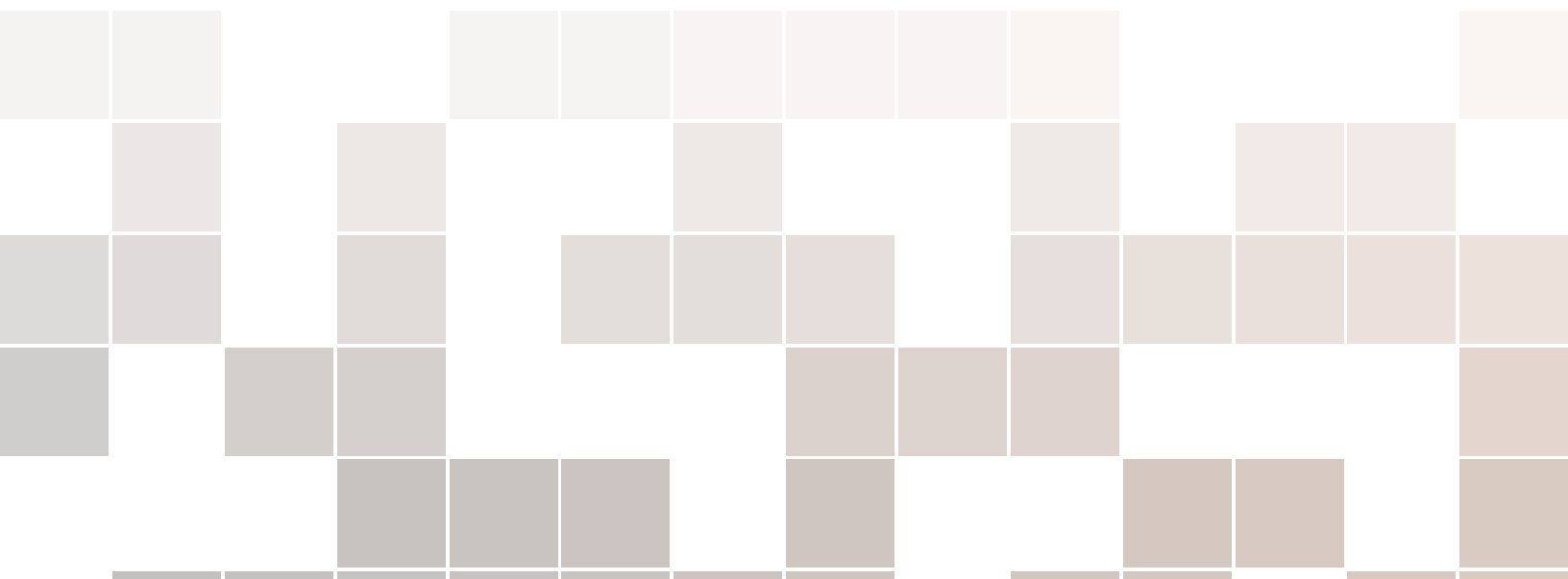


Programowanie funkcyjne

Materiały z wykładu i laboratorium

Łukasz Bartczuk



2. Algebraiczne typy danych w F#

Do tej pory zajmowaliśmy się tylko prostymi wartościami oraz funkcjami. Oczywiście są one niewystarczające do stworzenia większej aplikacji. Często potrzebujemy połączyć wiele wartości w jedną strukturę opisującą jakiś rzeczywisty obiekt. W F# możemy to zrobić wykorzystując tzw. algebraiczne typy danych. Należą do nich m.in. pary, n-tki rekordy oraz unie z dyskryminatorem¹. W tej części przedstawimy krotki i rekordy. Unie z dyskryminatorem opiszemy później ponieważ w F# łączą się one ściśle z dopasowaniem wzorców, które będziemy omawiać na kolejnych zajęciach.

2.1 Pary i N-tki

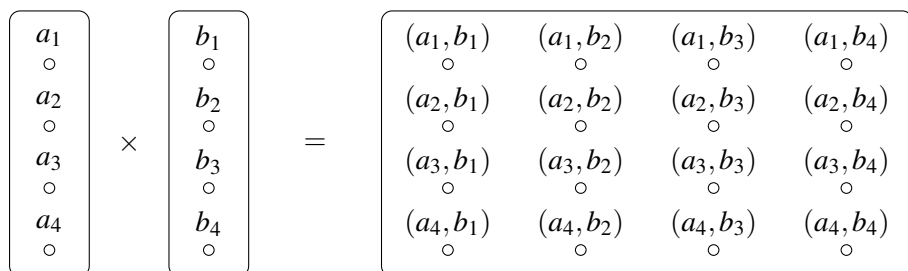
Najprostszym przypadkiem typów algebraicznych, są pary oraz ich rozszerzenie n-tki. Są one przykładem typów iloczynowych.

R Dla przypomnienia iloczynem kartezjańskim dwóch zbiorów A i B nazywamy zbiór wszystkich par uporządkowanych (a, b) , takich że $a \in A$ oraz $b \in B$

Czyli pary pozwalają one złączyć w jednej strukturze, dwie wartości różnych typów, co graficznie można przedstawić w następujący sposób:

W F# parę tworzymy podając dwie wartości (tego samego lub różnych typów), oddzielając je przecinkami:

¹Zaliczamy do nich również klasy, które także możemy tworzyć w F# , ale nimi nie będziemy się zajmować na tych zajęciach



Rysunek 2.1: Graficzna reprezentacja iloczynu kartezjańskiego

Kod 2.1: Tworzenie pary w F#

```
let para = (1, 'a');
```

W powyższym przykładzie zastosowanie nawiasów nie jest konieczne, choć często się je zapisuje dla jasności. W rezultacie na ekranie zobaczymy:

```
val para : int * char = (1, 'a')
```

Jak widzimy pary jako typ danych, są w F# określane poprzez podanie nazw tych typów oddzielonych symbolem '*'. W tym przypadku w F# mamy dwie różne notacje: jedną do zapisywania wartości (te oddzielamy przecinkiem), drugą do zapisywania typów (te oddzielamy gwiazdką). Aby pobrać wartość składowej pary w F# należy dokonać jej tzw. dekonstrukcji, czyli rozdzielić parę na poszczególne składowe (tutaj znowu nawiasy nie są wymagane, ale najczęściej stosowane):

Kod 2.2: Dekonstrukcja pary

```
let (a,b) = para;
```

co w rezultacie spowoduje przypisanie pierwszego elementu pary do zmiennej *a*, a drugiego do zmiennej *b*:

```
val b : char = 'a'
val a : int = 1
```

Jeżeli chcemy pobrać tylko jedną wartość, możemy to zrobić przygotowując specjalne funkcje. Przykładowo funkcja, która ma wyekstrahować pierwszy element może przyjąć postać:

Kod 2.3: Funkcja pobierająca pierwszy element pary

```
let pierwszy (a,b) = a
```

Powyższy zapis wyjaśnia dlaczego w F# podczas definiowania funkcji listy parametrów nie ujmujemy w nawiasach i nie oddzielamy przecinkami (w tym przypadku nawiasy

są już wymagane). Po prostu taki zapis, jak powyżej stworzy nam funkcję jednego parametru, ale parametr ten będzie w tym przypadku parą.

Ciekawy jest również rezultat powyższej instrukcji:

```
val pierwszy : a:'a * b:'b -> 'a
```

W tym przypadku, F# nie pokazuje nam bezpośrednio typu danych tylko symbole 'a oraz 'b. Są to tzw. zmienne typu, a sama funkcja jest funkcją generyczną (czyli może działać dla par o dowolnych składowych typach danych). Jest to logiczne, w końcu jeżeli chcemy tylko pobrać pierwszy element pary, to funkcja ta będzie działać w ten sam sposób, bez względu na jego typ.

- R** W funkcji przedstawionej we fragmencie 2.3 nie korzystamy z zmiennej b, choć musieliśmy ją podać, aby określić, że chodzi nam o to aby parametr ten był traktowany jako para. Takie dodatkowe, niewykorzystywane parametry (moglibyśmy określić je jako atrapy ang. *dummy parameters*), zmniejszają czytelność kodu, ponieważ jak będziemy czytać taką funkcję możemy się zastanawiać dlaczego parametr ten nie jest wewnątrz funkcji wykorzystany – czy jest to błąd czy celowe działanie. Oczywiście w powyższym fragmencie sprawa jest jasna ze względu na nazwę funkcji, ale nie zawsze jest to takie oczywiste. Dlatego też w F# do określania takich atrap wykorzystuje się symbol podkreślenia '_', który określa parametr anonimowy, na którym nam nie zależy, ale z różnych względów musimy go podać. Dlatego funkcję z fragmentu 2.3 możemy również przedstawić w następujący sposób:

Kod 2.4: Atrapy elementów pary

```
let pierwszy (a,_) = a
```

co jednoznacznie wskazuje na nasze intencje, co do drugiego parametru.

Na obraz i podobieństwo funkcji pierwszy możemy również zdefiniować funkcję, która zwróci drugi element pary:

Kod 2.5: Funkcja pobierająca drugi element pary

```
let drugi (_,b) = b
```

Ponieważ pary są bardzo często wykorzystywane w programowaniu funkcyjnym, nie ma potrzeby, abyśmy musieli samodzielnie definiować funkcje tj. pierwszy, czy drugi. Zostało to już zrobione przez twórców F#, a funkcje te nazywają się odpowiednio fst i snd. Przykładowo, aby pobrać pierwszy element pary zastosujemy następujący kod:

Kod 2.6: Przykładowa aplikacja wbudowanej funkcji fst

```
fst para
```

```
val it : int = 1
```

N-tki lub inaczej krotki, są rozszerzeniem par na dowolną liczbę elementów:

Kod 2.7: Tworzenie krotki

```
let krotka = (1,"Ala",true,1.23);;
```

```
val krotka : int * string * bool * float = (1, "Ala", true, 1.23);;
```

Jednak w tym przypadku musimy pamiętać, że funkcje do ich obsługi już nie istnieją i musimy je napisać samodzielnie.

- R** Musimy pamiętać, że podobnie jak inne wartości pary i krotki są w F# niemodyfikowalne, dlatego nie można zmienić wartości ich składowych, w inny sposób jak poprzez utworzenie nowej struktury, za pomocą przygotowanej funkcji:

Kod 2.8: Tworzenie krotki na podstawie istniejącej

```
let zNowymPierwszym (pierwszy,drugi) nowyPierwszy =  
    (nowyPierwszy,drugi)
```

Każda funkcja zawsze zwraca co najwyżej jedną wartość. Czasami zdarza się jednak, że tworzymy funkcję, która musi złamać (przynajmniej pod względem logicznym) tę zasadę. Na przykład chcemy napisać funkcję, która przyjmuje dwie liczby i chcemy aby zwróciła jednocześnie ich sumę oraz różnicę. Aby stworzyć taką funkcję należy właśnie wykorzystać krotki:

Kod 2.9: Zwracanie krotki z funkcji

```
let sumaIRoznica x y = (x+y, x-1)  
let wynik = sumaIRoznica 5 4  
printf "Suma wynosi %d, a roznica %d" (fst wynik) (snd wynik)
```

Formalnie nasza funkcja zwraca jedną wartość (właśnie krotkę) wobec tego dostęp do jej elementów składowych wymaga aplikacji funkcji `fst` i `snd` jak pokazano powyżej. Można to jednak zapisać prościej:

Kod 2.10: Dekonstrukcja rezultatu funkcji

```
let suma,roznica = sumaIRoznica 5 4  
printf "Suma wynosi %d, a roznica %d" suma roznica
```

Mechanizm ten nazywa się dekompozycją krotek i w tym przypadku lepiej oddaje intencje naszej funkcji `sumaIRoznica`, której celem jest zwrócenie dwóch wartości, które mogą być rozpatrywane osobno.

2.2 Rekordy

Ponieważ do elementów krotki dostajemy się na zasadzie pozycyjnej, interpretacja znaczenia poszczególnych jej elementów, zależy wyłącznie od programisty. Z tego powodu nie nadają się one do modelowania bardziej złożonych struktur ze świata rzeczywistego, jak np. punkt na ekranie. Jako para mógłby on być określony następująco `int*int`, ale czy pierwsza składowa takiej pary określa współrzędną x czy y ? Ponieważ jest to kwestia interpretacji i umowy pomiędzy twórcą takiego typu, a programistą, który będzie go wykorzystywał mogłoby często dochodzić do pomyłek.

Problem ten rozwiązują rekordy, które również zalicza się do typów iloczynowych, ale w przeciwieństwie do krotek, do jej składowych mamy dostęp nie za pomocą pozycji w strukturze, tylko ich nazw.

Przykładowo, poniżej jest przedstawiona struktura, która może reprezentować punkt na ekranie.

Kod 2.11: Deklaracja nowego typu rekordowego

```
type Punkt = {  
    X:int  
    Y:int  
    Kolor:string  
}
```

Jak widzimy nowy typ typu rekord definiujemy wykorzystując słowo kluczowe `type`, po którym podajemy nazwę typu i znak równości. Składowe rekordu zapisujemy w klamrach, podając ich nazwę oraz typ. Nazwę składowej od jej typu oddzielamy dwukropkiem. Poszczególne elementy rekordu separujemy znakiem nowej linii lub średnikiem.

W celu stworzenia nowej instancji danego rekordu również możemy zastosować notację nawiasów klamrowych, przy czym w tym przypadku nazwę składowej od wyrażenia wyznaczającego przypisaną jej wartość oddzielamy znakiem równości:

Kod 2.12: Utworzenie nowego rekordu

```
let p1 = {X=12; Y=25; Kolor="czerwony"}
```

Jak widzimy, nie musimy tutaj podawać nazwy typu rekordu, który chcemy utworzyć (F# jest w stanie go wywnioskować na podstawie nazw oraz typu podanych składowych). Należy pamiętać, że podanie wszystkich składowych rekordu jest obowiązkowe.

```
val p1 : Punkt = { X = 12  
                  Y = 25  
                  Kolor = "czerwony" }
```


Jeżeli będziemy chcieli uzyskać dostęp do składowej rekordu, możemy to zrobić za pomocą notacji kropkowej, przykładowo `p1.X`

Rekordy są również wartościami niemodyfikowalnymi. Co oznacza, jak w poprzednich przypadkach, że nie można zmienić wartości ich składowych. Możemy jednak na podstawie istniejącego rekordu utworzyć nowy, w którym zmodyfikujemy wartości wybranych składowych. Do tego celu w F# stworzono bardzo ciekawą i prostą składnię. Przykładowo, jeżeli chcemy utworzyć nowy punkt o takich samych współrzędnych jak podane w zdefiniowanym wcześniej punkcie `p1`, lecz innym kolorze, możemy to zrobić w następujący sposób:

Kod 2.13: Utworzenie nowego rekordu na podstawie istniejącego

```
let p2 = {p1 with Kolor="niebieski"}
```

Jeżeli za pomocą powyższej składni, będziemy chcieli zmodyfikować więcej niż jedną składową, to kolejne składowe oddzielamy od siebie średnikami.

Oczywiście rekordy możemy przekazywać do funkcji jako parametry. Na poniższym przykładzie mamy zdefiniowaną funkcję, która oblicza pole prostokąta określonego za pomocą dwóch punktów:

Kod 2.14: Obliczanie pola prostokąta określonego dwoma punktami

```
let poleProstokata p1 p2 =  
    abs(p2.X-p1.X)*abs(p2.Y-p1.Y)
```

Podobnie jak w przypadku deklarowania wartości typu rekordowego, tutaj również nie ma potrzeby jawnego określenia typu:

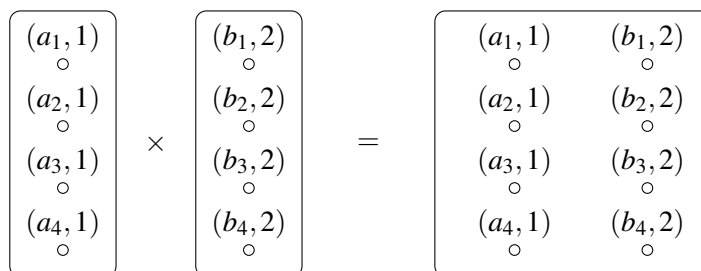
```
val poleProstokata : p1:Punkt -> p2:Punkt -> int
```

Kompilator/interpreter F# jest w stanie poprawnie go określić na podstawie składowych. Zwróćmy uwagę, że nie musimy wykorzystywać wszystkich składowych rekordu. Może się zdarzyć sytuacja, w której wewnątrz funkcji będziemy korzystać ze składowych rekordu o nazwach wykorzystywanych w więcej niż jednym rekordzie. W takim przypadku, konieczne będzie wsparcie mechanizmu wnioskowania typów poprzez jawne określenie typu parametru.

2.3 Unie z dyskryminatorem

Unie z dyskryminatorem są przykładem złożonej wartości typu suma. Przechowują one pojedynczą wartość wybraną z jednego z kilku, najczęściej różnych zbiorów. Dyskryminator, który występuje w nazwie tego typu jest to etykieta, która określa, z którego zbioru jest wybrana wartość.

Załóżmy, że chcielibyśmy napisać aplikację służącą do obliczania pól powierzchni różnych figur. Oczywiście dla każdej figury możemy stworzyć osobą funkcję obliczającą



Rysunek 2.2: Graficzna reprezentacja sumy z dyskriminatorem

wymaganą wartość, ale korzystanie z takiego zestawu funkcji mogłoby się okazać kłopotliwe. Ponadto, jeżeli chcielibyśmy posiadać w aplikacji kolekcję figur (o kolekcjach będziemy mówić następnym razem), wszystkie elementy muszą mieć jakiś wspólny mianownik (kolekcje w F# są homofobiczne tzn. wszystkie elementy muszą być tego samego lub pochodnego typu).

Dlatego do zdefiniowania typu danych opisujących pojedynczą figurę najwygodniej będzie nam wykorzystać unie z dyskriminatorem. W F# robimy to następująco:

Kod 2.15: Deklaracja unii z dyskriminatorem

```
type Figura =
    | Prostokat of szerokosc:float*wysokosc:float
    | Okrag of float
    | Trojkat of szerokosc:float*wysokosc:float
    | Prostopadloscian of
        szerokosc:float*wysokosc:float*float
```

W tym przypadku stworzyłem nowy typ danych o nazwie *Figura*, który zawiera cztery możliwe rodzaje figur. Nazwa danego rodzaju jest wspomnianą wcześniej etykietą, która pozwoli nam rozpoznać z jakim dokładnie typem mamy do czynienia. Ponadto stanowi ona konstruktor danego typu. Każdy rodzaj może (ale nie musi) przechowywać dodatkowe dane, które definiujemy po słowie kluczowym *of*. Może to być pojedyncza wartość lub zestaw wartości zdefiniowany w sposób podobny do krotek – na co wskazuje symbol *'*'*. W przeciwieństwie jednak do krotek, tutaj możemy podać nazwy poszczególnych składowych. Jeżeli do jakiejś składowej nie przypiszemy nazwy, będziemy ją określali mianem składowej anonimowej.

R W C# nie ma konstrukcji, która odpowiadałaby F# -owej unii z dyskriminatorem. W najnowszej wersji platformy .NET najbardziej podobna będzie hierarchia rekordów:

```
public abstract record Figura;

public record Prostokat(
```

```
double Szerokosc,  
double Wysokosc ) : Figura;  
  
public record Okrag(double Promien) : Figura;  
  
public record Trojkat(  
    double Szerokosc,  
    double Wysokosc) : Figura;  
  
public record Prostopadloscian(  
    double Szerokosc,  
    double Wysokosc,  
    double Glebokosc) : Figura;
```

Nie da się jednak ukryć, że już samo zdefiniowanie takiej hierarchii jest dużo bardziej czasochłonne i zdecydowanie mniej czytelne niż unia w F#. Ponadto ich wykorzystanie w programie jest dużo prostsze i przyjemniejsze niż taka hierarchia w C#.

W celu skonstruowania wartości określonych typów musimy podać ich nazwę oraz w nawiasie wymagane składowe np.

Kod 2.16: Utworzenie wartości typu Figura

```
let kwadrat = Prostokat(12.0, 12.0)  
let prostokat = Prostokat(szerokosc=12.0, wysokosc=24.0)  
let okrag = Okrag(12.0)  
let trojkat = Trojkat(wysokosc=12.0, szerokosc=24.0)  
let prostopadloscian = Prostopadloscian(12.0, 24.0, 12.0)
```

Parametry możemy podawać na dwa sposoby – pozycyjnie lub poprzez nazwę. W przypadku podawania parametrów pozycyjnie musimy je wprowadzić w takiej kolejności jak zostały zdefiniowane. Jeżeli chcemy je podać poprzez nazwę, kolejność podawania parametrów może być dowolna – jak nam pasuje. Zwróćmy jednak uwagę na przypadek `Prostopadloscian`. Zostały w nim przemieszane zarówno składowe nazwane, jak i anonimowe. W takim przypadku wartości parametrów anonimowych muszą być podane przed nazwanymi. Są one jednak wprowadzane w sposób pozycyjny, czyli pierwszy parametr pod pierwszą składową:

```
let prostopadloscian = Prostopadloscian(12.0, 24.0, 12.0)
```

Z tego powodu poniższa instrukcja:

```
let prostopadloscian = Prostopadloscian(12.0, szerokosc=24.0, ↵  
    wysokosc=12.0)
```

będzie błędna. F# będzie się starał dwukrotnie ustawić składową `szerokosc` (raz ponieważ jest to pierwszy parametr, drugi raz ponieważ jest on wymieniony z nazwy) co jest

niedopuszczalne. Dlatego w przypadku stosowania składowych zarówno nazwanych, jak i anonimowych, wszystkie składowe występujące przed składową anonimową należy traktować również jako anonimowe.

2.4 Dopasowywanie wzorców

W językach imperatywnych bardzo często stosowaną instrukcją jest instrukcja wyboru. Przykładowo w języku C++ przybiera ona postać komendy `switch` o składni:

```
switch(x) {  
    case wartosc_1: <instrukcje_1> break;  
    case wartosc_2:  
    case wartosc_3: <instrukcje_2> break;  
    default: <instrukcje_1> break;  
}
```

Podobna instrukcja istnieje również w języku F#. Jest to wyrażenie `match`, którego składnia jest następująca:

```
match <wyrażenie> with  
| wzorzec_1 -> wyrażenie_1  
| wzorzec_1 -> wyrażenie_2  
| ...
```

Składania tego wyrażenia jest zbliżona do składni instrukcji `switch`, przy czym zamiast słowa kluczowego `case` tutaj mamy pionową kreskę. W najprostszym przypadku wzorzec jest wartością stałą, która jest porównywana z wartością wyznaczoną przez wyrażenie. Z kolei wyrażenie jest instrukcją, która ma być wykonana dla danego wzorca. Musimy pamiętać, że instrukcja `match with` jest wyrażeniem zwracającym wartość. Wobec tego typ wszystkich wyrażeń składowych musi być taki sam:

Kod 2.17: Instrukcja wyboru w F#

```
let x = 1;  
match x with  
| 1 -> "jeden"  
| 2 | 3 -> "dwa lub trzy"  
| _ -> "inna wartosc"
```

W powyższym kodzie zamieniamy liczbę całkowitą na odpowiedni łańcuch znaków. Podobnie jak w pokazanej wcześniej instrukcji `switch` wykonywana jest seria porównań wartości symbolu `x` z podanymi wartościami, aż odnalezione zostanie dopasowanie. Wtedy obliczona jest wartość określonego wyrażenia i ona jest rezultatem instrukcji `match`.

Musimy pamiętać, że lista wzorców musi być kompletna dla danego typu danych. Jeżeli interesują nas tylko wybrane wartości (jak w powyższym przykładzie 1,2 lub

3), to ostatni wzorzec powinien przechwytywać wszystkie pozostałe wartości (jest to odpowiednik pozycji default z instrukcji switch).

Wzorcem stosowanym w wyrażeniu `match` nie muszą być wyłącznie wartości typu prostego, jak pokazano powyżej. W tabeli przedstawione są podstawowe rodzaje wzorców jakie mogą być wykorzystane podczas tworzenia aplikacji. Powodują one, że wyrażenie `match` jest dużo potężniejszym narzędziem niż instrukcja `switch`.

Tabela 2.1: Przykładowe rodzaje wzorów wyrażenia `match`

Nazwa	schemat wzorca
wartości	dowolna liczba, znak, string lub zdefiniowany literal
zmiennej	identyfikator
identyfikatora	Przypadek unii z dyskryminatorem
rekordu	{ identyfikator1=wzorzec1; ...; identyfikatorN=wzorzecN }
krotki	(wzorzec1, ... ,wzorzecN)
lub	wzorzec1 wzorzec2
i	wzorzec1 & wzorzec2

Poniżej schematy te zostaną wyjaśnione wraz z prostymi przykładami. Zwróćmy jednak uwagę, że jeżeli w jakimś schemacie pojawia się "wzorzec" to można je zagnieźdzać.

2.4.1 Wzorzec zmiennej

Wzorzec zmiennej pozwala na przypisanie dopasowanej wartości do symbolu, z którego możemy później skorzystać w wyrażeniu np.

Kod 2.18: Wykorzystanie wzorca zmiennej w wyrażeniu dopasowania

```
let x = 1;
match x with
| 1 -> "jeden"
| 2 | 3 -> "dwa lub trzy"
| x -> sprintf "inna wartosc: %d" x
```

2.4.2 Wzorzec krotki

Wzorzec krotki pozwala na dopasowanie wartości do zdefiniowanej krotki

Kod 2.19: Wykorzystanie wzorca krotki w wyrażeniu dopasowania

```
match para with
| (false, _) -> false
| (true, x) -> x
```

W powyższym przykładzie dopasowywana jest para wartości logicznych. Wyrażenie dopasowania wzorca określa, że jeżeli pierwszy element pary jest `false`, to bez względu na wartość drugiego elementu zwrócona zostanie wartość `false`. Jeżeli jednak pierwszy element pary jest `true`, to wartością całego wyrażenia będzie drugi element pary.

2.4.3 Wzorzec rekordu

Ten rodzaj wzorca pozwala nam na dekompozycję rekordu i dopasowanie się do wartości jego pól.

Kod 2.20: Wykorzystanie wzorca rekordu w wyrażeniu dopasowania

```
type Pojazd = {  
    marka:string  
    model:string  
}  
  
let rekord = {marka="Ford"; model="Focus"; rokProdukcji=2021}  
match rekord with  
| {marka="Ford"} -> "To jest Ford"  
| {marka="Opel"; model=model } ->  
    sprintf "To jest Opel %s" model  
| _ -> "Inne auto"
```

W tym przykładzie zdefiniowałem nowy typ `Pojazd` o składowych `marka` i `model`, a następnie wykorzystałem dopasowanie wzorca do określenia zachowania programu dla różnych marek aut. Jak widzimy możemy zdefiniować dokładnie o jaką wartość składowej nam chodzi lub przypisać wartość składowej do zmiennej. W przypadku wzorca rekordu istotne jest to, że może niekompletny (tzn. nie musimy definiować wzorców dla wszystkich składowych).

2.4.4 Wzorzec identyfikatora

Pozwala na dopasowanie wartości do poszczególnych przypadków unii z dyskriminatorem. Przykładowo, jeżeli będziemy chcieli wypisać w konsoli komunikaty dla poszczególnych rodzajów figur z przykładu przedstawionego w punkcie 2.3, możemy to zrobić za pomocą następujących instrukcji:

Kod 2.21: Wykorzystanie wzorca identyfikatora w wyrażeniu dopasowania

```
match prostokat with  
| Trojkat (wysokosc=h; szerokosc=w)  
    -> printfn "Figura jest trojkatem %f %f" h w  
| Prostokat (wysokosc, szerokosc)  
    -> printfn "Figura jest prostokatem %f %f" wysokosc szerokosc
```

```
| Okrag (promien)
    -> printfn "Figura jest okregiem o promieniu %f" promien
| Prostopadloscian (wysokosc,szerokosc,glebokosc)
    -> printfn "Figura jest prostopadloscianem %f %f %f" ←
        wysokosc szerokosc glebokosc
```

Dane składowe przypadku podajemy w nawiasach, przypisując każdej z nich nazwę, za pomocą której możemy się do niej odwoływać w wyrażeniu przypisanym do danego wzorca. Jeżeli podczas definiowania typu wykorzystaliśmy składowe nazwane, możemy je wykorzystać jak pokazałem to w przypadku `Trojkat`. Nazwy za pomocą, których w wyrażeniu będziemy się odnosić do składowych tego przypadku przypisujemy po znaku równości, oddzielając kolejne składowe średnikiem. Kolejność składowych nie ma znaczenia. W przypadku gdy, korzystaliśmy ze składowych anonimowych, ich nazwy będą przypisywane w kolejności zdefiniowania.

Liczba podanych wzorców musi odpowiadać liczbie przypadków zdefiniowanych w unii z dyskryminatorem (musi to być lista kompletna). Jeżeli nie chcemy rozpatrywać jakichś przypadków, musimy w wyrażeniu dopasowania wzorca zdefiniować wzorzec domyślny:

Kod 2.22: Wzorzec identyfikatora w wyrażeniu dopasowania – przypadek domyślny

```
match prostokat with
| Trojkat (wysokosc=h; szerokosc=_)
    -> printfn "Figura jest trojkatem o wysokosci %f" h
| Prostokat (wysokosc, _)
    -> printfn "Figura jest prostokatem o wysokosci %f" wysokosc
| _ -> printfn "Inna figura"
```

Symbol podkreślenia możemy również wykorzystać, jeżeli w wyrażeniu nie jesteśmy zainteresowani jakąś składową przypisaną do danego przypadku (musimy pamiętać, że lista składowych musi być kompletna – tak jak została zdefiniowana – jeżeli jakaś składowa nas nie interesuje, nie możemy jej tak po prostu pominąć).

2.4.5 Wzorce warunkowe

Do każdej pozycji wyrażenia `match` możemy dołączyć klauzulę `when`, które określają dodatkowe warunki jakie muszą spełniać zmienne występujące w wzorcu aby można było wzorzec uznać za dopasowany. Wyrażenie takie określane jest mianem strażnika. Przykładowo:

Kod 2.23: Wykorzystanie wzorca warunkowego w wyrażeniu dopasowania

```
type Pojazd = {
    marka:string
```

```
    model:string
    rokProdukcji:int
}

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

W tym przypadku trzecia pozycja wyrażenia `match` jest dodatkowo zabezpieczona warunkiem określającym rok produkcji samochodu. Musimy pamiętać, że po słowie kluczowym `when` może występować dowolne wyrażenie logiczne, przy czym jego wartość będzie wyznaczona tylko jeżeli wcześniej nastąpiło dopasowanie do wzorca skojarzonego z rozpatrywanym strażnikiem.

2.4.6 Funkcje dopasowania wzorca

Wyrażenie `match with` możemy umieścić wewnątrz funkcji, przykładowo:

Kod 2.24: Funkcja zawierająca wyrażenie `match`

```
let jakoString figura =
    match figura with
    | Trojkat (wysokosc=h; szerokosc=w)
        -> sprintf "Figura jest trojkatem %f %f" h w
    | Prostokat (wysokosc, szerokosc)
        -> sprintf "Figura jest prostokatem %f %f" wysokosc <-
            szerokosc
    | Okrag (promien)
        -> sprintf "Figura jest okregiem o promieniu %f" promien
    | Prostopadloscian (wysokosc,szerokosc,glebokosc)
        -> sprintf "Figura jest prostopadloscianem %f %f %f" <-
            wysokosc szerokosc glebokosc
```

Zwróćmy uwagę, że jedyne zadanie powyższej funkcji to dopasowanie wzorca oraz wyznaczenie dla każdego przypadku jakiejś wartości. Funkcje takie są w F# tworzone bardzo często. Z tego powodu twórcy języka opracowali uproszczoną składnię, która umożliwia ich definiowanie. Powyższą funkcję można zapisać również w formie:

Kod 2.25: Funkcja dopasowania wzorca


```

let jakoString = function
| Trojkat (wysokosc=h; szerokosc=w)
    -> sprintf "Figura jest trojkatem %f %f" h w
| Prostokat (wysokosc, szerokosc)
    -> sprintf "Figura jest prostokatem %f %f" wysokosc <-
        szerokosc
| Okrag (promien)
    -> sprintf "Figura jest okregiem o promieniu %f" promien
| Prostopadloscian (wysokosc,szerokosc,glebokosc)
    -> sprintf "Figura jest prostopadloscianem %f %f %f" <-
        wysokosc szerokosc glebokosc

```

Zwróćmy uwagę, że w powyższej funkcji zniknął nam argument (kompilator/interpreter zakłada, że dopasowanie wzorca odbywa się po ostatnim parametrze) oraz wyrażenie `match with`. Zamiast niego pojawiło się słowo kluczowe `function`.

Mając już wszystkie posiadane informacje, napisanie funkcji obliczającej pole powierzchni figury, nie powinno być dla was problemem:

Kod 2.26: Obliczanie pola powierzchni figury, jako funkcja dopasowania wzorca

```

let polePowierzchni = function
| Prostokat (s,w) -> s*w
| Trojkat (p,w) -> 0.5*p*w
| Okrag (r) -> Math.PI*r**2.0
| Prostopadloscian (s,w,g) -> 2.0*(s*w+w*g+s*g)

```

2.5 Równość wartości typów algebraicznych

Jeżeli porównujemy wartości typów prostych np. wartości typu `int`, to intuicyjnie chcemy porównywać ich wartości. Czyli jeżeli zdefiniujemy dwie zmienne zawierające tę samą wartość typu `int` i porównamy je ze sobą, to będziemy chcieli uzyskać wartość `true` o ile tylko ich wartości będą zgodne. We wszystkich językach programowania odbywa się to w ten sam sposób.

Sprawa trochę się komplikuje w przypadku wartości złożonych takich jak krotki czy rekordy. Dla tych typów porównywanie może odbywać się na dwa sposoby:

1. **porównywanie wartości** - dwie wartości złożone są sobie równe jeżeli ich składowe mają te same dane.
2. **porównywanie referencji** - dwie wartości złożone są sobie równe, jeżeli są umieszczone w tym samym miejscu w pamięci.

Domyślnie w F# typy algebraiczne są traktowane jako typy wartościowe², czyli ich

²odbywa się to bez względu na rodzaj struktury, która jest dla takiego typu danych tworzona. Przykładowo rekord jest przez kompilator definiowany jako klasa zapieczętowana C#, a te domyślnie są typami

równość jest określana na podstawie porównywania wartości. Przykładowo, jeżeli zdefiniujemy dwie zmienne typu Punkt i porównamy je ze sobą:

Kod 2.27: Równość dwóch rekordów

```
let p1 = {X=12; Y=24; Kolor="czerwony"}
let p2 = {X=12; Y=24; Kolor="czerwony"}
p1=p2 //true
```

to w rezultacie uzyskamy wartość `true`, pomimo że są one umieszczone w różnych miejscach w pamięci. Gdybyśmy chcieli porównać referencje tych dwóch zmiennych możemy to zrobić wykorzystując funkcję `PhysicalEquality` modułu `LanguagePrimitives`:

Kod 2.28: Porównywanie referencji w F#

```
LanguagePrimitives.PhysicalEquality p1 p2
```

W tym przypadku dla zmiennych `p1` i `p2` uzyskamy oczywiście wartość `false`.

2.6 Definiowanie metod w rekordach oraz uniach z dyskryminatorem

Zarówno do rekordów, jak również do unii z dyskryminatorem możemy dodawać nowe lub nadpisywać istniejące w nich metody. Chodź podejście takie kojarzy się bardziej z programowaniem obiektowym, czasem może być bardzo przydatne. Załóżmy, że chcemy napisać aplikację, która będzie wykorzystywała liczby zespolone. Liczę taką możemy zapisać w formie następującego rekordu:

Kod 2.29: Zdefiniowanie nowego typu `ComplexNumber`

```
type ComplexNumber = {
    real:double;
    im:double;
}
```

Stworzenie wartości tego typu nie stanowi żadnego problemu:

Kod 2.30: Stworzenie wartości typu `ComplexNumber`

```
let c1 = { real = 5; im = 12; }
```

Jeżeli jednak będziemy chcieli wyświetlić wartość tego typu nie będzie to najlepiej wyglądać. W interpreterze zostanie ona przedstawiona w sposób standardowy dla rekordów:

referencyjnymi, czyli domyślnie porównywane jest miejsce ich umieszczenia w pamięci

```
{ real = 1.0  
  im = 5.0 }
```

Jednak w tym przypadku wartość tę lepiej przedstawić za pomocą standardowego, matematycznego zapisu liczby zespolonej: $1.0 + 5.0i$. W tym celu możemy nadpisać metodę `ToString`. Można to zrobić w następujący sposób:


Kod 2.31: Nadpisanie metody `ToString` w typie `ComplexNumber`

```
type ComplexNumber = {  
    real : double;  
    im : double;  
} with  
    override this.ToString() =  
        sprintf "%f+%fi" this.real this.im
```

Metoda ta wykorzystana będzie na przykład z funkcją `printfn`. Musimy jednak pamiętać, że jako określenie formatu należy zastosować `"%0"`:

Kod 2.32: Wyświetlenie wartości typu `ComplexNumber` za pomocą `printfn`

```
printfn "%0" c1
```

-  Nawet jeżeli nadpiszemy metodę `ToString` domyślnie interpreter i tak z niej nie skorzysta. Możemy go jednak do tego zmusić dodając tzw. drukarkę. Robimy to wydając w interpreterze polecenie:

Kod 2.33: Dodawanie drukarki interpretera dla typu `ComplexNumber`

```
fsi.AddPrinter(fun (cn:ComplexNumber) -> cn.ToString());;
```

Innym ciekawym przypadkiem, kiedy wskazane jest dodawanie metod do rekordów lub unii jest zdefiniowanie własnych operatorów (lub nadpisanie istniejących). Załóżmy, że będziemy na naszych listach zespolonych chcieli wykonywać podstawowe operacje arytmetyczne. W tym celu możemy zdefiniować następujące funkcje:

Kod 2.34: Funkcje definiujące podstawowe operacje arytmetyczne

```
let add c1 c2 = { real = c1.real + c2.real;  
                 im = c1.im + c2.im }  
  
let sub c1 c2 = { real = c1.real - c2.real;  
                 im = c1.im - c2.im }
```

```

let mul c1 c2 = {real = c1.real * c2.real - c1.im * c2.im;
                 im = c1.real * c2.im + c1.im * c2.real }

let div c1 c2 =
  let denom = c2.real*c2.real + c2.im*c2.im
  { real = (c1.real * c2.real - c1.im * c2.im) / denom;
    im = (c1.real * c2.im + c1.im * c2.real) / denom }

let neg c1 = { real = -c1.real; im = -c1.im }

```

Możemy teraz z tych funkcji korzystać, jednak zapis nie jest zbyt wygodny:

Kod 2.35: Zastosowanie funkcji add

```

let c1 = { real = 1; im = 5 }
let c2 = { real = 2; im = 7 }
let c3 = add c1 c2

```

Zdecydowanie wygodniej byłoby zapisać po prostu `let c3 = c1 + c2`. Jednak w tym celu musimy zdefiniować dla typu `ComplexNumber` podstawowe operatory binarne: `+`, `-`, `*`, `/` oraz operator unarny `-`.

```

type ComplexNumber = {
  real:double;
  im:double;
} with
  override this.ToString()
    = sprintf "%f+%fi" this.real this.im

  static member (+) (c1, c2) = {
    real = c1.real + c2.real;
    im = c1.im + c2.im }

  static member (-) (c1, c2) = {
    real = c1.real - c2.real;
    im = c1.im - c2.im }

  static member (*) (c1, c2) = {
    real = c1.real * c2.real - c1.im * c2.im;
    im = c1.real * c2.im + c1.im * c2.real }

  static member (/) (c1, c2) =

```

```
let denom = c2.real*c2.real + c2.im*c2.im
{ real = (c1.real * c2.real - c1.im * c2.im) / denom;
  im = (c1.real * c2.im + c1.im * c2.real) / denom }
```

2.7 Przydatne funkcje

1. `Console.ReadKey ()` - metoda zwraca informacje o klawiszu wciśniętym przez użytkownika. Przedstawione są one w formie obiektu typu `ConsoleKeyInfo`. Zawiera on m.in. składową `Key`, która jest enumeracją opisującą klawisze np. `ConsoleKey.D1` oznacza klawisz z liczbą 1.
2. `Console.Clear ()` - metoda czyszcząca konsolę

2.8 Zadania

Zadanie 2.1 Dany jest program w C++:

```
#include<iostream>
using namespace std;

int main() {
    int wartosc;
    cout<<"Podaj_wartosc:_";
    cin>>wartosc;

    switch(wartosc) {
        case 1: cout<<"Wprowadziles_1"<<endl; break;
        case 2: cout<<"Wprowadziles_2"<<endl; break;
        case 3: cout<<"Wprowadziles_3"<<endl; break;
        case 4: cout<<"Wprowadziles_4"<<endl; break;
        default:
            cout<<"Wprowadziles_inna_wartosc_niz_1,2,3_lub_4"<<endl;
            break;
    }
}
```

Napisz równoważny mu program w F#

Zadanie 2.2 Napisz funkcję, która będzie wczytywała od użytkownika dwie liczby, i zwracała je w formie pary. Następnie wykorzystaj dopasowanie wzorców do krotki i wyświetl na ekranie informacje czy pierwsza liczba z pary jest większa niż druga, czy druga jest większa czy obie są równe.

Zadanie 2.3 Napisz funkcję, która jako parametry będzie przyjmowała długości trzech boków trójkąta oraz zwracała jego pole i obwód. Rezultat zamodeluj za pomocą pary. Wyświetl wynik na ekranie z komunikatem w stylu "pole trójkąta to: ..., a obwód to: ...".

Zadanie 2.4 Napisz funkcję, która będzie przyjmowała jako parametr łańcuch znaków określających adres email i wydzielala z niego identyfikator użytkownika oraz adres domenowy serwera

Zadanie 2.5 Wykorzystaj funkcję napisaną w poprzednim zadaniu do podziału adresu na części, a następnie wyświetl informacje czy podany adres należy do domeny PCz, czy nie. Komunikat powinien mieć formę "Email użytkownika (nazwa użytkownika) należy do domeny PCz", lub że nie należy. Wykorzystaj dopasowanie wzorca.

Zadanie 2.6 Napisz funkcję obliczającą odległość Euklidesową pomiędzy dwoma punktami w przestrzeni 3D. Punkty zamodeluj za pomocą krotki.

Zadanie 2.7 Napisz funkcję, która sprawdzi czy podany punkt znajduje się wewnątrz okręgu. Jako parametry powinna przyjmować środek okręgu, jego promień oraz punkt, który chcemy sprawdzić.

Zadanie 2.8 Napisz aplikację, w której zdefiniujesz nowy typ danych opisujący ułamki zwykłe. Napisz funkcje umożliwiające wykonanie podstawowych operacji arytmetycznych: dodawanie, odejmowanie, mnożenie i dzielenie. Wykorzystaj krotki.

Zadanie 2.9 Napisz tę samą aplikację co powyżej, ale z wykorzystaniem rekordów.

Zadanie 2.10 Napisz program, w którym zadeklarujesz nowy typ opisujący datę (dzień, miesiąc, rok), a następnie wykorzystasz go do określenia, jaki jest dzień tygodnia w którym ta data wypada np. jeżeli użytkownik poda 1.03.2021 program powinien odpowiedzieć poniedziałek. Jako punkt wyjścia możemy przyjąć, że 1.01.1990 wypadł również w poniedziałek.

Zadanie 2.11 Napisz funkcję realizującą bezpiecznie dzielenie. Funkcja powinna przyjmować dwie liczby, które chcemy podzielić. Jeżeli jest to możliwe to funkcja powinna zwracać wynik dzielenia, a jeżeli nie komunikat "Nie można dzielić przez 0". Jaki będzie najlepszy typ do reprezentowania rezultatu takiej funkcji.

Zadanie 2.12 Zdefiniuj odpowiedni typ danych do przechowywania informacji zakodowanych w numerze VIN (Vehicle Identification Number), a następnie napisz program pozwalający dekodować ten numer. Informacje o VIN możesz zaczerpnąć np. z Wikipedii https://pl.wikipedia.org/wiki/Vehicle_Identification_Number

Zadanie 2.13 Napisz funkcję, która jako parametry będzie przyjmowała długości trzech boków trójkąta oraz zwracała jego pole i obwód. Rezultat zamodeluj za pomocą pary. Wyświetl wynik na ekranie z komunikatem w stylu "pole trójkąta to: ..., a obwód to: ...". Jeżeli z podanych boków nie da się stworzyć trójkąta, to zwróć komunikat "Z podanych wartości nie da się stworzyć trójkąta"

Zadanie 2.14 Napisz program, który będzie wczytywał od użytkownika współczynniki a , b , c równania kwadratowego $ax^2 + bx + c = 0$ rozpatrz wszystkie możliwe przypadki. Zamodeluj je za pomocą unii z dyskryminatorem. Po obliczeniach wyświetl odpowiednie komunikaty.

Zadanie 2.15 Napisz program, który będzie pozwalał przechowywać następujące informacje o osobie: imię, nazwisko, wiek. Przygotuj odpowiedni typ danych. Napisz program, który będzie pozwalał na wprowadzenie informacji o osobie, modyfikowanie ich oraz wyświetlenie. Po uruchomieniu programu użytkownik powinien zobaczyć menu:

- 1 - utworzenie rekordu
- 2 - modyfikacja rekordu
- 3 - pokaz rekord
- 4 - zakończenie programu

Po wybraniu odpowiedniej opcji użytkownik powinien mieć możliwość wprowadzenia danych (program powinien wyświetlać informacje o jakie dane prosi użytkownika), lub ich zobaczenia. Podczas modyfikowania danych jeżeli użytkownik wprowadzi łańcuch pusty (w przypadku imienia lub nazwiska) lub 0 (w przypadku wieku) program powinien zachowywać wcześniejsze dane. Po zakończeniu wprowadzania lub przeglądania danych oraz w przypadku, gdy użytkownik wybierze inną opcję niż od 1 do 4 program powinien wracać do menu głównego. Wykorzystaj dopasowanie wzorców.

Zadanie 2.16 Zmodyfikuj poprzednie zadanie, tak aby wyświetlając dane wprowadzonej osoby pojawiała się również informacja czy jest ona pełnoletnia, czy nie. Wykorzystaj dopasowanie wzorca do rekordu.