Sprawdźmy na początku czy pamiętasz coś z ostatnich zajęć ;)

ZADANIE 1

Napisz funkcje rekurencyjne w języku F# obliczające:

a) n!

b) n-ty wyraz ciągu Fibonacciego

c) największy wspólny dzielnik dwóch liczb naturalnych (za pomocą odejmowania)

d) największy wspólny dzielnik dwóch liczb naturalnych (algorytm Euklidesa)

e) sumę cyfr podanej liczby

KROTKI

Krotka zwana również entką, to zbiór oddzielonych przecinkami wartości, ujęty w nawiasy okrągłe, np. `let krotka = (18, 26.58, "Adam", "Nowak").`

Jak widać każdy element krotki może mieć inny typ.

Krotki po utworzeniu są niezmienne.

Używa się ich do przechowywania współrzędnych punktów, wartości kolorów, formatowania danych, jako algorytm i wyniki funkcji.

Przykład

```
open System
let srednia (a, b) =
    let sum = a + b
    sum / 2.0
printfn "Wartość średnia: %.2f" (srednia(2.22, 3.68))
Console.ReadKey() |> ignore
```

Przykład

```
open System
let iloraz (x,y) =
    x / y, x % y
printfn "Wynik ilorazu w krotce (wynik dzielenia całkowitoliczbowego
wraz z resztą: %A" (iloraz (20, 3))

printfn "Wynik dzielenia: %d" (fst (iloraz (20, 3)))
printfn "Reszta z dzielenia: %d" (snd (iloraz (20, 3)))

Console.ReadKey() |> ignore
```

Jak widać powyżej funkcje fst i snd umożliwiają pobranie odpowiednio pierwszej i drugiej wartości krotki.

W celu wybrania dowolnej wartości z krotki stosujemy:

Przykład

```
open System
let krotka = ("Ola", "ma", "kota", "Bonifacego")
let (_,_,_,d) = krotka
printfn "Czwarty element: %s" d
Console.ReadKey() |> ignore
```

ZADANIE 2

Napisz funkcję sprawdzającą, czy podane dwie liczby mają taką samą resztę z dzielenia przez liczbę 4.

SEKWENCJE

Sekwencje stanowią logiczny ciąg elementów jednego typu – bardzo przydatne w dużych zbiorach danych.

Niekoniecznie wymagają użycia wszystkich elementów.

W sekwencji pojedyncze elementy obliczane są tylko w miarę potrzeby.

ZADANIE 3

Przepisz poniższy kod i sprawdź jego działanie:

```
open System
let seq1 = seq {1 .. 10 }
let seq2 = seq {0 .. 10 .. 100 }

let seq3 = seq { for i in 1 .. 10 do yield i * i}
let seq4 = seq { for i in 1 .. 10 -> i * i}
printfn "seq1: %A" seq1
printfn "seq2: %A" seq2
printfn "seq3: %A" seq3
printfn "seq4: %A" seq4
Console.ReadKey() |> ignore
```

Wybrane metody modyfikujące sekwencję:

Metoda	Opis
`append`	Łączy dwie sekwencje.
`average`	Zwraca wartość średnią elementów w sekwencji.
`averageBy`	Zwraca wartość średnią z wyników uzyskanych przez zastosowanie funkcji do każdego elementu sekwencji (np. konwersji typu).
`empty`	Tworzy pusty ciąg.
`init`	Tworzy nową sekwencję, z podanej liczby elementów, wyliczoną z podanej funkcji. Wyniki wywołanej funkcji nie są zapisywane. Funkcja jest wywoływana ponownie dla elementów niezbędnych do regeneracji.
`initInfinite`	Tworzy nową nieskończoną sekwencję. Elementy są tworzone w miarę potrzeb.
`iter`	Iteracja (obliczanie) danej funkcji na każdym elemencie sekwencji.
`singleton`	Zwraca ciąg z jednym elementem.
`skip`	Pomija określoną liczbę elementów sekwencji.
`skipWhile`	Pomija elementy spełniające podany warunek.
`sum`	Zwraca sumę elementów w sekwencji.
`truncate`	Obcina sekwencję do określonej liczby elementów.

Dodatkowo duży zestaw metod działających na sekwencjach oferuje biblioteka: Microsoft.FSharp.Collections.Seq

ZADANIE 4

Przetestuj powyższe (z tabeli) metody do modyfikacji sekwencji.

OPCJE

Typ opcji w języku F# jest używany, gdy aktualna wartość **nie może istnieć** dla nazywanej wartości lub zmiennej. Opcja może przechowywać wartości tego typu lub może nie mieć wartości.

Typ opcji może posiadać dwie wartości: Some(..) lub None.

Typy te są często używane do reprezentowania w obliczeniach wartości lub wskazania, że dane obliczenia udało się wykonać, czy też nie.

Przykład
Podczas dzielenia przez 0 typ Option przyjemnie wartość „None" a w pozostałych przypadkach wykona operację dzielenia:

```
let iloraz (x, y) =
    match y with
    | 0 -> None
    | _ -> Some (x/y, x%y)
```

Moduł Option zawiera przydatne metody do wykorzystania z użyciem opcji. Najbardziej użyteczne z nich to:

- isSome – zwraca wartość true, jeśli opcja ma wartość
- isNone – zwraca wartość true, jeżeli opcja nie ma wartości
- get – pobiera wartość z opcji

ZADANIE 5

Napisz funkcję, która zwróci resztę z dzielenia podanych dwóch liczb lub wartość pustą, jeśli reszta z dzielenia jest równa 0.

LISTY

Lista w języku F# jest uporządkowanym i niezmiennym szeregiem elementów tego samego typu.

Przykład

```
let lista = [1; 2; 3]
let lista2 = [
    1
    2
    3 ]
let lista3 = [1 .. 5]
let lista4 = [for i in 1..10 -> i * i ]
let lista5 = 1 :: 2 :: 3 :: 4 :: []

let lista6 = List.init 5 (fun x-> x*2)
```

Przykład

Dodawanie kolejnego elementu na początku:

```
let przykladowa = [1; 2; 3]
let y = 10::lista
```

otrzymamy wówczas: [10; 1; 2; 4]

ZADANIE 6

Przetestować poniższe kody:

a) właściwości:

```
printfn "Czy lista jest pusta : %b" (przykladowa.IsEmpty)
printfn "Długość listy : %d" (przykladowa.Length)
printfn "Pierwszy element listy : %d" (przykladowa.Head)
printfn "Drugi element listy : %d" (przykladowa.Tail.Head)
printfn "Trzeci element listy : %d" (przykladowa.Tail.Tail.Head)
printfn "Element o indeksie(1) : %d" (przykladowa.Item(1))
```

b) sortowanie listy:

```
let posortowana = List.sort [-4; 2; 9; -11; 0; 4]
printfn "Lista posortowana: %A" posortowana
```

c) wyszukiwanie pierwszego elementu listy spełniającego kryterium:

```
let szukana = List.find (fun x -> x % 79 = 0) [80 .. 200]
printfn "pierwsza liczba podzielna przez 17: : %d" szukana
```

d) suma elementów:

```
let suma = List.sum [5 .. 56]
```

```
printfn "suma: : %d" suma
let suma2 = List.sumBy (fun x -> x*x) [1 .. 10]
printfn "suma kwadratów elementów z listy: : %d" suma2
```

e) wartość średnia z elementów:

```
let srednia = List.average [3.5; 8.6; 4.3; 1.0]
printfn "srednia: : %.2f" srednia
let srednia2 = List.averageBy (fun x -> x*x) [3.5; 8.6; 4.3; 1.0]
printfn "srednia z kwadratów elementów z listy: %.2f" srednia2
```

f) łączenie list

```
let zip2 = List.zip lista lista2
printfn "połączenie dwóch: %A" zip2
```

g) utworzenie nowej na podstawie starej

```
let nowa = List.map (fun x -> x*x + 1) lista
printfn "nowa lista na podstawie istniejącej: %A" nowa
```

h) filtrowanie listy

```
let parzyste = List.filter (fun x -> x%2=0) [1..100]
printfn "parzyste: %A" parzyste
```

Mamy jeszcze więcej metod działających na listach. Oto niektóre z nich:

Metoda	Opis
append	Zwraca nową listę, która zawiera elementy pierwszej listy, a następnie elementy drugiej listy.
average	Zwraca wartość średnią elementów z listy.
averageBy	Zwraca wartość średnią elementów wygenerowanych poprzez zastosowanie funkcji do każdego elementu listy.
empty	Zwraca pustą listę danego typu.
exists	Sprawdza, czy jakiś element z listy spełnia dany predykat.
filter	Zwraca nowy zbiór zawierający tylko elementy kolekcji, dla których dany predykat zwróci true.
find	Zwraca pierwszy element, dla którego dana funkcja zwraca true.
findIndex	Zwraca indeks pierwszego elementu na liście, który spełnia dany predykat.
head	Zwraca pierwszy element listy.
init	Tworzy listę poprzez wywołanie danej funkcji dla każdego elementu.

`isEmpty`	Zwraca true jeśli lista nie zawiera żadnych elementów, inaczej – false.
`length`	Zwraca długość listy.
`map`	Tworzy nową listę, na podstawie starej, przetworzonej przez podaną funkcję.
`map2`	Tworzy nową listę na podstawie dwóch list z elementami przetworzonymi przez funkcję.
`max`	Zwraca największy ze wszystkich elementów listy.
`min`	Zwraca najmniejszy ze wszystkich elementów listy.
`ofArray`	Tworzy listę z danej tablicy.
`sort`	Sortuje listę.
`sortBy`	Sortuje listę – wg podanej funkcji.
`sum`	Zwraca sumę elementów listy.
`sumBy`	Zwraca sumę wyników uzyskanych przez zastosowanie funkcji do każdego elementu listy.
`tail`	Zwraca listę bez pierwszego elementu.
`toArray`	Tworzy tablicę z podanej listy.
`unzip`	Dzieli listę par krotek na krotkę dwóch list.
`zip`	Łączy dwie listy w listę par krotek. Obie listy muszą mieć równy długości.
`zip3`	Łączy trzy listy w listę trójelementowych krotek. Listy muszą być równej długości.

Pełny zestaw metod działających na listach zawiera metoda Microsoft.FSharp.Collections.Lists

TABLICE

Tablice są zmiennymi kolekcjami uporządkowanych elementów danych, tego samego typu, o stałej wielkości i numerowanymi od zera. Do tablic uzyskujemy swobodny dostęp poprzez ich indeksy (numery).

Przykład

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

Zaś dostęp do podanych elementów tablicy wygląda następująco:

```
tablica.[0]
tablica2.[0..2]
tablica3.[..2]
tablica3[2..]
```

Nie ma składni do definiowania tablicy wielowymiarowej. Mimo to takie tablice mogą być tworzone. W celu zdefiniowania tablicy dwuwymiarowej używamy operatora arry2D dla tablicy tablic, tablicy list lub listy list.

Przykład

```
let tablica2D = array2D[| [1; 2]; [3; 4] |]
```

ZADANIE 7

Przetestować poniższe kody:

```
//tworzenie tablicy
let tablica = [| 1; 2; 3 |]
printfn "tablica: %A" tablica
let tablica1 = Array.create 5 1
printfn "tablica1: %A" tablica1


//długość tablicy
printfn "długość tablicy: %A" tablica.Length


//zmiana elementu
tablica.[2] <- 7
printfn "zmiana elementu tablicy1: %A" tablica1
Array.set tablica1 2 7
printfn "zmiana elementu tablicy1: %A" tablica1
```

```
//pobranie elemetu
printfn "pobranie elementu o numerze indeksu(2) tablicy: %A"
tablica.[2]
printfn "zmiana elementu o numerze indeksu(2) tablicy1: %A"
(Array.get tablica1 2)


//utworzenie pustej tablicy
let tablica2 = Array.empty
printfn "pusta tablica2: %A" tablica2


//utworzenie tablicy z samych zer
let tablica3: int array = Array.zeroCreate 5
printfn "tablica3 samych zer: %A" tablica3


//kopiowanie tablicy
let tablica4 = Array.copy tablica3
printfn "tablica4, czyli kopia tablicy3: %A" tablica4


//wypełnianie tablicy nowymi zmiennymi
Array.fill tablica3 1 3 5
printfn "tablica3 z nowymi elementami: %A" tablica3


//utworzenie tablicy z użyciem części elementów z poprzedniej tablicy
let tablica5 = Array.sub tablica3 1 2
printfn "tablica5, czyli częśc tablicy3: %A" tablica5


//połączenie tablic
let tablica6 = Array.append tablica3 tablica5
printfn "tablica6, czyli połaczenie tablicy3 i tablicy5: %A" tablica6


// filtrowanie - parzyste
printfn "parzyste o 1 do 20: %A" (Array.filter (fun x -> x % 2 = 0)
[|1..20|])


//tablica odwrócona
printfn "tablica odwrócona: %A" (Array.rev tablica5)


//tablica 2D
let tablica7 = Array2D.init 2 3 (fun i j -> (i+1)*(j+1))
printfn "tablica 2D : %A" tablica7


//zamiana elementu tablicy2D
tablica7.[0,1] <- 8
printfn "tablica 2D po zamaianie: %A" tablica7
```

Moduł Microsoft.FSharp.Collections.Array zawiera metody i właściwości służące do wykonywania zadań na tablicach jednowymiarowych.

Moduł Array2D, Array3D i Array4D zawierają natomiast funkcje, które służą odpowiednio dla tablic o dwóch, trzech i czterech wymiarach.

REKORDY

Rekordy stanowią proste agregaty nazywanych wartości różnych typów.

Przykład

a) definicja typu

```
type Punkt = {x: float; y: float; z:float}
```

b) deklaracja typu

```
let p1 = {x = 1.0; y = 1.0; z = -1.0}
printfn "p1: %A" p1
let p2 = {Punkt.x = 1.0; Punkt.y = 1.0; Punkt.z = -1.0}
printfn "p2: %A" p2
```

Dodatkowo:

```
//aktualizacja rekordu
let p3 = {p2 with y = 1.0; z = -1.0}
printfn "p3: %A" p3

//definicja z polem zmiennym
type Osoba = {
    Nazwisko: string
    Imie: string
    mutable Wiek: int
}
let osoba1 = { Nazwisko = "Kowalski"; Imie = "Jan"; Wiek = 16 }
printfn "osoba1: %A" osoba1

//aktualizacja pola osoba1
osoba1.Wiek <- osoba1.Wiek + 8
printfn "osoba1 po aktualizacji: %A" osoba1
```

RZUTOWANIE I KONWERSJA TYPÓW

W języku F# działa mechanizm wnioskowania typu danych. Nie oznacza to jedna automatycznie z jednego typu do innego. Typy liczbowe nie są niejawnie konwertowane. Konwersje musimy wykonać sami w jawny sposób.

Przykład

```
let a: int = 3
let b: byte = byte a
```

Poniżej w tabeli zostały przedstawione operatory rzutowania typów.

Operator	Opis
byte	Konwersja do typu byte
sbyte	Konwersja do typu byte ze znakiem.
int16	Konwersja do 16-bitowej liczby całkowitej.
uint16	Konwersja do 16-bitowej liczby całkowitej bez znaku.
int32, int	Konwersja do 32-bitowej liczby całkowitej.
uint32	Konwersja do 32-bitowej liczby całkowitej bez znaku.
int64	Konwersja do 64-bitowej liczby całkowitej.
uint64	Konwersja do 64-bitowyej liczby całkowitej bez znaku.
nativeint	Konwersja do natywnego typu całkowitego.
unativeint	Konwersja do natywnego typu całkowitego bez znaku.
float, double	Konwersja do 64-bitowej zmiennoprzecinkowej liczby podwójnej precyzji.
float32, single	Konwersja do 32-bitowej zmiennoprzecinkowej liczby pojedynczej precyzji.
decimal	Konwersja do typu Decimal.
char	Konwersja do typu Char (znak Unicode).
enum	Konwersja do typu wyliczeniowego

Alternatywą jest użycie metod zawartych w bibliotekach .NET.