Programowanie funkcyjne

Materiały z wykładu i laboratorium

Łukasz Bartczuk

4. Struktury danych wbudowane w F#

4.1 Funkcje wyższych rzędów

W językach funkcyjnych takich jak F# funkcje są podstawową jednostką, z których możemy budować nasze aplikacje. Ponadto w językach tych funkcje mogą być traktowane jak dane tzn. mogą być przekazywane do funkcji jako wartości argumentów lub z funkcji zwracane, jako rezultat ich działania. Mówimy, że funkcje są wartościami pierwszej klasy - podobnie jak wartości numeryczne, łańcuchy znaków czy inne. Funkcje które przyjmują inne funkcje jako argumenty lub je zwracają określa się mianem **funkcji wyższych rzędów**. Funkcje takie występują również w matematyce. Operacje takie jak pochodne czy całki nieokreślone przyjmują funkcje i funkcje zwracają.

Aby zrozumieć przydatność przekazywania funkcji jako argument do innej przeanalizujmy następujący program:

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

let rec suma wynik =
  function
  | Pusta -> wynik
  | Element (x,ogon) -> suma (wynik+x) ogon

let rec iloczyn wynik =
  function
```

```
  | Pusta -> wynik
  | Element (x,ogon) -> iloczyn (wynik*x) ogon

let lista = Element(1,Element(2,Element(3,Element(4,Pusta))))
suma 0 lista //10
iloczyn 1 lista //24
```

Zawiera on definicję listy oraz dwie funkcje pozwalające wyznaczyć sumę oraz iloczyn jej elementów. Każdy z was bez problemu rozpozna tu operację agregacji.

Zwróćmy uwagę, że funkcje `suma` i `iloczyn` różnią się tylko i wyłączenie fragmentem oznaczonym kolorem zielonym. Jeżeli ten fragment kodu udałyby się sparametryzować, a przez to zmieniać w zależności od potrzeb, to będziemy mogli stworzyć funkcję ogólną, która pozwala zrealizować algorytm agregacji wartości na liście. Dzięki możliwości przekazywania funkcji jako argumentów działanie to jest bardzo proste:

```
let agregacja agregat wartoscPoczatkowa lista =
  let rec agregacja wynik =
    function
    | Pusta -> wynik
    | Element (x,ogon) -> agregacja (agregat wynik x) ogon
  agregacja wartoscPoczatkowa lista
```

Sygnatura tej funkcji jest następująca:

```
agregacja :
  agregat:('a ->'b -> 'a) -> wartoscPoczatkowa:'a ->
    lista:Lista<'b> -> 'a
```

Jak widzimy `agregacja` jest funkcją trzech parametrów, gdzie pierwszy parametr `agregat` jest również funkcją. Zapisaliśmy w ten sposób ogólny algorytm, który pozwala nam dokonać dowolnej agregacji wartości na dowolnej liście. Jak dokładnie agregacja będzie przebiegała określimy dopiero podczas aplikacji funkcji `agregacja`. Na przykład:

```
let suma a b = a+b
let iloczyn a b = a*b

let listaInt = Element(1,Element(2,Element(3,
                 Element(4,Pusta))))

agregacja suma 0 listaInt //10
agregacja iloczyn 1 listaInt //24
```

Zwróćmy uwagę, że parametr `agregat` został rozpoznany jako funkcja generyczna. Wobec tego jedynymi wymaganiami co do typów parametrów funkcji, którą można przekazać do operacji `agregacja` jest to, aby pierwszy był tego samego typu co wartość początkowa i wynik funkcji, a drugi tego samego typu co elementy na liście.

```
let listaFloat = Element(1.0, Element (2.0,
                    Element (3.0, Element(4.0, Pusta))))
let listaString = Element("Ala", Element ("ma",
                     Element ("kota", Pusta)))

let sumaFloat a b = a+b
let polacz a b = a+" "+b

agregacja sumaFloat 0.0 listaFloat //10.0
agregacja polacz "" listaString //" Ala ma kota"
```

Do zdefiniowania powyższych agregacji wykorzystałem funkcje, które miały wcześniej przypisane nazwy. Tworzenie takich funkcji ma sens tylko gdy będziemy z nich korzystać wielokrotnie w aplikacji. Jeżeli jednak funkcja jest prosta skorzystać z wyrażeń lambda (funkcji anonimowych, które definiujemy w miejscu wykorzystania):

```
agregacja (fun a b->a+b) 0 listaInt //10
agregacja (fun a b->a+b) 0.0 listaFloat //10.0
agregacja (fun a b->a+" "+b) "" listaString // " Ala ma kota"
```

Takie działanie, może oczywiście prowadzić do niewielkiej nadmiarowości w kodzie, ale w przypadku prostych funkcji nie będzie to stanowiło większego problemu. Zyskaliśmy natomiast na czytelności aplikacji, ponieważ aby zrozumieć powyższe funkcje nie trzeba ”skakać” po kodzie, tylko wszystko mamy w jednym miejscu.

Dzięki temu, że w F# dowolny operator możemy potraktować jak funkcje, powyższy kod można przedstawić nawet prościej:

```
agregacja (+) 0 lista
agregacja (+) 0.0 lista1
```

Stwórzmy teraz funkcję, która będzie realizowała zadanie filtrowania listy:

```
let rec filtruj predykat = function
  | Pusta -> Pusta
  | Element(glowa, ogon) when predykat(glowa)
      -> Element(glowa, filtruj predykat ogon)
  | Element(_, ogon) -> filtruj predykat ogon
```

Jest to funkcja wyższych rzędów, którą można wykorzystać np. w następujący sposób:

```
let lista = generujZZakresu (-5, 5)
filtruj (fun x->x%2=0) lista
```

Zwróćmy jednak uwagę na poniższe aplikacje:

```
filtruj (fun x->x>0) lista
filtruj (fun x->x>3) lista
filtruj (fun x->x>-2) lista
```

We wszystkich przypadkach przekazywane predykaty są do siebie bardzo podobne. Różnią się tylko wartością, z którą porównywane są kolejne elementy listy. Ktoś mógłby więc powiedzieć, że w tym przypadku znów mamy do czynienia z nadmiarowością.

Oczywiście nie możemy napisać funkcji w stylu:

```
let wiekszeNiz x y = x>y
```

i przekazać jej do funkcji filtruj, ponieważ ta ostatnia spodziewa się funkcji tylko jednego, a nie dwóch parametrów. Możemy jednak wykorzystać fakt, że funkcje wyższych rzędów mogą nie tylko przyjmować inne funkcje jako parametry, ale również je zwracać. Uzyskujemy wtedy generator funkcji:

```
let wiekszeNiz n = fun x->x>n
```

W powyższym przykładzie zdefiniowałem funkcję jednego parametru `wiekszeNiz`. W wyniku jej działania uzyskujemy nową funkcję - w tym przypadku również jednego parametru - która porównuje porównania.

Zobaczymy tę funkcję w działaniu:

```
let predykat = wiekszeNiz 3;;
predykat 5;;
predykat 2;;
```

w rezultacie w interpreterze zobaczymy:

```
val predykat : (int -> bool)
val it : bool = true
val it : bool = false
```

Czyli `predykat` jest tak jak się spodziewamy funkcją przyjmującą jeden argument typu `int` i zwracającą wartość logiczną `bool`. Jej działanie również jest zgodnie z naszymi oczekiwaniami, co pokazują następne dwie aplikacje.

Pamiętajmy jednak, jak działa aplikacja funkcji. Jeżeli dokonamy następującego wywołania: `wiekszeNiz 3`, to na stosie zostanie utworzona ramka aktywacji dla tej funkcji, która będzie zawierała m.in. wartość argumentu `n`. Funkcja ta tworzy nową funkcję, która ponieważ jest w zakresie funkcji `wiekszeNiz` ma dostęp do wartości jej parametrów i zmiennych lokalnych. Następnie funkcja ta jest zwracana jako rezultat i usuwana jest utworzona ramka aktywacji dla funkcji `wiekszeNiz`. Powoduje to również usunięcie ze stosu wartości jej parametrów. Jednak w "magiczny" sposób funkcja ta pamięta wartość parametru `n` i potrafi poprawnie wykonywać powierzone jej działania. Jest to możliwe ponieważ jako rezultat funkcji `wiekszeNiz` nie jest zwracana typowa funkcja, tylko tzw. **dopełnienie**, czyli kod funkcji wraz z wartościami wszystkich symboli, które podczas wywołania mogą już nie być normalnie dostępne.

Zdefiniowaną funkcję `wiekszeNiz` możemy teraz wykorzystać do generowania predykatów dla funkcji `filtruj` w następujący sposób:

```
filtruj (wiekszeNiz 0) lista
filtruj (wiekszeNiz 3) lista
filtruj (wiekszeNiz -2) lista
```

4.2 Listy

Listy w F# są zaimplementowane jako typ `List<'a>`. Jest ona zaimplementowana jako lista łączona - podobna do tej jaką implementowaliśmy na poprzednich zajęciach. [1].

Listy możemy definiować jako:

Kod 4.1: Definicja listy z określeniem pełnej nazwy typu

```
let lista:List<int> = ...
```

To samo możemy jednak zapisać prościej:

Kod 4.2: Definicja listy z wykorzystaniem aliasu typu

```
let lista:int list = ...
```

Ten drugi zapis jest zdecydowanie bardziej popularny i z niego będziemy od tej pory korzystać.

4.2.1 Tworzenie list

Najprostszym sposobem na utworzenie listy jest wymienienie w nawiasach kwadratowych wszystkich jej elementów oddzielając je średnikami:

[1] C# również zawiera typ o podobnej nazwie, jednak jest ona implementowana jako tablica dynamiczna – opisana w punkcie 4.4

Kod 4.3: Definiowanie elementów na liście

```
let lista = [1;2;3;4;5]
```

Oczywiście typ elementów listy może być dowolny, przykładowo:

Kod 4.4: Definiowanie listy rekordów

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

let lista = [
  {imie="Ala", nazwisko="Kot", wiek=23};
  {imie="Ewa", nazwisko="Nowak", wiek=34}
]
```

Jeżeli poszczególne elementy listy będziemy wymieniali w osobnych liniach, jak w powyższym przykładzie, to średniki są opcjonalne.

Oczywiście, jeżeli będziemy chcieli uzyskać listę pustą, to nie podajemy jej elementów:

Kod 4.5: Definiowanie listy pustej

```
let listaPusta = []
```

Do określania elementów jakie mają znaleźć się na liście można również wykorzystać operator zakresu `..`:

Kod 4.6: Definiowanie listy z wykorzystaniem zakresów

```
let lista = [1..10]
let lista = [1..2..10]
```

Zakres określany jest poprzez podanie najmniejszej i największej wartości jaka może się znaleźć na liście oraz opcjonalnego kroku z którymi kolejne wartości będą generowane. Czyli powyższe instrukcje stworzą odpowiednio listy `[1;2;3;4;5;6;7;8;9;10]` oraz `[1;3;5;7;9]`.

Kolejnym sposobem jest wykorzystanie mechanizmy skracania list (ang. *list comprehension*):

Kod 4.7: Skracanie list w F#

```
let lista = [for i in 1..10 -> 2*i]
```

W tym przypadku zakres służy do generowania kolejnych liczb, które są podstawiane pod symbol `i` i wykorzystane do obliczenia wartości elementu, który zostanie umieszczony na liście.

Wszystkie powyższe metody możemy wykorzystać w przypadku, gdy jesteśmy wstanie wymienić wszystkie elementy listy lub określić sposób ich generowania już na etapie jej tworzenia. Jeżeli jednak kolejne elementy będziemy chcieli dodawać do listy w sposób inkrementacyjny wtedy możemy wykorzystać tzw. operator cons `::` :

Kod 4.8: Dołączanie nowego elementu listy za pomocą operatora `::`

```
let lista = [1;2;3]
let lista1 = 0::lista
```

W tym przykładzie najpierw jest tworzona lista trzyelementowa, a następnie tworzona jest nowa lista, której pierwszym elementem będzie 0. Zwróćmy uwagę, że operator `::` (jak i inne operatory oraz funkcje działające na listach) zachowuje niemutowalność listy, więc lista wejściowa nie jest modyfikowana tylko tworzona jest nowa, jak to było pokazane na poprzednich zajęciach.

4.2.2 Właściwości list

Po utworzeniu listy mamy dostęp do następujących właściwości:

Nazwa	Opis
`Head`	zwraca pierwszy element listy
`IsEmpty`	zwraca wartość `true` jeżeli lista jest pusta, `false` w przeciwnym przypadku
`Length`	zwraca liczę elementów listy
`Tail`	ogon listy (listę bez pierwszego elementu)

Przykładowo:

Kod 4.9: Dostęp do właściwości list

```
let lista = [1;2;3]
lista.Head //1
lista.Length //3
```

Podobnie możemy dostać się do dowolnego elementu listy za pomocą operatora nawiasów kwadratowych[2]. Przykładowo:

Kod 4.10: Dostęp do elementu listy za pomocą operatora `[]`

```
lista.[0]
```

[2]W najnowszej wersji platformy .NET możemy zrezygnować z tej denerwującej kropki

Musimy pamiętać, że podobnie jak w innych językach programowania elementy listy numerowane są od 0.

Wykorzystanie tych elementów jest jednak bardziej charakterystyczne dla programowania obiektowego niż funkcyjnego. Dlatego każda z powyższych właściwości, jak również operator dostępu do elementu są dostępne jako funkcje modułu `List` np.

Kod 4.11: Wykorzystanie funkcji modułu `List`

```
List.head lista
List.length lista
List.item 0 lista
```

4.2.3 Wykorzystanie list

Podobnie jak na poprzednich zajęciach operacje na listach możemy zdefiniować w sposób rekurencyjny. Przykładowo operacja sumowania elementów listy liczb całkowitych możemy zdefiniować jako:

Kod 4.12: Rekurencyjna funkcja sumująca elementy na liście

```
let rec suma = function
| [] -> 0
| glowa::ogon -> glowa + (suma ogon)
```

Zwróćmy uwagę, że funkcja ta jest identyczna jak zdefiniowana przez nas na poprzednich zajęciach - różnią się ona tylko zastosowanymi wzorcami. W powyższym przykładzie zastosowane zostały wzorce dla list. Pierwszy dopasowuje się do listy pustej, a drugi do listy składającej się z głowy i ogona.

Na szczęście F# został wyposażony w bardzo dużą liczbę funkcji wykonujących operacje na listach, przez co w większości zastosowań nie musimy tworzyć własnych funkcji, tylko możemy wykorzystać już istniejące. Przykładowo w celu utworzenia 100-elementowej listy liczb całkowitych wygenerowanych losowo z przedziału od -50 do 50, z których następnie wybierzemy liczby dodatnie oraz obliczymy ich kwadraty, a następnie zsumujemy możemy zastosować następujący kod:

Kod 4.13: Przykładowe operacje wykonywane na liście

```
let gll = new Random ()
let lista1 = List.init 100 (fun i->gll.Next(-50,51))
let lista2 = List.filter (fun v->v>0) lista1
let lista3 = List.map (fun v->v*v) lista2
List.fold (fun a c->a+c) 0 lista3
```

Pełna lista funkcji dostępnych w module `List` jest przedstawiona w dodatku do tej instrukcji.

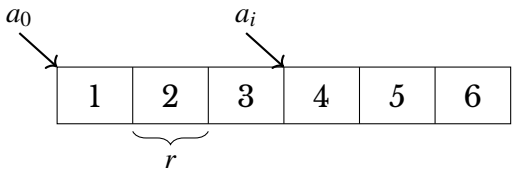
4.3 Tablice

Pokazane w poprzedniej sekcji listy łączone są bardzo dobrą strukturą, gdy potrzebujemy niemodyfikowalnej kolekcji danych. Operacje dodawania lub usuwania elementu z jej początku zaimplementowane są bardzo efektywnie. Czas modyfikowania elementów rośnie wraz z ich indeksem [3]. Problemem może stanowić operacja odczytu dowolnego elementu. Ponieważ każda pozycja na liście może znajdować się w innym miejscu w pamięci, jeżeli będziemy wymagali efektywnego dostępu do dowolnego elementu kolekcji lista łączona może nie być najlepszym wyborem.

W tym zadaniu zdecydowanie lepiej sprawdzą się tablice. Jest to kolekcja danych o modyfikowalnych elementach i stałym rozmiarze. Przedstawiona jest ona na rys. 4.1. Dzięki temu, że wszystkie dane w tablicy przechowywane są w ciągłym obszarze pamięci określenie adresu, pod którym dostępny jest dany element sprowadza się do wykonania prostych obliczeń:

$$a_i = a_0 + r * i$$

gdzie a_i to adres i-tego elementu ($i = 0..n-1$), a_0 określa adres początku danych w tablicy, n liczbę jej elementów, a r rozmiar elementu.



Rysunek 4.1: Tablica

W języku F# tablice są homogeniczną kolekcją, którą można tworzyć w sposób podobny do opisywanych wcześniej list:

```
let tablica1 = [| 1;2;3 |]
let tablica2 = [| 1..10 |]
let tablica3 = [| for i in 1..10 -> 10 * i |]
```

Zwróćmy uwagę, że w tym przypadku inna jest forma operatora – zamiast samych nawiasów kwadratowych podajemy `[||]`.

Innym sposobem tworzenia tablic jest zastosowanie jednej z funkcji modułu `Array` tj. np. `Array.empty`, `Array.create`, `Array.zeroCreate`. Pierwsza z nich tworzy pustą tablicę, druga tablicę zawierającą n-krotnie powtórzoną podaną wartość, a trzecia tablicę zawierającą n-krotnie powtórzoną wartość domyślną dla danego typu danych. Przykładowo wydając następujące instrukcje

```
let tablica1:int array = Array.empty
let tablica2:int array = Array.create 10 1
```

[3] zwłaszcza jeżeli będziemy chcieli zapewnić niemodyfikowalność

```
let tablica3:int array = Array.zeroCreate 10
```

otrzymamy następujące tablice:

```
val tablica1: int array = [||]
val tablica2: int array = [|1; 1; 1; 1; 1; 1; 1; 1; 1; 1|]
val tablica3: int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

Do poszczególnych elementów tablicy możemy dostać się za pomocą notacji nawiasów kwadratowych [4]:

```
tablica3.[2]
```

Pamiętajmy, że tablice są indeksowane od 0. Ciekawą cechą F# jest możliwość tworzenia tablic poprzez określanie zakresu indeksów wartości, które chcemy skopiować do nowej tablicy. Poniższe polecenia:

```
let tablica = [|1..10|]
let fragment1 = tablica.[3..5]
let fragment2 = tablica.[..5]
let fragment3 = tablica.[3..]
```

uzyskamy:

```
val fragment1: int[] = [|4; 5; 6|]
val fragment2: int[] = [|1; 2; 3; 4; 5; 6|]
val fragment3: int[] = [|4; 5; 6; 7; 8; 9; 10|]
```

Na tablicach jednowymiarowych w F# możemy stosować funkcje zdefiniowane w module `Array`.

4.3.1 Tablice wielowymiarowe

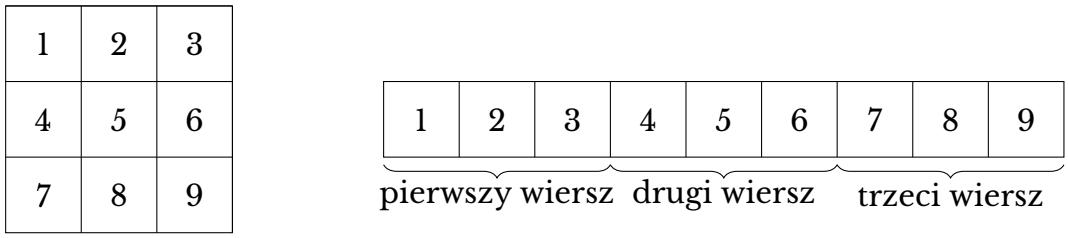
F# wspiera tablice wielowymiarowe (do 4 wymiarów). Dane dalej są przechowywane w ciągłym obszarze pamięci, przy czym zapisywane są od najbardziej wewnętrznych indeksów (w przypadku macierzy, czyli tablicy dwuwymiarowych dane będą zapisywane wierszami). Graficzna reprezentacja tablicy dwuwymiarowej w pamięci przedstawiona jest na rys. 4.5

Adres dowolnego elementu macierzy możemy określić za pomocą wzoru:

$$a_{ij} = a_0 + r * (i * c + j)$$

Podobnie można zdefiniować wzór na określenie adresu w przypadku tablic trzy- lub czterowymiarowych.

[4] Podobnie jak w przypadku list w .NET 6 nie ma już potrzeby podawania kropki pomiędzy nazwą tablicy, a operatorem nawiasów kwadratowych



Rysunek 4.2: Tablica dwuwymiarowa

F# nie posiada składni wspierającej tworzenie tablic wielowymiarowych. Dla macierzy możemy zastosować funkcje `array2D`, która umożliwia utworzenie tablicy dwuwymiarowej z sekwencji sekwencji:

```
let macierz = array2D [[1;2];[3;4]]
```

W rezultacie zobaczymy:

```
val macierz1: int[,] = [[1; 2]
                        [3; 4]]
```

Tablice trzy i czterowymiarowe możemy tworzyć tylko z wykorzystaniem odpowiednio funkcji `Array3D.init` oraz `Array4D.init`:

```
let macierz3D = Array3D.init 4 4 4 (fun i j k -> i*j*k)
```

W tym przypadku interpreter F# pokaże nam jednak:

```
val macierz3D: int[,,] = [|rank=3|]
```

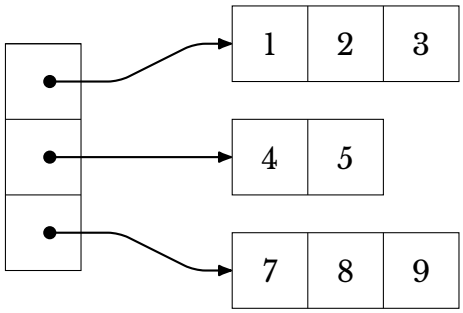
Jak widzimy, dla tablic 3D (podobnie dla 4D) F# nie pokazuje już elementów tablicy, tylko wyświetla liczbę wymiarów.

Do wykonywania podstawowych operacji na tablicach wielowymiarowych w F# możemy zastosować funkcje znajdujące się w modułach `Array2D`, `Array3D` oraz `Array4D`.

4.3.2 Tablice tablic

Opisywane w poprzedniej sekcji tablice wielowymiarowe są przechowywane w ciągłym obszarze pamięci, co umożliwia bardzo szybki dostęp do dowolnego ich elementu. W przypadku tablic dwuwymiarowych wymaga to jednak, aby każdy wiersz zawierał dokładnie tyle samo elementów (dlatego nazywane są one również tablicami prostokątnymi). Czasami jednak, będziemy chcieli, aby poszczególne wiersze miały różną długość. Graficzna reprezentacja takiej tablicy jest pokazana na rys. 4.3.

Ze względu na możliwą różną długość poszczególnych wierszy tablice takie nazywane są postrzępionymi (ang. *jagged array*). Wymagają one trochę większej ilości



Rysunek 4.3: Tablica tablic

pamięci, gdyż oprócz jednowymiarowych tablic z danymi mamy także jednowymiarową tablicę wskaźników do poszczególnych wierszy. Nie mamy również gwarancji, że poszczególne wiersze będą umieszczone w pamięci obok siebie, przez co operacje na takich tablicach mogą być wolniejsze niż w przypadku tablic prostokątnych.

W F# tablice takie możemy stworzyć w następujący sposób:

Kod 4.14: Tworzenie tablicy postrzępionej

```
let tablicaTablic = [|[|1;2;3|];
                      [|4;5|];
                      [|7;8;9|];
                     |];
```

Interpreter pokaże nam następujący rezultat tej instrukcji:

```
val tablicaTablic: int[][] = [|[|1; 2; 3|]; [|4; 5|]; [|7; 8; 9|]|]
```

Dostęp do poszczególnych jej elementów uzyskujemy za pomocą operatora nawiasów kwadratowych. W tym przypadku, każdy wymiar ma swój zestaw nawiasów:

Kod 4.15: Dostęp do elementów tablicy postrzępionej

```
tablicaTablic.[0].[1]
```

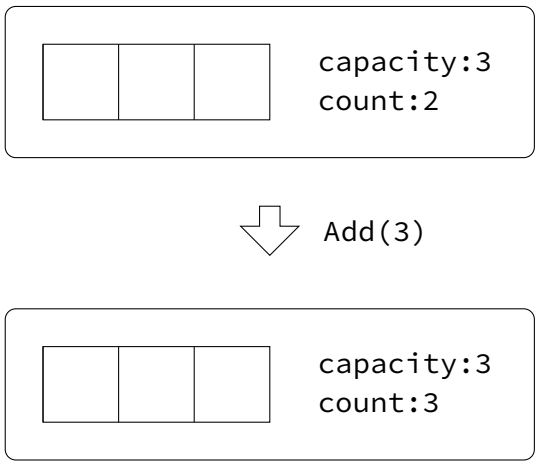
Jakie to szczęście, że w .NET 6 nie musimy już pisać kropek. Domyślnie w F# nie ma modułu zawierającego funkcje działające na tego typu macierzach.

4.4 Tablice dynamiczne

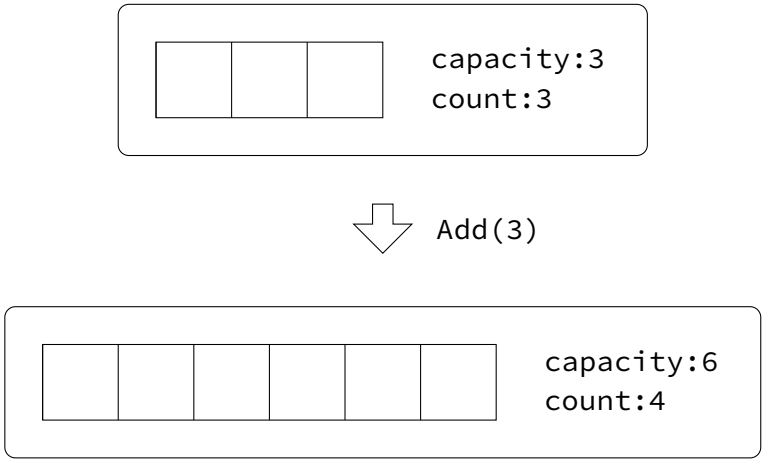
Tablicę dynamiczną możemy określić jako strukturę, która funkcjonalnie znajduje się pomiędzy tablicami, a listami łączonymi. Implementowane są one jako tablice, jednak posiadają również metody pozwalające na dodawanie i usuwanie elementów. Dzięki temu, że wewnętrznie jest to ciągły obszar pamięci operacja dostępu do dowolnego elementu tablicy dynamicznej jest wykonywana w tym samym czasie jak w zwykłej

tablicy. Czas konieczny na dodawanie lub usuwanie elementu jest jednak zależny od miejsca, w którym element ten się znajduje oraz czy konieczne jest zwiększenie rozmiaru tablicy.

4.4.1 Dodawanie / usuwanie elementu z ostatniej pozycji



Rysunek 4.4: Tablica



Rysunek 4.5: Tablica

4.4.2 Dodawanie / usuwanie elementu z pozycji innej niż ostatnia

4.5 Tablice

Tablice w F# są kolekcją elementów tego samego typu umieszczonymi w ciągłym obszarze pamięci o stałym rozmiarze. Nie jest to jednak typ danych, który najlepiej wpisuje się w paradygmat programowania funkcyjnego. Przede wszystkim jest to kolekcja mutowalna, co oznacza, że w przeciwieństwie do list można zmienić wartość przechowywaną w tablicy. Jest to związane głównie z względami wydajnościowymi.

Kopiowanie całej kolekcji podczas modyfikowania pojedynczego elementu byłoby ogromnym marnotrawstwem czasu i pamięci.

Szerzej tablice opisane są m.in. pod adresem:
https://docs.microsoft.com/pl-pl/dotnet/fsharp/language-reference/arrays.
My nie będziemy się zajmować tym tematem na naszych zajęciach.

4.6 Zbiory

Zbiory są kolekcją nieuporządkowaną. Oznacza to, że w przeciwieństwie do list nie zachowują porządku wprowadzania danych. Nie umożliwiają również przechowywania duplikatów elementów.

4.6.1 Tworzenie zbiorów

Podstawową metodą tworzenia zbiorów jest wykorzystanie funkcji `set`. Funkcja ta może przyjąć jako parametr dowolną kolekcję policzalną:

```
let zbior1 = set [1;2;3;4;5]
let zbior2 = set [1..10]
let zbior3 = set [for i in 1..10 -> 2*i]
```

Po utworzeniu zbioru, kolejne elementy możemy do niego dodawać za pomocą funkcji `add` modułu set:

```
let zbior4 = Set.add 6 zbior1
```

Ponieważ podobnie jak w przypadku list, zbiory są kolekcją niemodyfikowalną, funkcja `add` nie modyfikuje istniejącego zbioru, tylko tworzy nowy.

Ponieważ zbiory są kolekcją nie pozwalającą na przechowywanie duplikatów nie zawsze liczba elementów w zbiorze będzie się równać liczbie elementów kolekcji wejściowej. Przykładowo instrukcja:

```
let zbior = set [1;2;3;4;5;1;2;3]
```

stworzy zbiór zawierający tylko pięć elementów $[1;2;3;4;5]$ pozostałe elementy, jako duplikaty nie będą w nim umieszczone:

```
val zbior : Set<int> = set [1; 2; 3; 4; 5]
```

Zbiór pusty możemy stworzyć za pomocą funkcji `Set.empty`. Jest to użyteczne podczas inkrementacyjnego tworzenia zbioru.

Ponieważ zbiory są kolekcją nieuporządkowaną, nie istnieje operator, który pozwoli dostać się do określonego elementu zbioru na podstawie jego indeksu.

Wszystkie operacje jakie możemy wykonywać na zbiorach są zawarte w module `Set`.

4.7 Mapy

Mapy są strukturą podobną do zbiorów, ale pozwalają na połączenie elementu z jego kluczem. Wobec tego elementami mapy są pary, które można przedstawić w formie (klucz, wartość). Zwróćmy uwagę, że w tym względzie mapy są podobne do tablic w których kluczem jest indeks elementu w tablicy, a wartością - element przypisany do tego indeksu. O ile w przypadku tablic, kluczem jest zawsze indeks - czyli najczęściej liczba naturalna, o tyle w przypadku map może to być dowolna wartość wspierająca operator równości. Jest on wymagany ponieważ w mapie nie może być dwóch elementów o tym samym kluczu.

Mapy możemy tworzyć na podstawie tablicy, listy lub sekwencji wykorzystując do tego jedną z funkcji modułu `Map`: `ofArray`,`ofList`,`ofSeq`. Funkcje te jako parametr przyjmują odpowiednie kolekcje kolekcję par (klucz, wartość), przykładowo:

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

Podobnie jak w przypadku zbiorów nowy element możemy - z zachowaniem niemodyfikowalności - dodawać do mapy za pomocą funkcji `Map.add`:

```
let nowaMapa = Map.add "Genowefa" 8 mapa
```

Klucz powinien jednoznacznie wskazywać na dany element w mapie, dlatego w ramach jednej struktury nie może on się powtórzyć. Próba dodania do dwukrotnie tego samego klucza (bez względu, czy będzie przypisany do tych samych czy różnych wartości) nie będzie miała żadnego widocznego efektu.

Do pojedynczego elementu mapy możemy dostać się za pomocą operatora `.[]`, gdzie w nawiasach kwadratowych podajemy wartość klucza. Przykładowo podając:

```
mapa.["Ala"]
```

uzyskamy w rezultacie wartość:

```
3
```

Inne operacje jakie można wykonywać na mapach są zdefiniowane w module `Map`.

4.8 Sekwencje

Sekwencje w F# są specjalnym rodzajem kolekcji, w której elementy są zwracane w kolejności ale tylko wtedy, gdy są potrzebne. Jest to więc kolekcja leniwa, w przeciwieństwie do np. list, gdzie musimy wyznaczyć wszystkie elementy zanim otrzymamy kolekcję z którą będziemy mogli wykonać jakieś działania.

Przykładowo załóżmy, że chcielibyśmy obliczyć silnię dla pierwszych 20000 liczb całkowitych, a następnie wyświetlić ją na ekranie. Wykorzystując listy możemy to zrobić w następujący sposób:

```
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
```

W powyższym programie zdefiniowałem znaną już wam funkcję `silnia`, a następnie wykorzystałem ją do stworzenia listy 20000 par określających wartość n oraz odpowiadającą jej wartość funkcji `silnia`[5]. Następnie wykorzystując funkcję `iter` wyświetlam każdy element listy na ekranie.

Jeżeli uruchomimy ten program w interpreterze, to spokojnie możemy iść na kawę, zanim zobaczymy pierwszy rezultat. Jest to spowodowane tym, że cała lista musi być najpierw zbudowana (zajmując czas i miejsce w pamięci), zanim będziemy mogli coś zrobić z pierwszym jej elementem.

Jeżeli jednak nasz program napiszemy w następujący sposób:

```
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
```

Zauważymy, że kolejne wyniki będą wyświetlane jak tylko zostaną wyznaczone.

Zwróćmy uwagę, że w tym przypadku zamiast tworzyć listę za pomocą operatora `[]`, zastosowałem konstrukcję `seq {}`. Pozwala ona na zdefiniowanie przepisu jak wyznaczyć kolejne wartości w kolekcji danych, bez konieczności uruchomienia tego

[5]Ja wiem, że jest to najmniej efektywny sposób rozwiązania tego zadania, ale nie w tym rzecz aby teraz było to wydajne.

procesu. Dopiero funkcja `Seq.iter`, która chce skorzystać z tych danych, powoduje, że rozpoczyna się ich generowanie. Przy czym nie są one generowane wszystkie na raz, tylko po kolei, co pozwala tej funkcji pobierać z sekwencji kolejne wartości i wyświetlać je na ekranie.

Sekwencje mogą być bardzo przydane podczas wykonywania wielu operacji na innych strukturach danych. Jeżeli przykładowo chcemy wybrać z kolekcji dane spełniające określone przez nas warunki, posortować te dane, wybrać z nich odpowiednie informacje i wyświetlić je na ekranie możemy to przykładowo zrobić w następujący sposób:

```
type Student = {
    imie:string;
    nazwisko:string;
    wiek:int;
    rokStudiow:int;
}

[<EntryPoint>]
let main argv =

  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

  0
```

W powyższym programie wszystkie wymienione wcześniej operacje realizuję wykorzystując funkcje modułu `List`. Powoduje to jednak, że każde wywołanie funkcji `filter`, `sortBy`, `map` tworzy nową listę, która jest umieszczona w pamięci. Jednak listę jako wynik przetwarzania potrzebujemy tylko na końcu tego procesu, tuż przed jej wyświetleniem na ekranie. Wykorzystując sekwencję, ten sam program możemy napisać w następujący sposób:

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

  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

  0
```

Zauważmy, że w powyższym programie lista jest budowana tylko w dwóch miejscach: na początku przy stworzeniu danych wejściowych oraz podczas wywołania funkcji `Seq.toList`. W rezultacie przy przetwarzaniu dużych list danych powinniśmy zobaczyć wzrost wydajności naszej aplikacji.

Dodatkowo zamieniając wywołanie: `List.iter (fun o->printfn "%A"o)wynik` na `Seq.iter (fun o->printfn "%A"o)dane`, w ogóle pozbędziemy się konieczności budowania listy wynikowej, co pozwoli jeszcze odrobinę zmniejszyć czas wykonania naszej aplikacji. Jednak ostatnia zmiana, przy dużych listach wejściowych znacząco poprawi odczucia użytkownika, ponieważ aby zobaczyć wyniki nie będzie on musiał czekać, aż całkowicie zakończy się proces przetwarzania danych, tylko wyniki będzie mógł zobaczyć już w takcie trwania tego procesu (podobnie jak w pokazywanym wcześniej przykładzie z silnią).

4.9 Użyteczne funkcje

4.9.1 Obsługa plików w F#

W F# pliki są obsługiwane poprzez typy umieszczone w przestrzeni nazw: `System.IO`. Najważniejszy z naszego punktu widzenia jest typ `File`, który zawiera m.in. funkcje:

1. `File.CreateText nazwaPliku` - tworzy nowy pusty plik tekstowy o podanej nazwie (nazwa może zawierać ścieżkę dostępu) i zwraca obiekt umożliwiający wprowadzanie łańcuchów znaków do strumienia.

2. `File.ReadAllLines nazwaPliku` - wczytuje plik tekstowy o podanej nazwie i zwraca tablicę stringów. Każda pozycja w tablicy odpowiada jednej linii tekstu w pliku
3. `File.ReadAllText nazwaPliku` - wczytuje plik tekstowy o podanej nazwie. Całą zawartość pliku zwraca jako jeden łańcuch znaków
4. `File.ReadLines nazwaPliku` - wczytuje plik tekstowy o podanej nazwie i zwraca sekwencję pozwalającą na dostęp do poszczególnych linii w pliku.
5. `File.WriteAllLines nazwaPliku linie` - zapisuje do pliku tekstowego o podanej nazwie zawartość wszystkich linii podanych w tablicy `linie`. Jeżeli plik nie istnieje - tworzy go, jeżeli istnieje - nadpisuje.
6. `File.WriteAllText nazwaPliku str` - zapisuje do pliku tekstowego o podanej nazwie tekst przekazany poprzez parametr `linie`. Jeżeli plik nie istnieje - tworzy go, jeżeli istnieje - nadpisuje.
7. `File.AppendAllLines nazwaPliku linie` - dopisuje do pliku tekstowego o podanej nazwie zawartość wszystkich linii podanych w tablicy `linie`. Jeżeli plik nie istnieje - tworzy go.
8. `File.WriteAllText nazwaPliku str` - dopisuje do pliku tekstowego o podanej nazwie tekst przekazany poprzez parametr `linie`. Jeżeli plik nie istnieje - tworzy go.

Przykładowo załóżmy, że plik ”punkty.txt” zawiera kolekcję punktów, dla których chcemy sprawdzić czy znajdują się wewnątrz okręgu o podanych przez użytkownika parametrach (środku i promieniu). Jeżeli punkt jest w środku, to wyświetlamy go na ekranie. Każdy punkt w pliku opisany jest w osobnej linii jako dwie współrzędne x i y oddzielone od siebie spacją.

Zadanie to można zrealizować w następujący sposób:

```
open System
open System.IO

type Okrag = {
    srodek : float*float;
    promien : float;
}

let zamienNaPunkt (ls:string[]) = (float ls.[0], float ls.[1])

let wczytajPunkty nazwaPliku =
  let linie = File.ReadAllLines nazwaPliku
  let lancuchy = Seq.map (fun (l:string)->l.Split(' ')) linie
  let punkty  = Seq.map zamienNaPunkt lancuchy
  Seq.toList punkty
```

```
let wczytajWartosc komunikat =
  printf "%s" komunikat
  Console.ReadLine()

let wczytajOkrag () =
  let x = float (wczytajWartosc "Podaj wspolrzedna x: ")
  let y = float (wczytajWartosc "Podaj wspolrzedna y: ")
  let r = float (wczytajWartosc "Podaj promien: ")
  {srodek = (x,y); promien = r}

let odleglosc (x1,y1) (x2,y2) =
  sqrt ((x1-x2)**2.0+(y1-y2)**2.0)

let wSrodku okrag punkt =
  (odleglosc okrag.srodek punkt) < okrag.promien

let pokazPunkt (x,y) = printfn "%f %f" x y

[<EntryPoint>]
let main argv =

  let okrag = wczytajOkrag ()
  let punkty = wczytajPunkty "punkty.txt"

  let punktyWSrodku =
     Seq.filter (fun p -> wSrodku okrag p) punkty

  Seq.iter pokazPunkt punktyWSrodku
  0
```

Zwróćmy szczególną uwagę na funkcję `wczytajPunkty`. Jako argument przyjmuje ona nazwę pliku do wczytania. Ponieważ każdy punkt zapisany jest w osobnej linii wykorzystałem metodę `ReadAllLines`, która odczytała plik i całą jego zawartość zwróciła jako tablicę. Następnie każdy element tej tablicy został rozdzielony na dwie części względem spacji i zamieniony na parę liczb całkowitych opisujących pojedynczy punkt. Aby nie tworzyć pośrednich kolekcji wykorzystałem sekwencje wobec tego na samym końcu z sekwencji tej stworzyłem listę.

W funkcji tej dwa elementy mogą wydawać się niepotrzebne. Po pierwsze wczytujemy całą zawartość pliku na raz. Jednak jeżeli plik ten będzie bardzo duży może się okazać, że nie zmieści się w pamięci. Działanie to jest jednak zupełnie nie potrzebne ponieważ w tym przypadku na raz nie rozpatrujemy więcej niż jednego punktu. Wobec tego możemy wczytać jeden punkt, przetworzyć go i dopiero pobrać następny.

Wymaga to drobnych zmian w naszej funkcji:

```
let wczytajPunkty nazwaPliku =
  let linie = File.ReadLines nazwaPliku
  let lancuchy = Seq.map (fun (l:string)->l.Split(' ')) linie
  let punkty  = Seq.map zamienNaPunkt lancuchy
  punkty
```

Teraz wykorzystuję funkcję `ReadLines`, która nie wczytuje całego pliku na raz, tylko zwraca go linia po linii. Nie buduję również na końcu listy punktów tylko zwracam sekwencję. Dzięki tym prostym zabiegom nasza aplikacja będzie wymagała dużo mniejszej ilości pamięci.

Jeżeli na końcu programu wyniki zamiast wyświetlać na ekranie będziemy chcieli zapisać do pliku to funkcję `main` powinniśmy zmienić w następujący sposób:

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

  let okrag = wczytajOkrag ()
  let punkty = wczytajPunkty "punkty.txt"

  let punktyWSrodku =
    Seq.filter (fun p -> wSrodku okrag p) punkty
  use strumien = File.CreateText "wyniki.txt"
  let linie = Seq.map
    (fun p -> sprintf "%f %f" (fst p) (snd p))
    punktyWSrodku
  Seq.iter (fun (l:string) -> strumien.WriteLine(l)) linie
  0
```

W powyższym przykładzie pojawiło się nowe słowo kluczowe `use`, które realizuje wiązanie symbolu z wartością podobnie jak `let`, ale dodatkowo dla obiektów implementujących interfejs `IDisposable` automatycznie wywołuje metodę `Dispose`, gdy tylko obiekt ten przestaje być potrzebny. W tym przypadku wywołanie tej metody spowoduje zamknięcie strumienia i odblokowanie pliku dla innych aplikacji.

4.10 Zadania

Zadanie 4.1 Na poprzednich zajęciach tworzyliśmy listę od podstaw. Napisz funkcję wyższych rzędów `mapuj`, która będzie dokonywała dowolnego mapowania. Sposób zamiany elementów zdefiniuj jako parametr funkcji `mapuj`.

Zadanie 4.2 Napisz funkcję wyższych rzędów, która pozwoli dokonać dowolnej agregacji elementów zapisanych w drzewie.

Zadanie 4.3 Napisz funkcję wyższych rzędów, która sprawdzi, czy w drzewie umieszczona jest wartość spełniająca warunek przekazany jako parametr.

Zadanie 4.4 Napisz funkcję, która przyjmuje listę F# i buduje z niej drzewo binarne.

Zadanie 4.5 Napisz funkcję, która jako parametr przyjmuje uporządkowane drzewo binarne i zwraca listę F#

Zadanie 4.6 Stwórz listę losowych liczb całkowitych zawierającą wartości z przedziału od -10 do 10. Następnie wykorzystaj funkcje z modułu List w celu podzielenia tej listy na dwie części: pierwszą zawierającą wartości dodatnie, drugą zawierającą wartości ujemne. Z jakich funkcji możesz skorzystać, aby zrealizować to zadanie.

Zadanie 4.7 Stwórz listę losowych liczb całkowitych zawierającą wartości z przedziału od -10 do 10. Następnie wykorzystaj funkcje z modułu List w celu podzielenia tej listy na dwie części: pierwszą zawierającą wartości powyżej średniej, drugą zawierającą wartości poniżej średniej. Z jakich funkcji możesz skorzystać, aby zrealizować to zadanie.

Zadanie 4.8 Wczytaj plik ”zad8.txt”. Zawiera on zestaw współczynników dla równania kwadratowego (każde równanie jest opisane w osobnym wierszu). Podziel te parametry na trzy osobne listy w zależności czy równanie opisane tymi parametrami ma zero, jedno rozwiąznie, dwa rozwiązania lub inne.

Zadanie 4.9 Wczytaj plik ”zad9.txt”. W każdym wierszu zawiera on współrzędne punktu na płaszczyźnie dwuwymiarowej. Oblicz odległości pomiędzy dowolnymi dwoma punktami, a następnie zapisz do pliku ”rozwiązanie9.txt” pary tych punktów wraz z odległościami pomiędzy nimi posortowane rosnąco względem odległości.

Zadanie 4.10 Napisz program podobny do przedstawionego w punkcje 1.8 tylko podziel punkty na dwie części. Osobno te które są wewnątrz okręgu oraz te które są poza nim. Zapisz wyniki do dwóch osobnych plików. Wykorzystaj plik ”zad9.txt”.

Zadanie 4.11 Wczytaj plik ”iris.txt”. Opisuje on trzy gatunki kwiatów irysa: Setosa, Versicolour i Virginica. Każdy kwiat jest opisany za pomocą czterech liczb określających odpowiednio długość: listka kwiatu, szerokość listka kwiatu, długość płatka i szerokość płatka. Dla każdej kolumny danych wyznacz wartość minimalną, maksymalną oraz średnią.

Zadanie 4.12 Wczytaj plik ”iris.txt”. Podziel dane na osobne grupy zgodnie z gatunkami kwiatu, a następnie dla każdej grupy osobno wyznacz w każdej kolumnie wartość minimalną, maksymalną oraz średnią.

Zadanie 4.13 Stwórz listę losowych liczb całkowitych zawierającą wartości z przedziału od -10 do 10 (niech lista zawiera ok 1000 elementów). Określ ile razy każda z tych wartości wystąpiła na liście. Wyniki przedstaw jako mapę, gdzie kluczem jest liczba od -10 do 10, a wartością liczba wystąpień tej liczby.

Zadanie 4.14 Wczytaj plik ”iris.txt”. Dla każdej kolumny wyznacz jej histogram, a następnie zrób to samo, ale dla każdego gatunku kwiatu osobno.