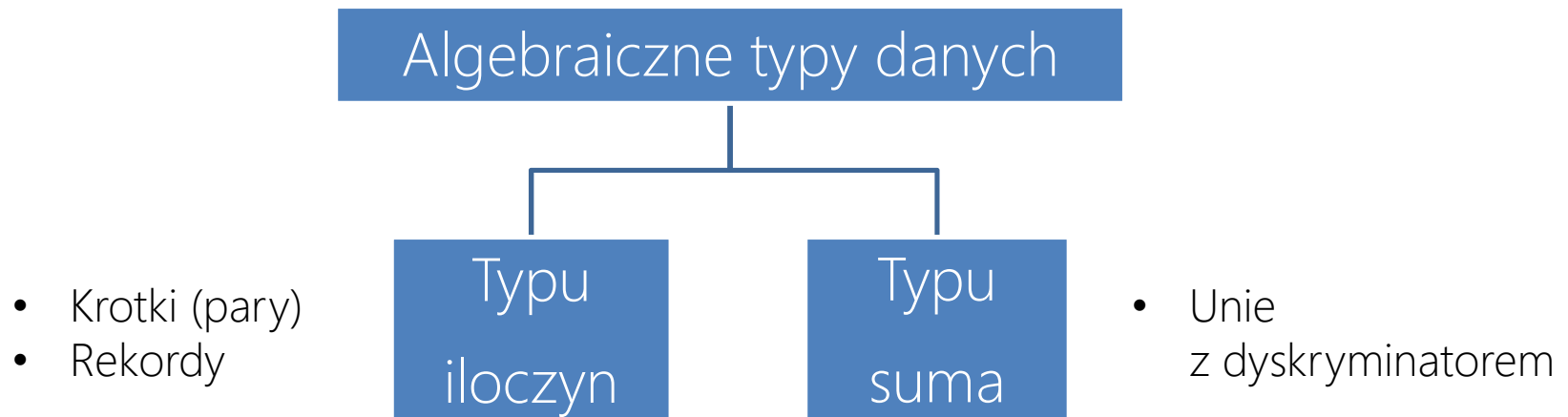


# TYPY ALGEBRAICZNE

# Typy złożone

---

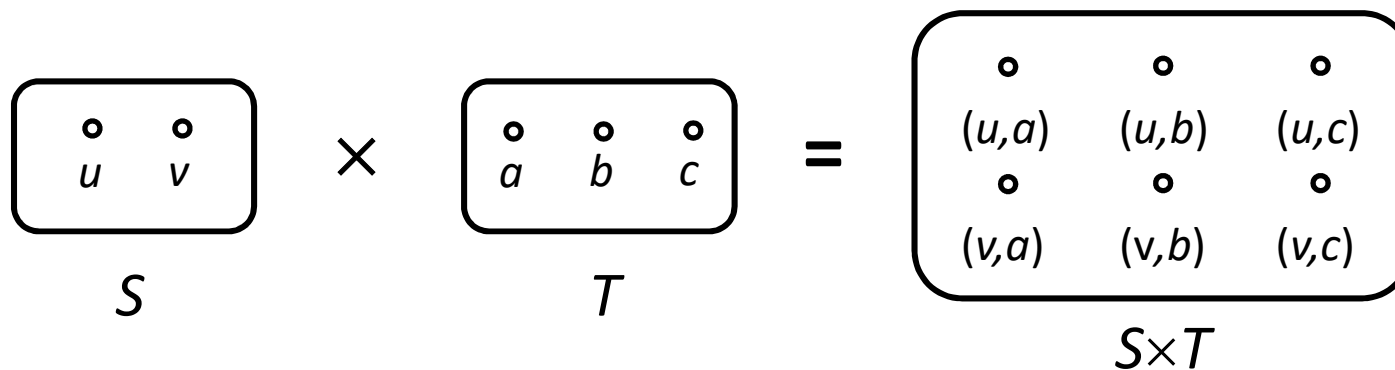


# Iloczyn kartezjański

---

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;;
```



```
val second : a:'a * b:'b * c:'c * d:'d -> 'b
```

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

## Krotki (n-tki)

---

Krotki mogą być również zwracane z funkcji:

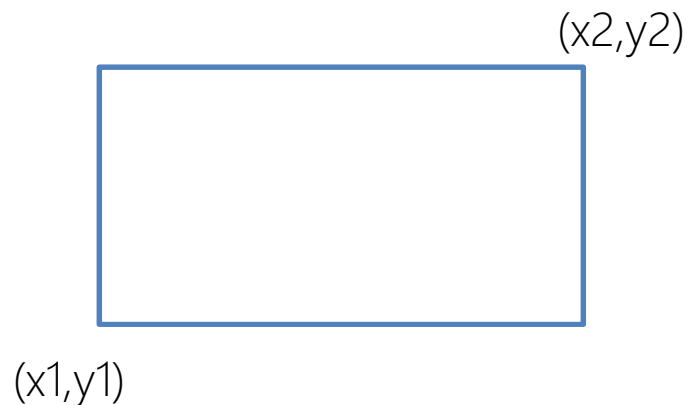
```
let funkcja a b = (a+b, a-b)
```

```
let suma,roznica = funkcja 5 4
```

# Funkcja jednego czy wielu argumentów?

---

Założmy, 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

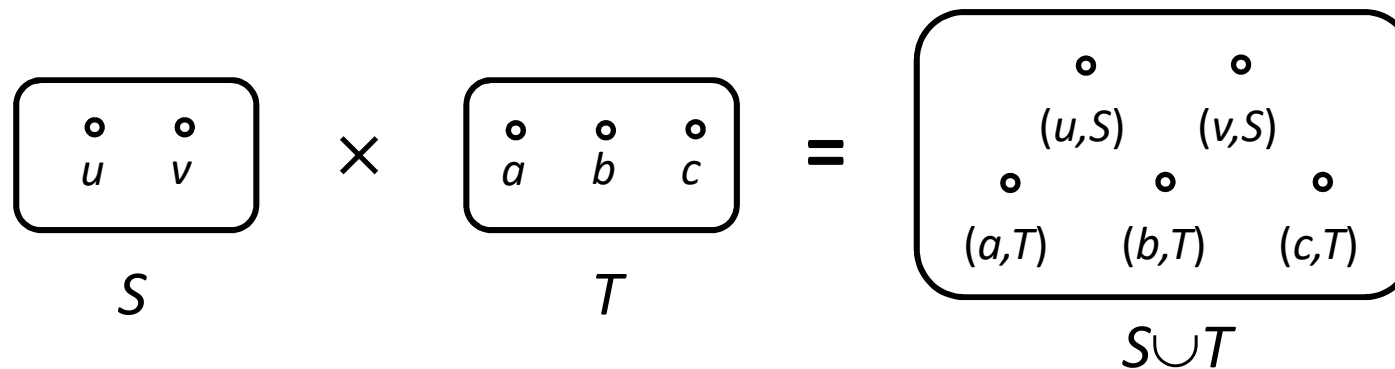
```
type Osoba = {  
    Imie : string;  
    Nazwisko : string  
}
```

```
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

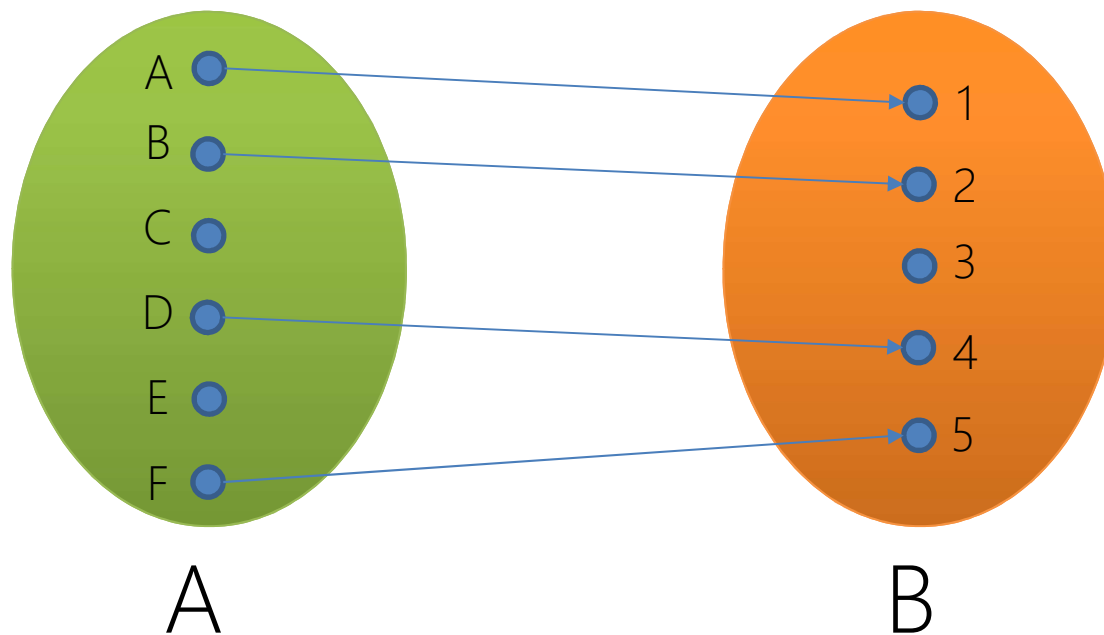
---

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

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

# Funkcje częściowe

---



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

# Funkcje częściowe

---

Założmy, 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

---

Założmy, 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

---

Założmy, 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
```

```
| Bład wiadomosc -> printfn "%s" wiadomość
```



Liczy się pierwsze dopasowanie



Lista wyrażeń musi być kompletna dla danego typu danych

# Funkcje dopasowania wzorców

---

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

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