

(<id>/<operator> <wyrażenie>*)

Np.

```
(+ 3 4)
> 7
(+ [* 1 2] {sqrt 4})
> 4
```


Identyfikatory

W języku Scheme identyfikatory można tworzyć praktycznie ze wszystkich znaków jakie można wprowadzić z klawiatury, przy czym identyfikator nie może być liczbą.

!@#$%^ *- to może być poprawna nazwa funkcji*

Zalecane jest stosowanie następujących reguł nazewnictwa:

- predykaty powinny kończyć się znakiem ?
- nazwy funkcji, które mogą mieć skutki uboczne powinny kończyć się znakiem !
- nazwy funkcji konwertujących wartości różnych typów zapisuje się jako typ1->typ2

Definiowanie identyfikatorów

Wyrażenie w postaci:

(define <id> <wyrażenie>)

przypisuje do identyfikatora <id> do wyniku wyrażenia <wyrażenie>

```
>(define trzy (+ 1 2))
>trzy
3
```


Wyrażenia warunkowe if

(if <warunek>
<wyr_prawda>
<wyr_fałsz>)

```
>(define x 3)
>(if (= x 3) (+ x x) (* x x))
6
```


Wyrażenie warunkowe cond

(cond
(<warunek_1> <wyrażenie_1>)
(<warunek_2> <wyrażenie_2>)
…
(else <wyrażenie_else>))

(if <warunek_1> <wyrażenie>
(if <warunek_2> <wyrażenie> <wyrażenie_else>))

Wyrażenie case

(case <wyrażenie>
  ((<wartosc_1>) <wyrażenie_1>)
  ((<wartosc_2>) <wyrażenie_2>)
  …
  (else <wyrażenie_else>))

Funkcje logiczne

- and

```
>(and #t #f)
#f
```

- or

```
>(or #t #f)
#t
```

- not

```
>(not #f)
#f
```


Wyrażenia lambda

Wyrażenia lambda tworzy się za pomocą specjalnego operatora `lambda`:

(lambda (<param>*) <wyrażenie>)

```
>(lambda (x y) (+ x y))
#<procedure>
>((lambda (x y) (+ x y)) 1 2)
3
```

Definiowanie funkcji

Funkcje nazwane tworzy się za pomocą połączenia definicji identyfikatora i wyrażenia lambda:

(define <id>
  (lambda (<param>*) <wyrażenie>))

Np.

```
>(define kwadrat (lambda (x) (* x x))
>(kwadrat 12)
144
```


Definiowanie funkcji

Dodatkowo w Scheme istnieje prostsza składnia definicji funkcji

(define (<id> <param>*) <wyrażenie>)

Np.

```
>(define (kwadrat x) (* x x))
>(kwadrat 12)
144
```


Rekurencja

```
(define (! n)
   (if (= n 1)
      1
      (* n (! (- n 1)))))
```

```
>(! 100)
933262154439441526816992388562667004907159682
643816214685929638952175999932299156089414639
761565182862536979208272237582511852109168640
000000000000000000000000
```

Identyfikatory lokalne

```
(define (! n) (if (= n 1) 1
                  (* n (! (- n 1)))))

(define (f x n) (/ (+ x (! n)) (! n)))


>(f 1 20000)
```


Identyfikatory lokalne

(let (
(<id1> <wyrażenie_1>)
(<id2> <wyrażenie_2>)
…
)
<ciało_let>)

Identyfikatory lokalne

```
(define (! n) (if (= n 1) 1
                  (* n (! (- n 1)))))

(define (f x n)
   (let ((n! (! n)))
     (/ (+ x n!) n!)))


>(f 1 20000)
```

Identyfikatory lokalne

```
(define f
   (let ((x 2)
         (z x))
     (* z x)))
```

x: name is not defined, not a parameter, and not a primitive name

```
(define x 20)
(define f
   (let ((x 2)
         (z x))
     (* z x)))
```


Identyfikatory lokalne

(let* (
(<id1> <wyrażenie_1>)
(<id2> <wyrażenie_2>)
…
)
<ciało_let>)

```
(define f
   (let* ((x 2)
          (z x))
     (* z x)))
```


Identyfikatory lokalne

(letrec (
(<id1> <wyrażenie_1>)
(<id2> <wyrażenie_2>)
…
)
<ciało_let>)

```
>(letrec  ((is-even? (lambda (n)
             (or (zero? n) (is-odd? (sub1 n)))))
           (is-odd? (lambda (n)
             (or (= n 1) (is-even? (sub1 n)))))
      )
   (is-odd? 11))
#t
```

Pary

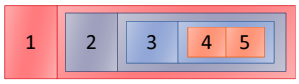
(cons <pierwszy_element> <drugi_element>)

(car <para>) – zwraca pierwszy element pary
(cdr <para>) – zwraca drugi element pary

```
>(define para (cons 1 2))
>para
(1 . 2)
>(car para)
1
>(cdr para)
2
```


Listy

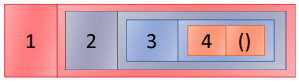
```
(cons 1 (cons 2 (cons 3 (cons 4 5))))
```




```
>(define lista (cons 1 (cons 2 (cons  3 (cons 4 5)))))
>lista
(1 2 3 4 . 5)
>(car ( cdr (cdr lista)))
3
>(caddr lista)
3
```


Listy

```
(cons 1 (cons 2 (cons 3 (cons 4 empty))))
```




```
>(define lista (cons 1 (cons 2 (cons  3 (cons 4 empty)))))
(1 2 3 4)
>(car ( cdr (cdr lista)))
3
>(caddr lista)
3
>(cddddr lista)
()
```

Listy

(list <wartosc>+)

```
(list 1 2 3 4)
```

```
(cons 1 (cons 2 (cons 3 (cons 4 empty))))
```

'(<wartosc>+)

```
'(1 2 3 4 5)
```


Listy - funkcje

- (list? <lista>)
- (empty? <lista>)
- (length <lista>)
- (list-ref <lista> <pozycja>)
- (append <lista> <element>)
- (revese <lista>)
- (map <funkcja> <lista>+)
- (filter <predykat> <lista>)
- (remove <wartosc> <lista> <funkcja_porownujaca>)
- (sort <lista> <funkcja_porownujaca_mniejsze_niz>)

Listy - funkcje - map

```
>(define (podwoj x) (+ x x))
>(define lista (list 1 2 3 4 5 6))
>(map podwoj lista)
(2 4 6 8 10 12)

>(map + lista)
(1 2 3 4 5 6)

>(map + lista lista)
(2 4 6 8 10 12)

>(map + lista lista lista)
(3 6 9 12 15 18)
```

Programy jako listy

```
>(define program '(+ 1 2))
>program
(+ 1 2)

>(eval program)
3
```


Programy jako listy

```
>([lambda (arg)
    (list arg (list ‘quote arg))]
   ‘(lambda (arg) (list arg (list ‘quote arg))))

((lambda (arg)
    (list arg (list ‘quote arg)))
   ‘(lambda (arg) (list arg (list ‘quote arg))))
```


struktury

```
(define-struct  <nazwa_struktury>
    (<składowa_struktury_1>
     <składowa_struktury_2>
     …
     <składowa_struktury_n>))

(make-nazwa_struktury <składowa>*)
(nazwa_struktury-składowa <struktura>)
(nazwa_struktury? <struktura>)
```

struktury

```
>(define-struct punkt (x y))
>(define P (make-punkt 12 12))
>(punkt-x P)
12
>(punkt-y P)
12
>(punkt? P)
true
>(punkt? 12)
false
```


Rachunek lambda

Rachunek lambda

Rachunek Lambda:

- zdefiniowany przez Alonzo Churcha w latach 30 XX w. jako alternatywna konwencja zapisywania funkcji matematycznych
- prosty język programowania z kilkoma konstrukcjami składniowymi i prostą semantyką
- stanowi teoretyczne podstawy języków funkcyjnych
- udowodnione zostało, że moc wyrazu rachunku lambda jest taka sama jak moc wyrazu maszyny Turinga

Wyrażenia i funkcje w rachunku lambda

Do zapisu wyrażeń w rachunku lambda wykorzystywana jest notacja prefiksowa (tzw. notacja polska):

** x 2*
+(tan x) 8

(+ (12 10) (/ 123 15))*

Wyrażenia i funkcje w rachunku lambda

Funkcje w rachunku lambda (lambda abstrakcję) zapisuje się w postaci:

λx. x 2*

```
function f(x : int) : int begin f := x * 2; end;
```
*λ x . * x 2*

Wyrażenia i funkcje w rachunku lambda

Funkcje wielu zmiennych w rachunku lambda zapisuje się w postaci:

*λyx. + y * x 2*

W przypadku implementacji komputerowej częściej korzysta się z innej notacji tzw. ***Curring*** pozwalającej traktować wszystkie funkcje jako funkcje jednej zmiennej

*λy.(λx.(+ y * x 2))*

Wyrażenia i funkcje w rachunku lambda

Aplikację funkcji w rachunku lambda przedstawia się jako proste zestawienie:

$$F\ x$$

$$(\lambda yx.\ +\ y\ *\ x\ 2)\ 3\ 4$$

Składania rachunku lambda

```
<wyrażenie> ::=
|    <stała>
|    <zmienna>
|    (<wyrażenie> <wyrażenie>)
|    λ<zmienna>.<wyrażenie>
```


Zmienne wolne i związane

Zmienną związaną nazywa się zmienną występuje w ciele funkcji, w której jest zdefiniowana.

Zmienną wolną nazywa się zmienną, która nie jest związana.

$$\lambda y.(\ \underbrace{\lambda x.(\ +\ y\ *\ x\ 2\ )}\ )$$

$$\lambda x.(\ \underbrace{+\ y\ *\ x\ 2}\ )$$

$$+\ y\ *\ x\ 2$$

Zmienne wolne i związane

Zbiór zmiennych wolnych zdefiniowany jest jako:

$FV(x) = \{x\}$
$FV(M\ N) = \{FV(M) \cup FV(N)\}$
$FV(\lambda x.M) = \{FV(M) \setminus \{x\}\}$

Zbiór zmiennych związanych zdefiniowany jest jako:

$BV(x) = \{\emptyset\}$
$BV(M\ N) = \{BV(M) \cup BV(N)\}$
$BV(\lambda x.M) = \{BV(M) \cup \{x\}\}$

Wyrażenie jest domknięte, jeżeli nie posiada zmiennych wolnych.

β - redukcja

β-redukcją nazywamy proces aplikacji funkcji $\lambda x.M$ do argumentu N, w taki sposób, że każda wolna zmienna wyrażenia M pozostaje wolna po tej operacji

$$(\lambda x.M)\ N \rightarrow_\beta M[x:=N]$$

$$(\lambda x.*\ x\ 4)\ 5 \rightarrow_\beta (*\ 4\ 5) \rightarrow_\beta 20$$

β-redex ***reductum***

β - redukcja i postać normalna

Jeżeli $M \rightarrow_\beta M'$ to zachodzi

- $MN \rightarrow_\beta M'\ N$
- $NM \rightarrow_\beta NM'$
- $\lambda x.M \rightarrow_\beta \lambda x.M'$

Wyrażenie jest w ***postaci normalnej***, jeżeli nie ma możliwości aby dalej go redukować.

$$(\lambda x.*\ x\ 4)\ 5 \rightarrow_\beta (*\ 4\ 5) \rightarrow_\beta 20$$

β - redukcja

kwad ≡ $\lambda f.\ \lambda x.[f\ (f\ x)]$

(kwad kwad) $= (\{\lambda f.\ \lambda x.[f\ (f\ x)]\}$ kwad)
$\rightarrow_\beta \lambda x.[$ kwad $(\ $kwad $x)]$
$= \lambda x.\{$ kwad $[\{\lambda f.\ \lambda x.[f\ (f\ x)]\}\ x]\}$
$\rightarrow_\beta \lambda x.\{$ kwad $\lambda x.[x\ (x\ x)]\}$
$= \lambda x.\{(\lambda f.\ \lambda x.[f\ (f\ x)]\ \lambda x.[x\ (x\ x)]\}$
$\rightarrow_\beta \lambda x.\{(\lambda x.\{\lambda x.[x\ (x\ x)]\ (\lambda x.[x\ (x\ x)])\ x\})\}$

Gdzie jest błąd w rozumowaniu?

β - redukcja

β-redukcja jest poprawna tylko, gdy zmienne wolne argumentu nie wchodzą w konflikt z żadnym parametrem formalnym w ciele lambda abstrakcji.

α-konwersja

α-konwersja polega na zamianie nazw zmiennych związanych

$$\lambda x.M \rightarrow_\alpha \lambda y.M[x{:=}y] \qquad y \notin \mathrm{FV}(M)$$

$$\lambda x.+\ x\ 2 =_\alpha \lambda y.+\ y\ 2$$

β - redukcja i α-konwersja

(kwad kwad) $= (\lambda f.\ \lambda x.[f\ \{f\ x\}]$ kwad$)$
$\rightarrow_\beta \lambda x.($ kwad $[$ kwad $x])$
$= \lambda x.($ kwad $[\{\lambda f.\ \lambda x.(f\ [f\ x])\}\ x])$
$\rightarrow_\alpha \lambda x.($ kwad $[\{\lambda f.\ \lambda y.(f\ [f\ y])\}\ x])$
$\rightarrow_\beta \lambda x.($ kwad $\lambda y.[x\ \{x\ y\}])$
$= \lambda x.([\lambda f.\ \lambda x.\{f\ (f\ x)\}]\ \lambda y.[x\ \{x\ y\}])$
$\rightarrow_\alpha \lambda x.([\lambda f.\ \lambda z.\{f\ (f\ z)\}]\ \lambda y.[x\ \{x\ y\}])$
$\rightarrow_\beta \lambda x.(\lambda z.[\lambda y.\{x\ (x\ y)\}\ \{(\lambda y.[x\ \{x\ y\}])\ z\}])$
$\rightarrow_\beta \lambda x.(\lambda z.[\lambda y.\{x\ (x\ y)\}\ \{x\ (x\ z)\}])$
$\rightarrow_\beta \lambda x.(\lambda z.[x\ \{x\ (x\ [x\ z])\}])$
$\rightarrow_\alpha \lambda f.(\lambda x.[f\ \{f\ (f\ [f\ x])\}])$

Jedna droga redukcji?

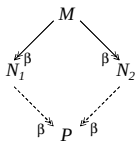
$$(\lambda f.[\lambda x.f\ x]\ a)\ b$$

$(\lambda f.[\lambda x.f\ x]\ a)\ b$
$\rightarrow_\beta (\lambda f.\{f\ a\}\ b)$
$\rightarrow_\beta \{b\ a\}$

$(\lambda f.[\lambda x.f\ x]\ a)\ b$
$\rightarrow_\beta\ [\lambda x.\{b\ x\}\ a]$
$\rightarrow_\beta \{b\ a\}$

Twierdzenie Churcha-Rossera

Jeżeli term M redukuje się do termów N_1 i N_2, za pomocą różnych β-redukcji, to istnieje term P do którego można zredukować termy N_1 i N_2



Operacje logiczne w rachunku lambda

Podstawowe wartości logiczne prawda i fałsz definiowane są w następujący sposób:

$$\underline{\textbf{true}} \equiv \lambda xy.x$$
$$\underline{\textbf{false}} \equiv \lambda xy.y$$

a operacje:

$$\underline{\textbf{if}} \equiv \lambda bxy.bxy$$
$$\underline{\textbf{not}} \equiv \lambda b.\underline{\textbf{if}}\ b\ \underline{\textbf{false}}\ \underline{\textbf{true}}$$
$$\underline{\textbf{and}} \equiv \lambda b_1 b_2.\underline{\textbf{if}}\ b_1\ b_2\ \underline{\textbf{false}}$$
$$\underline{\textbf{or}} \equiv \lambda b_1 b_2.\underline{\textbf{if}}\ b_1\ \underline{\textbf{true}}\ b_2$$

Numerały Churcha

Dowolną liczbę naturalną w rachunku lambda przedstawia się za pomocą następującego funkcji:

$$\underline{c_n} \equiv \lambda fx.f^n(x)$$

$$\underline{0} \equiv \lambda fx.x$$
$$\underline{1} \equiv \lambda fx.f\,x$$
$$\underline{2} \equiv \lambda fx.f(f\,x)$$
$$\underline{3} \equiv \lambda fx.f(f(f\,x))$$
$$\ldots \quad \ldots \quad \ldots$$
$$\underline{c_n} \equiv \lambda fx.f^n(x)$$

Następniki

Zakładając, że n jest numerałem Churcha to jego ***następnik*** jest zdefiniowany za pomocą wzoru:

$$\underline{\textbf{succ}} \equiv \lambda nfx.f(n\,f\,x)$$

Numerały Churcha mogą być również zdefiniowane za pomocą następników:

$$\underline{0} \equiv \lambda fx.x$$
$$\underline{1} \equiv \underline{\textbf{succ}}\ \underline{0}$$
$$\underline{2} \equiv \underline{\textbf{succ}}\ \underline{1}$$
$$\underline{3} \equiv \underline{\textbf{succ}}\ \underline{2}$$
$$\ldots \quad \ldots \quad \ldots$$

Podstawowe funkcje arytmetyczne

Podstawowe operacje arytmetyczne w rachunku lambda są zdefiniowane następująco:

$$\underline{\textbf{add}} \equiv \lambda mnfx.mf(nfx)$$
$$\underline{\textbf{mult}} \equiv \lambda mnfx.m(nf)x$$
$$\underline{\textbf{exp}} \equiv \lambda mnfx.mnfx$$

Funkcja k-arg. stale równa zero: $\underline{\mathbf{Z}}_n \equiv \lambda m_1,\ldots m_k.0$
Projekcja funkcji k-arg. i-tą na współrzędną: $\mathbf{\Pi}_k^i \equiv \lambda m_1,\ldots m_k.m_i$

Pary

Para w rachunku lambda definiowana jest następująco:

$$\underline{\textbf{pair}} \equiv \lambda bxy.\underline{\textbf{if}}\ b\ x\ y$$

Do odczytania składowych pary można zastosować funkcje:

$$\textbf{fst} \equiv \lambda p.p\ \underline{\textbf{true}}$$
$$\textbf{snd} \equiv \lambda p.p\ \underline{\textbf{false}}$$

Krotki i listy

Krotka (jako uogólnienie pary):

$$(M_1,M_2,\ldots,M_k) \equiv \lambda x.x\ M_1,M_2,\ldots,M_k$$

Wybór i-tego elementu krotki: $\underline{\mathbf{sel}}_k^i\ \lambda p.p\ \mathbf{\Pi}_k^i$

Lista:

$$(E_1,E_2,\ldots,E_k) \equiv (E_1, (E_2,(E3, (\ldots, (E_{n-1}, \mathrm{E}_n)\ldots)))$$

Poprzedniki i odejmowanie

Przyjmijmy, że dana jest funkcja pomocnicza *h*:

$$\underline{\mathbf{h}} \equiv \lambda p.\underline{\mathbf{pair}}(\underline{\mathbf{snd}}\ p)(\underline{\mathbf{succ}}\ (\underline{\mathbf{snd}}\ \mathrm{p}))$$

Zakładając, że *n* jest numerałem Churcha to jego ***poprzednik*** jest zdefiniowany za pomocą funkcji:

$$\underline{\mathbf{pred}} \equiv \lambda n.\underline{\mathbf{fst}}(n\ h\ (\underline{\mathbf{pair}}\ \underline{0}\ \underline{0}))$$

Poprzedniki pozwalają na zdefiniowanie operacji odejmowania, która w rachunku lambda przedstawiana jest funkcją:

$$\underline{\mathbf{sub}} \equiv \lambda mn.m\ \underline{\mathbf{pred}}\ n$$

Programowanie w logice PROLOG

Programowanie w logice

Każdy student informatyki na PCz lubi Paradygmaty Programowania.

Tomek Nowak jest studentem informatyki na PCz.

Tomek Nowak lubi Paradygmaty Programowania!

Programowanie w logice

"Ideą programowania w logice jest wykorzystanie komputera do wyprowadzenia konkluzji z deklaratywnych opisów "

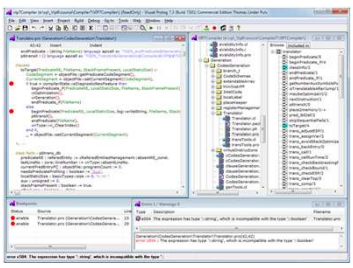
Ulf Nillsson, Jan Małuszyński "Logic, Programming and Prolog (2ED)"

PROLOG

- Język programowania stworzony w 1972 r. przez Alain Colmerauer'a i Phillipe Roussel'a, pierwotnie do rozwiązywania zagadnień związanych z przetwarzaniem języka naturalnego
- Jest realizacją paradygmatu programowania w logice
- Jest językiem deklaratywnym – pozwala na deklarowanie pewnych założeń dotyczących rozwiązywanego problemu

Implementacje

Visual Prolog



Oprogramowanie

SWI-Prolog

Podstawowe elementy języka PROLOG

- ***Fakty*** – podstawowa informacja na temat pewnego "świata" (analizowanego zagadnienia)
- ***Reguły*** – zdania łączące fakty umożliwiające przeprowadzenie wnioskowanie w analizowanym "świecie"

Baza wiedzy

- ***Zapytania*** – wydawane w celu uzyskania pewnych informacji na temat analizowanego świata

Fakty

Fakty – podstawowa informacja na temat pewnego "świata" (analizowanego zagadnienia)

```
mezczyzna(jan).
mezczyzna(tomasz).
kobieta(alicja).
kobieta(filomena).

rodzic(filomena, tomasz).
rodzic(filomena, alicja).
rodzic(jan, tomasz).
rodzic(jan, alicja).
```

Reguły

Reguły – zdania łączące fakty umożliwiające przeprowadzenie wnioskowania w analizowanym "świecie"

```
ojciec(X, Y)  :- rodzic(X, Y), mezczyzna(X).
matka(X, Y)   :- rodzic(X, Y), kobieta(X).
dziadek(X, Y) :- ojciec(X,Z), ojciec(Z,Y).
```


Zmienne w PROLOG-u

Prolog jest językiem o dynamicznym systemie typów.

Zmienne w Prologu zawsze rozpoczynają się od dużej litery.

```
mezczyzna(jan).
mezczyzna(tomasz).

ojciec(X, Y)  :- rodzic(X, Y), mezczyzna(X).
```


Zapytania

Zapytania – wydawane w celu uzyskania pewnych informacji na temat analizowanego świata

Podstawowe zapytania pozwalają potwierdzić "przypuszczenie" na podstawie faktów zapisanych w bazie wiedzy.

```
mezczyzna(jan).
mezczyzna(tomasz).
kobieta(alicja).
kobieta(filomena).

rodzic(filomena, tomasz).
```

```
?- mezczyzna(jan).
true

?- mezczyzna(teofil).
false
```

Zapytania

```
mezczyzna(jan).
mezczyzna(tomek).
kobieta(ewelina).
kobieta(maria).
matka(maria, tomek).
matka(ewelina, jan).
ojciec(tomek, jan).

?- mezczyzna(tomek).
true
? - matka(ewelina, jan).
true
?- matka(ewelina, tomek).
false
```


Zapytania

```
mezczyzna(jan).
mezczyzna(tomasz).
kobieta(alicja).
kobieta(filomena).

rodzic(filomena, tomasz).
rodzic(alicja, filomena).
rodzic(jan, tomasz).

?- mezczyzna(Kto).
Kto = jan ;
Kto = tomasz.

?- rodzic(Kto, tomasz).
Kto = filomena ;
Kto = jan.
```


Zapytania złożone

Zapytania o pojedyncze fakty można łączyć za pomocą operatorów logicznych.

Operator logiczny	Operator Prologu
⇐ (implikacja)	:-
∧ (koniunkcja)	,
∨ (alternatywa)	;

```
?- lubi(jan, aneta), lubi(tomek, aneta)
no
?- lubi(jan, X), lubi(tomek, X)
X=muzyka
```

Zapytania z regułami

```
mezczyzna(jan).
mezczyzna(tomasz).
mezczyzna(adam).
mezczyzna(tadeusz).
mezczyzna(maurycy).
mezczyzna(antoni).

rodzic(adam, tadeusz).
rodzic(adam, antoni).
rodzic(tadeusz, jan).
rodzic(jan, tomasz).
rodzic(antoni, maurycy).

ojciec(X, Y)  :- rodzic(X, Y), mezczyzna(X).
matka(X, Y)   :- rodzic(X, Y), kobieta(X).
dziadek(X, Y) :- ojciec(X, Z), ojciec(Z, Y).
```


Zapytania z regułami

```
ojciec(X, Y)  :- rodzic(X, Y), mezczyzna(X).
matka(X, Y)   :- rodzic(X, Y), kobieta(X).
dziadek(X, Y) :- ojciec(X, Z), ojciec(Z, Y).

?- dziadek(adam, tomasz).
```

```
   ojciec(adam, Z), ojciec(Z, tomasz).
```

```
   rodzic(adam, Z), mezczyzna(adam), ojciec(Z, tomasz).
```


Zapytanie z regułami

```
mezczyzna(jan).
mezczyzna(tomasz).
mezczyzna(adam).
mezczyzna(tadeusz).
mezczyzna(maurycy).
mezczyzna(antoni).

rodzic(adam, tadeusz).
rodzic(adam, antoni).
rodzic(tadeusz, jan).
rodzic(jan, tomasz).
rodzic(antoni, maurycy).

   rodzic(adam, Z), mezczyzna(adam), ojciec(Z, tomasz).
```

Z = tadeusz

Arytmetyka w PROLOG-u

Operator	Opis
X=:=Y	Równe
X=\=Y	Różne
X<Y	X mniejsze od Y
X>Y	X większe od Y
X =< Y	X mniejsze lub równe Y
X >= Y	X większe lub równe Y
X+Y	
X-Y	
X*Y	
X/Y	
X//Y	Dzielenie całkowite
X mod Y	Reszta z dzielenia

Równość w Prologu

W Prologu istnieje operator porównania =, ale jego znaczenie jest trochę odmienne niż w innych językach programowania.

```
?- 1 = 1.
true

?- X = Y.
X = Y.

?- X = ala, Y = X.
X = Y, Y = ala.

?- X = Y, Y = Z, X = ala.
X = Y, Y = Z, Z = ala.

?- 3 = (1+2).
false

?- X = (1+2).
X = 1+2.
```


Równość w Prologu

Aby wartość wyrażenia arytmetycznego stojącego po prawej stronie znaku równości została wyznaczona, konieczne jest zastąpienie tego operatora, operatorem **is**.

```
?- X = (1+2).
X = 1+2.

?- X is (1+2).
X = 3.

?- 3 is 1+2.
true.

?- X is Y.
ERROR: is/2: Arguments are not sufficiently instantiated

?- X is ala
ERROR: is/2: Arithmetic: `ala/0' is not a function
```

Arytmetyka w Prologu

```
wlada(rhodri, 874, 878).
wlada(anarawd, 878, 916).
wlada(hywel_ap_idwal, 916, 950).
wlada(lago_ap_idwal, 950, 979).
wlada(hywel_ap_ieauf, 979, 985).
wlada(cadwallon, 985, 986).
wlada(maredudd, 986, 999).
```

```
ludnosc(usa, 203).
ludnosc(indie, 548).
ludnosc(china, 800).
ludnosc(brazylia, 108).

obszar(usa, 3).
obszar(indie, 1).
obszar(chiny, 4).
obszar(brazylia, 3).
```

```
ksiaze(X, Y) :-
   wlada(X, A, B),
   Y >=A,
   Y <= B.
```

```
gestosc(X, Y) :-
   ludnosc(X, A),
   obszar(X, B),
   Y is A/B.
```

W.F. Clocksin, C.S. Mellish, "Prolog Programowanie". Helion

Rekurencja w PROLOG-u

Rekurencja w prologu polega na wykorzystaniu w regule jej samej.

```
silnia(0,W) :- W is 1.

silnia(N,W) :- X is N-1,
               silnia(X, W_N_1),
               W is N*W_N_1.
```


Rekurencja w PROLOG-u

```
?- silnia(4,X).
```

```
silnia(0,W) :- W is 1.
silnia(N,W) :- X is N-1,
               silnia(X, W_N_1),
               W is N*W_N_1.
```

```
   Call: (6) silnia(4, _G496) ? creep
^  Call: (7) _G571 is 4+ -1 ? creep
^  Exit: (7) 3 is 4+ -1 ? creep
   Call: (7) silnia(3, _G569) ? creep
^  Call: (8) _G574 is 3+ -1 ? creep
^  Exit: (8) 2 is 3+ -1 ? creep
   Call: (8) silnia(2, _G572) ? creep
^  Call: (9) _G577 is 2+ -1 ? creep
^  Exit: (9) 1 is 2+ -1 ? creep
   Call: (9) silnia(1, _G575) ? creep
^  Call: (10) _G580 is 1+ -1 ? creep
^  Exit: (10) 0 is 1+ -1 ? creep
   Call: (10) silnia(0, _G578) ? creep
^  Call: (11) _G580 is 1 ? creep
^  Exit: (11) 1 is 1 ? creep
```

```
   Exit: (10) silnia(0, 1) ? creep
^  Call: (10) _G583 is 1*1 ? creep
^  Exit: (10) 1 is 1*1 ? creep
   Exit: (9) silnia(1, 1) ? creep
^  Call: (9) _G586 is 2*1 ? creep
^  Exit: (9) 2 is 2*1 ? creep
   Exit: (8) silnia(2, 2) ? creep
^  Call: (8) _G589 is 3*2 ? creep
^  Exit: (8) 6 is 3*2 ? creep
   Exit: (7) silnia(3, 6) ? creep
^  Call: (7) _G496 is 4*6 ? creep
^  Exit: (7) 24 is 4*6 ? creep
   Exit: (6) silnia(4, 24) ? creep
W = 24 .
```

Listy jako złożony typ danych

- Lista w Prologu przedstawiana jest jako ciąg termów zawartych pomiędzy nawiasami kwadratowymi i oddzielonych przecinkami.
- Nie ma specjalnych funkcji tworzenia list i dostępu do elementów
- W regułach wykorzystuje się notację [X|Y]

```
polacz([], Lista, Lista).
polacz([G|Reszta], Lista, [G|Reszta_Wynik]) :-
        polacz(Reszta, Lista, Reszta_Wynik).

zawiera(Elem, [Elem|_]).
zawiera(Elem, [_|Reszta]) :- zawiera(Elem, Reszta).

odwroc([], []).
odwroc([G|Reszta], Lista) :-
        odwroc(Reszta,Wynik), polacz(Wynik, [G], Lista).
```

1. W pętli for(char i=0; i<256; i++) cout<<i<<endl; instrukcja cout wykona się :

więcej niż 257 razy (char z licznikiem daje over 9000)

2. Funkcja rozszerzająca, jest to funkcja

statyczna

3. Wartości grupowane są w n-tki w:

produktach kartezjańskich

4. Które z poniższych wyrażeń poprawnie definiuje w języku C# wyrażenie lambda

(x,y) => (x y)

5. Blok programu jest to konstrukcja

Ograniczająca zakres widoczności deklaracji wewnątrz niego

6. Która, z poniższych instrukcji deklaruje zmienne sekwencyjnie

(let* ((x 2) (a 3)) (+x a))

7. W językach o dynamicznym zakresie widoczności nazw

Funkcja jest wywoływana w kontekście środowiska w którym została wywołana

8. Deklaracje są to instrukcje, które

tworzą nowe wiązania

9. Instrukcja typedef języka C++ powoduje

przypisanie nowej nazwy do istniejącego typu

10. Wyrażenie języka Scheme: '((+ 1 2) (- 2 3) (-3 4))

Listę list symboli +, 1, 2, (); -, 2, 3, (); -, 3, 4, ()

11. C++ jest językiem programowania, w którym system typów:

jest statyczny

12. Które wyrażenie rachunku lambda (wzbogaconego instrukcję iloczynu) będzie miało to samo znaczenie, co instrukcja języka Scheme: (let ((x 2) (y 3)) (* x y))

λx.λy.(* x y) 2 3

13. Wyrażenie ((lambda (z) (lambda (x y) (+ x y z))) 1) zwraca

Funkcję

14. Zakresem widoczności zmiennych w wyrażeniu (let ((x 2) (y 3)) (+ x y)) jest

wyrażenie (+ x y)

15. Smalltalk jest językiem należącym do paradygmatu

obiektowego

16. Wyrażenie języka C#: x=>x+1 tworzy:

Funkcję anonimową

17. Językiem o statycznym systemie typów jest:

C ,ADA, C++, C#, Java, Pascal, Visual Basic

18. Do zwrócenie pierwszego elementu pary w języku Scheme służy funkcja

car

19. Który program języka PROLOG pozwoli poprawnie obliczyć wartość silni:

silnia(0,W) :- W is 1.silnia(N,W) :- X is N-1, silnia(X, W_N_1), W is N*W_N_1.

20. Komenda mezczyzna(jan)? języka PROLOG definiuje

ma błąd składniowy

21. Wyrażenie let* języka Scheme definiuje zmienne

sekwencyjnie

22. Wyrażenie warunkowe w rachunku lambda zapisuje się jako

λstx.stx

23. Wyrażenie let języka Scheme definiuje zmienne

równolegle

24. Przyjmując, że typ int zajmuje w pamięci 4 bajty, a typ char 1 bajt to zmienna typu union Przyklad { int x; char y; }; będzie zajmować

4 bajty

25. Wyrażenie języka C#: x=>x+2 nazywane jest

wyrażeniem lambda

26. Fakty w PROLOGU

są zawsze prawdziwe

27. Słówko kluczowe var w C# pozwala na

zdefiniowanie zmiennej dowolnego typu, która nie może być później zmieniona

28. Zakresem widoczności zmiennych w wyrażeniu (let* ((x 2) (y 3)) (+ x y)) jest

całe wyrażenie (let ((x 2) (y 3)) (+ x y))

29. Który z poniższych języków programowania jest językiem programowania obiektowego

C#

30. Postać normalna jest to

Wyrażenie, którego nie można dalej redukować

31. Alfa konwersja polega na zmianie

nazw zmiennych związanych

32. Operator koniunkcji jest w języku PROLOG oznaczany symbolem

,

33. Jaka funkcja zostanie automatycznie stworzona w chwili deklaracji struktury (define-strukt punkt (x y))

make-punkt

34. Która z poniższych instrukcji poprawnie deklaruje funkcję rozszerzającą:

public static string Funkcja(this string str) {…}

35. Która z poniższych instrukcji jest zawsze poprawna

(letrec ((foo (lambda (x) (bar x))) (bar (lambda (x) (+ 1 x)))) (foo 5))

36. Wynikiem operacji (λnfx.f(n f x) λst.ssst) będzie

λfx.ffffx

37. W rekordzie aktywacji funkcji przechowywane są

zmienne lokalne

38. Do zwrócenie drugiego elementu pary w języku Scheme służy funkcja

cdr

39. Przy przekazywaniu przez wartość argumentów "z" funkcji parametrem aktualnym może być:

zmienna

40. Zakresem widoczności zmiennych w wyrażeniu (letrec ((x 2) (y 3)) (+ x y)) jest

całe wyrażenie (let ((x 2) (y 3)) (+ x y))

41. Dowolna zmienna przechowywana na stercie jest

zawsze anonimowa

42. Wyrażenie letrec języka Scheme definiuje zmienne

rekurencyjnie

43. Zmienna związana jest to

zmienna, która występuje w ciele funkcji w której jest zdefiniowana

44. Program

```
#include <iostream>
using namespace std;
template <typename T>T max(T x, T y) { return (x > y)? x : y;}
int main() {
cout << max(3, 7) << std::endl;
cout << max(3.0, 7.0) << std::endl;
cout << max(3, 7.0) << std::endl;
return 0;
}
```

da w rezultacie

Żadne z powyższych/Błąd kompilacji(niejednoznaczność 3 wywołania)

45. Która, z poniższych instrukcji występuje tylko w paradygmacie imperatywnym i pochodnych

pętli

46. Za pomocą którego wyrażenia języka Scheme uzyskano poniższą strukturę

2 3 ()
1 4 ()
5 ()

(cons 1 (cons (cons 2 (cons 3 empty)) (cons (cons 5 empty) (cons 4 empty))))

47. Które wyrażenie rachunku lambda (wzbogaconego instrukcję iloczynu) będzie miało to samo znaczenie, co instrukcja języka Scheme: (let* ((x 2) (y 3)) (* x y))

λx.[λ y.(* x y)] 2 3

48. Programowanie generyczne pozwala na

napisanie jednego kodu dla różnych typów danych

49. Środowisko odniesienia jest to:

zbiór dostępnych wiązań w danym miejscu programu

50. Wyrażeniu języka C#: (x,y)=>(x*2)+(y*3) w języku Scheme odpowiada wyrażenie

(lambda (x y) (+ (* x 2) (* y 3)))

51. Wyrażenie języka Scheme: '(1 (+ 1 2) 4)

stworzy listę, której drugim elementem będzie lista

52. Deklaracje są to instrukcje, które

Tworzą nowe wiązania

53. Które zdanie jest prawdziwe:

Rachunek Lambda jest podstawą teoretyczną języków funkcyjnych

54. Tablice wielowymiarowe w języku C# (tworzone z pomocą instrukcji np.: var tab = new int[3,5]) są tablicami:

prostokątnymi

55. Które z poniższych wyrażeń jest poprawnym wyrażeniem Linq:

(from x in lista where x > 10 select x)

56. Funkcje wyższych rzędów to funkcje które:

mają co najmniej jeden parametr, którego wartość musi być funkcją lub funkcje zwracają

57. Które z poniższych instrukcji poprawnie deklaruje funkcje szablonowe w języku C++:

template <typename T> T dodaj (T a, T b);

58. Efekty uboczne w językach programowania:

utrudniają analizę działania programu

1. W pętli
   for(char i=0; i<256; i++) cout<<i<<endl;
   instrukcja cout wykona się
   a) 256 razy
   b) 257 razy
   c) więcej niż 257 razy
   d) 255 razy
2. Funkcja rozszerzająca, jest to funkcja
   a) pozwalająca rozszerzyć możliwości już istniejącej metody
   b) funkcja rozszerzająca kolekcję o nowe elementy
   c) funkcja pozwalająca na dostęp do zmiennych prywatnych klasy
   d) statyczna
3. Wartości grupowane są w n-tki w:
   a) produktach kartezjańskich
   b) uniach
   c) typach rekurencyjnych
   d) mapowaniach
4. Które z poniższych wyrażeń poprawnie definiuje w języku C# wyrażenie lambda
   λx.λy.(x y)
   a) (x,y) => y(x)
   b) (x,y) => (x y)
   c) (x,y) => (y x)
   d) (x,y) => x(y)
5. Blok programu jest to konstrukcja
   a) Która może się łączyć tylko z instrukcją warunkową
   b) Która nie istnieje w nowoczesnych językach programowania
   c) Ograniczająca zakres widoczności deklaracji wewnątrz niego
   d) Równoważna definicji klasy
6. Która, z poniższych instrukcji deklaruje zmienne sekwencyjnie
   a) (let* ((x 2) (a 3)) (+x a))
   b) (letrec ((x 2) (a 3)) (+ x a))
   c) (let ((x 2) (a x)) (+ x a))
   d) (let ((x 2) (a 3)) (+ x a))
7. W językach o dynamicznym zakresie widoczności nazw
   a) Funkcja jest wywoływana w kontekście środowiska w którym została zdefiniowana
   b) Funkcja jest wywoływana w kontekście środowiska w którym została wywołana
   c) Funkcji nie ma
   d) Funkcja jest wykonywana w dowolnym środowisku
8. Deklaracje są to instrukcje, które
   a) tworzą nowe funkcje
   b) tworzą nowe zmienne
   c) tworzą nowe stałe
   d) tworzą nowe wiązania
9. Instrukcja typedef języka C++ powoduje
   a) przypisanie nowej nazwy do istniejącego typu
   b) stworzenie nowej tablicy
   c) przypisanie nowej nazwy do istniejącej zmiennej
   d) stworzenie nowego typu danych

10. Wyrażenie języka Scheme: '((+ 1 2) (- 2 3) (-3 4))
    a) Listę wartości: 3, -1, -1, ()
    b) Listę funkcji
    c) Nie wykona się z powodu błędu składniowego
    d) Listę list symboli +, 1, 2, (); -, 2, 3, (); -, 3, 4, ()
11. C++ jest językiem programowania, w którym system typów:
    a) jest statyczny
    b) jest dynamiczny
    c) zawiera tylko typy złożone
    d) nie ma go
12. Które wyrażenie rachunku lambda (wzbogaconego instrukcję iloczynu) będzie miało to samo znaczenie, co instrukcja języka Scheme: (let ((x 2) (y 3)) (* x y))
    a) λx.λy.(* x y) 3 2
    b) λx.[λy.(* x y)] 3 2
    c) λx.[λ y.(* x y)] 2 3
    d) λx.λy.(* x y) 2 3
13. Wyrażenie ((lambda (z) (lambda (x y) (+ x y z))) 1) zwraca
    a) Funkcję
    b) Domknięcie
    c) Nie wykona się z powodu błędu
    d) Wartość liczbową
14. Zakresem widoczności zmiennych w wyrażeniu (let ((x 2) (y 3)) (+ x y)) jest
    a) lista ((x 2) (y 3))
    b) całe wyrażenie (let ((x 2) (y 3)) (+ x y))
    c) wyrażenie (y 3) i (+ x y)
    d) wyrażenie (+ x y)
15. Smalltalk jest językiem należącym do paradygmatu
    a) logicznego
    b) obiektowego
    c) imperatywnego
    d) funkcyjnego
16. Wyrażenie języka C#: x=>x+1 tworzy:
    a) wartość logiczną
    b) wartość całkowitą
    c) nie ma takiego wyrażenie w języku C#
    d) Funckję anonimową
17. Językiem o dynamicznym systemie typów nie jest:
    a) Scheme
    b) C
    c) Java Script
    d) PHP
18. Do zwrócenie pierwszego elementu pary w języku Scheme służy funkcja
    a) cdr
    b) begin
    c) end
    d) car
19. Który program języka PROLOG pozwoli poprawnie obliczyć wartość silni:

```
a) silnia(N,W) :- X is N-1,  silnia(X, W_N_1),  W is N*W_N_1. silnia(0,W) :- W is 1.
b) silnia(0,W) :- W = 1.silnia(N,W) :- X = N-1,  silnia(X, W_N_1),  W = N*W_N_1.
c) silnia(N,W) :- X = N-1,  silnia(X, W_N_1),  W = N*W_N_1. silnia(0,W) :- W = 1.
d) silnia(0,W) :- W is 1.silnia(N,W) :- X is N-1,  silnia(X, W_N_1),  W is N*W_N_1.
```

20. Komenda mezczyzna(jan)? języka PROLOG definiuje
    a) nowe zapytanie
    b) nowy fakt
    c) ma błąd składniowy
    d) nową relację
21. Wyrażenie let* języka Scheme definiuje zmienne
    a) równolegle
    b) rekurencyjnie
    c) których wartość można modyfikować
    d) sekwencyjnie
22. Wyrażenie warunkowe w rachunku lambda zapisuje się jako
    a) λstx.tx
    b) λstx.stx
    c) λstx.sxt
    d) λstx.sx
23. Wyrażenie let języka Scheme definiuje zmienne
    a) równolegle
    b) rekurencyjnie
    c) których wartość można modyfikować
    d) sekwencyjnie
24. Przyjmując, że typ int zajmuje w pamięci 4 bajty, a typ char 1 bajt to zmienna typu union Przyklad { int x; char y; }; będzie zajmować
    a) 4 bajty
    b) 8 bajtów
    c) 6 bajtów
    d) 5 bajtów
25. Wyrażenie języka C#: x=>x+2 nazywane jest
    a) funkcją rozszerzającą
    b) funkcją statyczną
    c) nie ma takiego wyrażenia w języku C#
    d) wyrażeniem lambda
26. Fakty w PROLOGU
    a) są zawsze fałszywe
    b) są albo prawdziwe albo fałszywe
    c) nie mają określonej wartości prawdy
    d) są zawsze prawdziwe
27. Słówko kluczowe var w C# pozwala na
    a) zdefiniowanie dowolnej zmiennej
    b) zdefiniowanie zmiennej dowolnego typu, który nie może być później zmieniany
    c) zdefiniowanie zmiennej typu podstawowego Object
    d) zdefiniowanie zmiennej dowolnego typu, który może być później dowolnie zmieniany
28. Zakresem widoczności zmiennych w wyrażeniu (let* ((x 2) (y 3)) (+ x y)) jest
    a) lista ((x 2) (y 3))
    b) całe wyrażenie (let ((x 2) (y 3)) (+ x y))
    c) wyrażenie (y 3) i (+ x y)
    d) wyrażenie (+ x y)

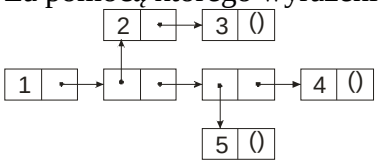
29. Który z poniższych języków programowania jest językiem programowania obiektowego
    a) Haskell
    b) C#
    c) Prolog
    d) ML
30. Postać normalna jest to
    a) Wyrażenie, którego nie można dalej redukować
    b) Wyrażenie, które nie zawiera żadnej lambda abstrakcji
    c) Wyrażenie, które nie zawiera zmiennych związanych
    d) Wyrażenie, które nie zawiera zmiennych wolnych
31. Alfa konwersja polega na zmianie
    a) nazw zmiennych związanych
    b) lambda abstrakcji na aplikację
    c) aplikacji na lambda abstrakcję
    d) nazw zmiennych wolnych
32. Operator koniunkcji jest w języku PROLOG oznaczany symbolem
    a) ,
    b) ;
    c) ?
    d) .
33. Jaka funkcja zostanie automatycznie stworzona w chwili deklaracji struktury (define-strukt punkt (x y))
    a) make-punkt
    b) get-punkt
    c) put-punkt
    d) set-punkt
34. Która z poniższych instrukcji poprawnie deklaruje funkcję rozszerzającą:
    a) public static string Funkcja(string str) {…}
    b) public string Funkcja(this string str) {…}
    c) static string Funkcja(this string str) {…}
    d) public static string Funkcja(this string str) {…}
35. Która z poniższych instrukcji jest zawsze poprawna
    a) (let* ((foo (lambda (x) (bar x))) (bar (lambda (x) (+ 1 x)))) (foo 5))
    b) (letrec ((foo (lambda (x) (bar x))) (bar (lambda (x) (+ 1 x)))) (foo 5))
    c) (let~ ((foo (lambda (x) (bar x))) (bar (lambda (x) (+ 1 x)))) (foo 5))
    d) (let ((foo (lambda (x) (bar x))) (bar (lambda (x) (+ 1 x)))) (foo 5))
36. Wynikiem operacji (nfx.f(n f x) st.ssst) będzie
    a) λst.ssst
    b) λst.sst
    c) λfx.ffffx
    d) λfx.fffx
37. W rekordzie aktywacji funkcji przechowywane są
    a) zmienne lokalne
    b) stałe
    c) żadne z powyższych
    d) zmienne globalne

38. Do zwrócenie drugiego elementu pary w języku Scheme służy funkcja
    a) cdr
    b) begin
    c) end
    d) car
39. Przy przekazywaniu przez wartość argumentów "z" funkcji parametrem aktualnym może być:
    a) stała
    b) zmienna
    c) dowolnie
    d) wyrażenie
40. Zakresem widoczności zmiennych w wyrażeniu (letrec ((x 2) (y 3)) (+ x y)) jest
    a) lista ((x 2) (y 3))
    b) całe wyrażenie (let ((x 2) (y 3)) (+ x y))
    c) wyrażenie (y 3) i (+ x y)
    d) wyrażenie (+ x y)
41. Dowolna zmienna przechowywana na stercie jest
    a) zawsze typu złożonego
    b) zawsze anonimowa
    c) tworzona zawsze na początku programu
    d) zawsze typu prostego
42. Wyrażenie letrec języka Scheme definiuje zmienne
    a) równolegle
    b) rekurencyjnie
    c) których wartość można modyfikować
    d) sekwencyjnie
43. Zmienna związana jest to
    a) zmienna do której jest przypisana wartość
    b) zmienna, która występuje w ciele funkcji w której jest zdefiniowana
    c) która nie istnieje, gdyż istnieją tylko zmienne wolne
    d) zmienna do której przypisana jest inna zmienna
44. Program

```
#include <iostream>
using namespace std;
template <typename T>T max(T x, T y) { return (x > y)? x : y;}
int main() {
    cout << max(3, 7) << std::endl;
    cout << max(3.0, 7.0) << std::endl;
    cout << max(3, 7.0) << std::endl;
    return 0;
}
```

    da w rezultacie
    a) Błąd kompilacji ze względu na brak typu danych w wywołaniu funkcji szblonowej
    b) Błąd kompilacji ze względu na niejednoznaczność trzeciego wywołania funkcji max
    c) Żadne z powyższych
    d) wartości 7, 7.0, 7.0

45. Która, z poniższych instrukcji występuje tylko w paradygmacie imperatywnym i pochodnych
    a) wywołania funkcji
    b) deklaracji
    c) warunkowa
    d) pętli
46. Za pomocą którego wyrażenia języka Scheme uzyskano poniższą strukturę



    a) (cons 1 (cons (cons 2 (cons 3 empty)) (cons (cons 5 empty) (cons 4 empty))))
    b) (list 1 2 3 4 5)
    c) (cons 1 (cons (cons 2 3) (cons 5 4))))
    d) (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 empty)))))
47. Które wyrażenie rachunku lambda (wzbogaconego instrukcję iloczynu) będzie miało to samo znaczenie, co instrukcja języka Scheme: (let* ((x 2) (y 3)) (* x y))
    a) λx.λy.(* x y) 3 2
    b) λx.[λ y.(* x y)] 3 2
    c) λx.[λ y.(* x y)] 2 3
    d) λx.λy.(* x y) 2 3
48. Programowanie generyczne pozwala na
    a) napisanie jednego kodu dla różnych algorytmów
    b) napisanie jednego kodu dla różnych nazw zmiennych
    c) nie ma czegoś takiego
    d) napisanie jednego kodu dla różnych typów danych
49. Środowisko odniesienia jest to:
    a) zbiór dostępnych zmiennych w danym miejscu programu
    b) zbiór dostępnych funkcji w danym miejscu programu
    c) zbiór dostępnych stałych w danym miejscu programu
    d) zbiór dostępnych wiązań w danym miejscu programu
50. Wyrażeniu języka C#: (x,y)=>(x*2)+(y*3) w języku Scheme odpowiada wyrażenie
    a) (define f (lambda (x y) ((x*2)+(y*3))))
    b) (lambda (x y) ((x*2)+(y*3)))
    c) (lambda (x y) (+ (* x 2) (* y 3)))
    d) (define f (lambda (x y) (+ (* x 2) (* y 3))))