

FUNKCYJNE STRUKTURY DANYCH

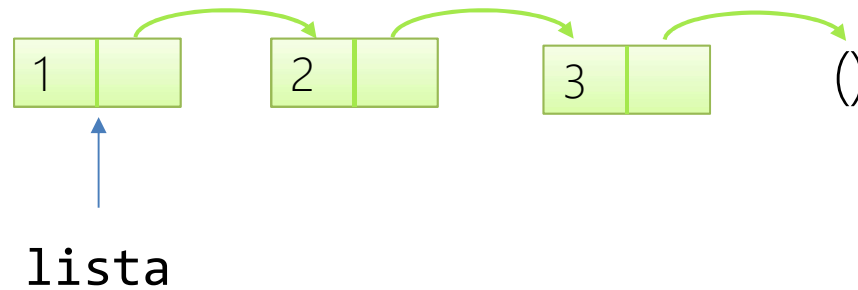
Listy

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



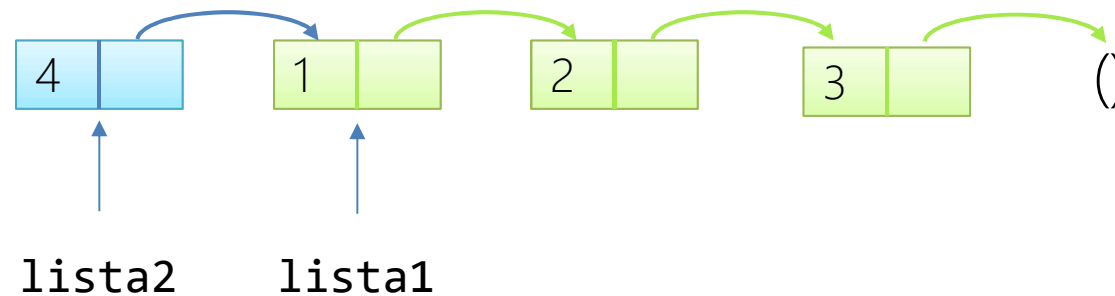
Lista

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



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

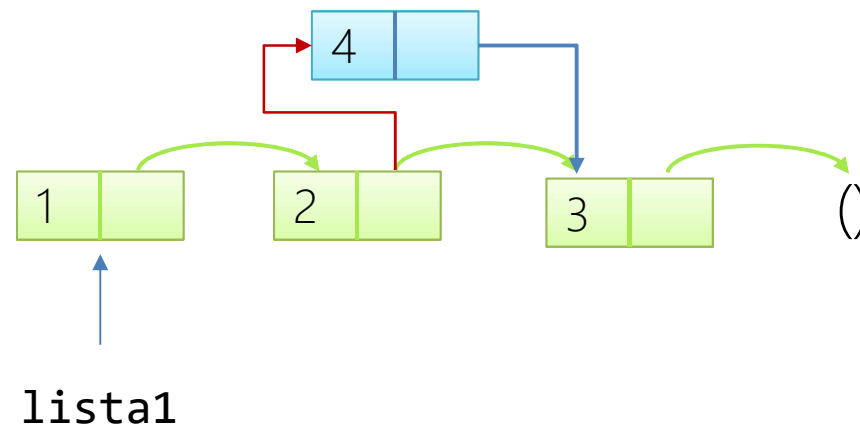
Listy – dodawanie elementu



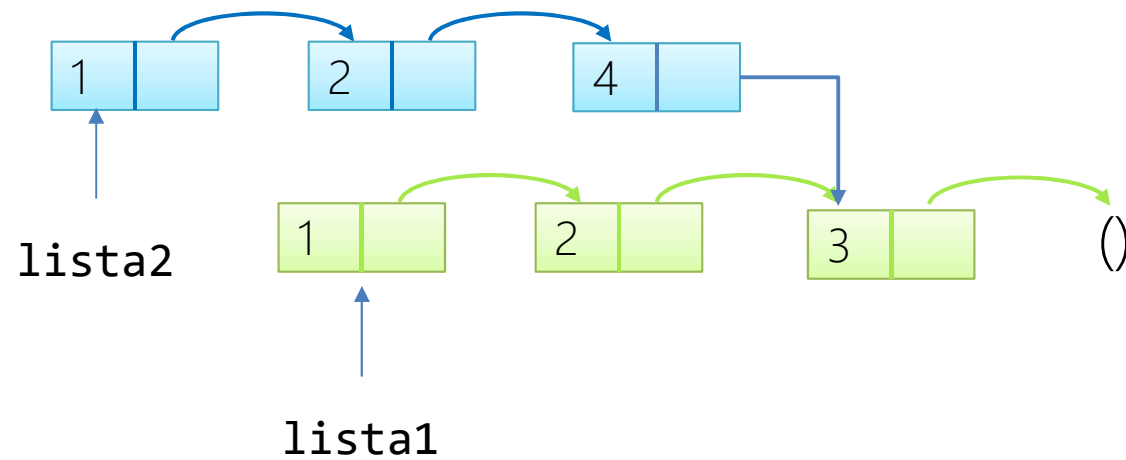
```
let lista1 = Element(1, Element(2, Element(3, Pusta)))
```

```
let lista2 = Element(4, lista1)
```

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
```

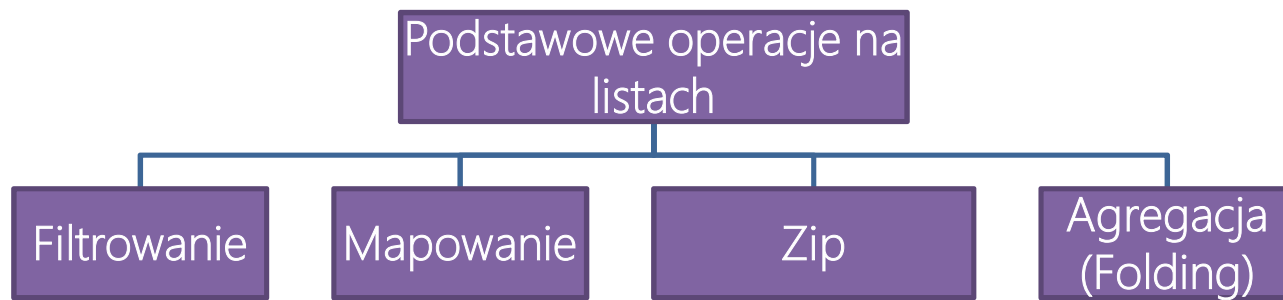
Listy – dodawanie elementu

```
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.

E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	E ₇	E ₈	E ₉
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------



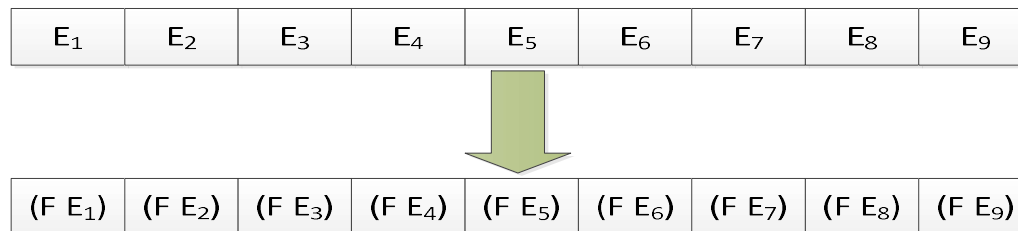
E ₁	E ₂	E ₃	E ₆	E ₇	E ₉
----------------	----------------	----------------	----------------	----------------	----------------

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

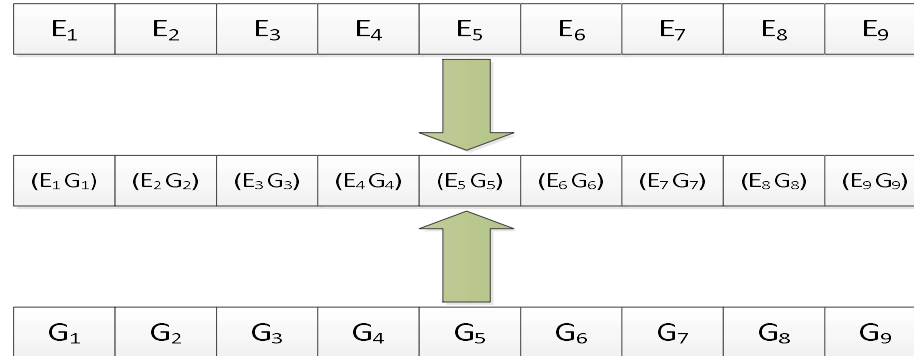


Mapowanie

```
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

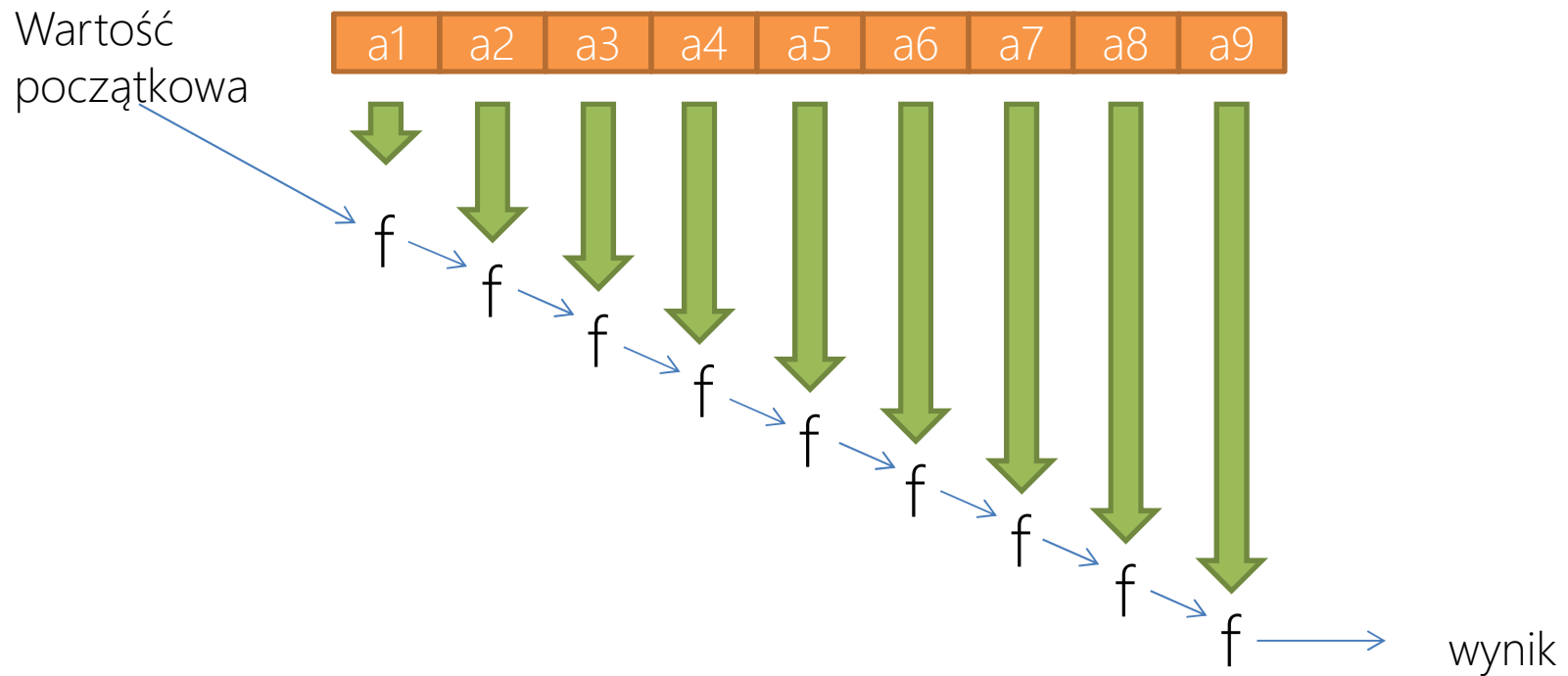


zip

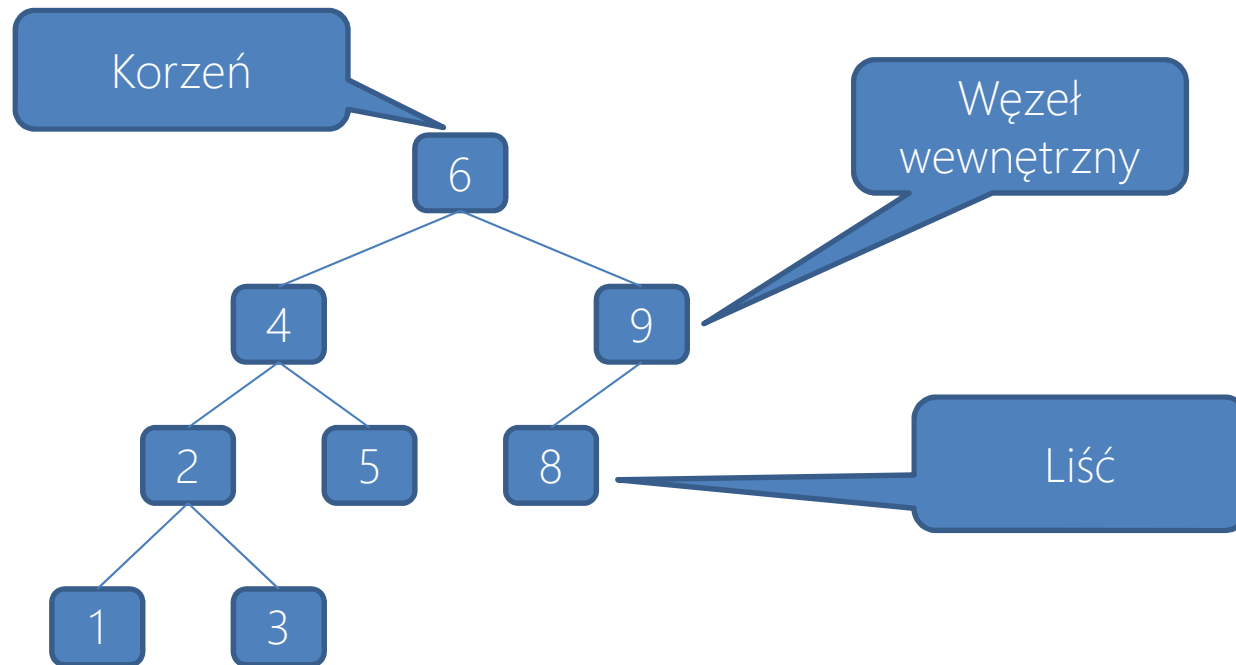
```
let rec zip lista1 lista2 =  
  match (lista1, lista2) with  
  | (Pusta, Pusta) -> Pusta  
  | (Element (glowa1,ogon1), Element (glowa2,ogon2))  
    -> Element ((glowa1,glowa2), zip ogon1 ogon2)  
  | (Pusta, _) | (_, Pusta)  
    -> failwith "Obie listy powinny mieć taką samą liczbę elementów"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

Agregacja

Jest to operacja zamieniająca listę elementów na wartość skalarną

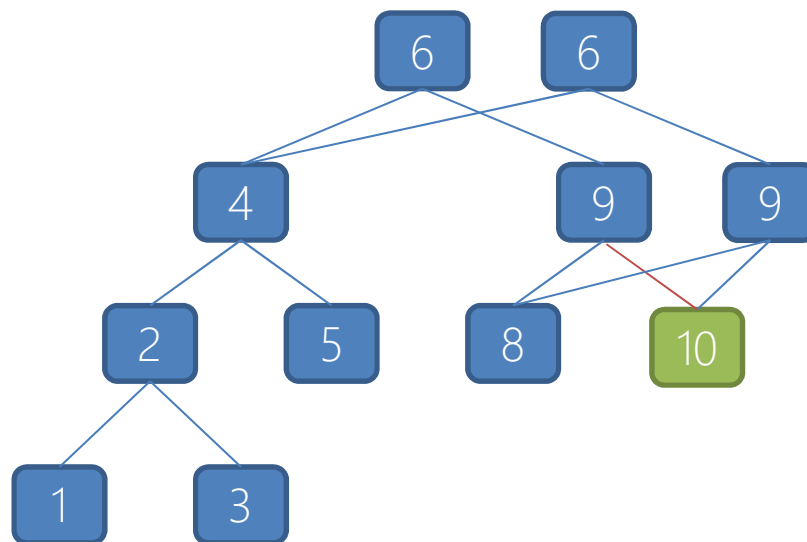


Struktury drzewiaste



$$Drzewo = \left\{ \begin{matrix} n \\ (n * Drzewo * Drzewo) \end{matrix} \right.$$

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
```