

Platforma .NET – Wykład 6 Programowanie w F#

Marek Sawerwain

Instytut Sterowania i Systemów Informatycznych
Uniwersytet Zielonogórski

Ostatnia kompilacja pliku: 22 marca 2018, t: 22:25

V1.1 – 1/ 52

Notatki

Spis treści

Wprowadzenie

Plan wykładu

Języki funkcyjne i paradygmaty

Paradygmaty

Programowanie w F#

Struktura programu
Typy dostępne w F#
Wyrażenia z let
Rekurencja, tylko i wyłącznie rekurencja
Konstrukcje iteracyjne
Struktury danych
Przykłady z listami
Sortowanie szybkie

Przykłady

Przeciążanie operatorów
Windows.Forms
Analiza syntaktyczno-leksykalna

Już za tydzień na wykładzie

V1.1 – 2/ 52

Notatki

Plan wykładu – spotkania tydzień po tygodniu

- (1) Informacje o wykładzie, pojęcie platformy, podstawowe informacje o platformie .NET,
- (2) Składowe platformy .NET: CLR, CTS, języki programowania, biblioteki klas, pojęcie podzespołu (ang. assembly),
- (3) Programowanie w C# – środowisko VS, Mono-Develop, syntaktyka C#, wyjątki, współpraca z DLL,
- (4) Programowanie w C# – model obiektowy, typy uogólnione, lambda wyrażenia,
- (5) Programowanie w C# – konwersje pomiędzy typami, operatory „is” oraz „as”, hierarchia wyjątków, aplikacje „okienkowe”, programowanie wielowątkowe,
- (6) ⇒ Programowanie w F# – podstawy, przetwarzanie danych tekstowych, ⇐
- (7) "Klasówka I", czyli wstęp do egzaminu część pierwsza,
- (8) Kolekcje i dostęp do baz danych,

V1.1 – 3 / 52

Notatki

Plan wykładu – tydzień po tygodniu

- (9) Język zapytań LINQ,
- (10) Obsługa standardu XML,
- (11) Technologia ASP.NET,
- (12) ASP.NET Model View Controller,
- (13) Tworzenie usług sieciowych SOA i WCF,
- (14) Bezpieczeństwo aplikacji .NET,
- (15) „Klasówka II”, czyli wstęp do egzaminu część druga.

V1.1 – 4 / 52

Notatki

Plan wykładu

1. ogólne informacje o paradygmatach programowania
 - 1.1 paradygmaty,
 - 1.2 języki funkcyjne,
2. język F#,
 - 2.1 podstawowe typy,
 - 2.2 podstawowe konstrukcje
 - 2.3 rekurencja
 - 2.4 wzorce,
 - 2.5 listy i krotki
3. kilka przykładów,
 - 3.1 przeciążanie operatorów,
 - 3.2 Windows.Forms,
 - 3.3 analiza gramatyczno-leksykalna.

Notatki

Paradygmaty programowania

Paradygmat to charakterystyka stylu programowania, obejmuje koncepcje, idee i abstrakcje w stosowane elementów dostępnych w danym języku programowania jak np.: obiekty, funkcje, zmienne, i etc.. Paradygmat obejmuje także poszczególne kroki obliczeniowe jak przypisanie, kontynuacje i przepływ danych:

- ▶ imperatywny,
- ▶ strukturalny/proceduralny,
- ▶ funkcyjny,
- ▶ zorientowany-obiektowo,
- ▶ deklarytywny/logiczny.

Notatki

Języki funkcyjne

Programowanie funkcyjne, to paradygmat programowania, gdzie obliczenia są traktowane jako realizacja/ewaluacja funkcji (matematycznych) zamiast jawnej zmiany stanu i przekształcenia danych. Funkcje i ich stosowanie stanowi najważniejszy element tego paradygmatu, w przeciwieństwie do stylu imperatywnego gdzie najważniejsza jest zmiana stanu.

Korzenie programowania funkcyjnego wywodzą się z rachunku lambda, systemu formalnego powstałego w roku 1930 roku, którego przeznaczenie jest badanie definicji, zastosowań i rekurencji funkcji. Wiele języków funkcyjnych (jak np.: Lisp, Scheme) wywodzą się bezpośrednio z rachunku lambda.

Przykłady języków funkcyjnych:

- ▶ λ-rachunek, Information Processing Language – IPL,
- ▶ Lisp, Scheme
- ▶ ML (Meta-Language), Miranda, Objective Caml, Erlang, Scala,
- ▶ Haskell, Clean,
- ▶ F# (odmiana Objective Caml), Nemerle.

Notatki

Paradygmaty w języku F#

Język F# jest językiem programowania wspierającym kilka paradygmatów programowania w ramach platformy .NET:

- ▶ F# wspiera programowanie funkcyjne, w tym stylu metody programowania podkreślają co program na zrobić, bez ścisłego określenia jak program powinien pracować,
- ▶ wspierane jest także programowanie obiektowe, dostępne są min. klasy abstrakcyjne, a powstałe klasy naturalnie zgodne są z modelem obiektowym dostępnym w .NET,
- ▶ wspierana są również techniki programowania imperatywnego, co pozwala na łatwą obsługę plików, czy obsługę sieci,
- ▶ język F# jest językiem statycznie typowym, co oznacza iż typ jest znany w trakcie kompilacji,
- ▶ F# jest językiem należącym do platformy .NET, działa w ramach CLI, co pozwala na wykorzystywanie możliwości platformy jak min. „garbage collection”, możliwy jest dostęp do zbioru klas, oraz do takich pojęć jak delegaci, zdarzenia i etc.

Notatki

Funkcja main nie jest konieczna

Pełny program w F#:

```
printfn "Hello, World"
```

Można również utworzyć odpowiednik metody main:

```
open System

[<EntryPoint>]
let main (args : string[]) =

    if args.Length <> 2 then
        failwith "Błąd: nie podano dwóch parametrów"

    let greeting, thing = args.[0], args.[1]
    let timeOfDay = DateTime.Now.ToString("hh:mm tt")

    printfn "%s, %s at %s" greeting thing timeOfDay

    0
```

V1.1 – 9/ 52

Notatki

Funkcja main nie jest konieczna

Jednak program w F# jest niejako wykonywany z góry do dołu, to nie trzeba jawnie wskazywać funkcji głównej za pomocą [<EntryPoint>]:

```
open System

let rec fib = function
    | 0 -> 0
    | 1 -> 1
    | n -> fib (n - 1) + fib (n - 2)

let mainfun() =
    Console.WriteLine("fib 5: {0}", (fib 5))

mainfun()
```

V1.1 – 10/ 52

Notatki

Słowo let i komentarze

Słowo kluczowe **let** realizuje operację wiązania wartości z symbolem

```
let x = 10
    albo
let myAdd = fun x y -> x + y
```

ważny jest fakt iż związanie wartości z symbolem nie może zostać zmienione.

Komentarz jednoliniowy:

```
// słowa w komentarzu
```

Komentarz wieloliniowy:

```
(*
  pierwsza linia
  druga linia
  trzecia linia
*)
```

V1.1 – 11/ 52

Typy w F# i .NET – typy prymitywne

Podstawowe (prymitywne) typy danych dostępny w języku F#

Przykład	Typ F#	Typ .NET	Opis
"Witaj\t""Świecie\n"	string	System.String	ciąg znaków, znak \to symbol escape
@'c:\dir\fs"	string	System.String	ciąg w postaci bezpośredniej
"bytesbytesbytes" B	byte array	System.Byte[]	ciąg znaków zapisany jak tablica bajtów
'c	char	System.Char	znak
true, false	bool	System.Boolean	wartości logiczne
0x22	int/int32	System.Int32	liczba całkowita w postaci szesnastkowej
0o42	int/int32	System.Int32	liczba całkowita w postaci ósemkowej
0b10010	int/int32	System.Int32	liczba całkowita w postaci binarnej
34y	sbyte	System.SByte	bajt ze znakiem
34uy	byte	System.Byte	An unsigned byte
34s	int16	System.Int16	A 16-bit integer
34us	uint16	System.UInt16	An unsigned 16-bit integer
34l	int/int32	System.Int32	A 32-bit integer
34ul	uint32	System.UInt32	An unsigned 32-bit integer
34n	nativeint	System.IntPtr	A native-sized integer
34un	unativeint	System.UIntPtr	An unsigned native-sized integer
34L	int64	System.Int64	A 32-bit integer
34UL	uint64	System.UInt64	An unsigned 32-bit integer
3.0F, 3.0f	float32	System.Single	A 32-bit IEEE floating-point number
3.0	float	System.Double	A 64-bit IEEE floating-point number
3474262622571I	bigint	Microsoft.FSharp.Math.BigInt	An arbitrary large integer
474262612536171N	bignum	Microsoft.FSharp.Math.BigNum	An arbitrary large number

Lambda wyrażenia

Słowo kluczowe **fun** jest stosowane do tworzenia lambda-wyrażeń (funkcji anonimowych). Syntaktyka tego rodzaju funkcji jest następująca:

```
fun parameter-list -> expression
```

Kilka przykładów tworzenia λ -wyrażeń:

```
fun x -> x + 1  
fun a b c -> printfn "%A %A %A" a b c  
fun (a: int) (b: int) (c: int) -> a + b * c  
fun x y -> let swap (a, b) = (b, a) in swap (x, y)
```

Obliczenie kwadratów elementów listy

```
let list = List.map (fun i -> i * i) [1;2;3]  
printfn "%A" list
```

V1.1 – 15/ 52

jak wyżej ale z argumentem *l*:

Polecenie match – dopasowanie wzorca

Wyrażenie **match** pozwala na porównywanie wartości wyrażenia ze zbiorem wzorców i wybór rezultatu jeśli wyrażenie testowe jest zgodne z jednym ze wzorców:

```
// schemat dla match  
match test-expression with  
| pattern1 [ when condition ] -> result-expression1  
| pattern2 [ when condition ] -> result-expression2  
| ...  
  
// schemat dla funkcji  
function  
| pattern1 [ when condition ] -> result-expression1  
| pattern2 [ when condition ] -> result-expression2  
| ...
```

W przypadku połączenie match z tworzeniem lambda wyrażeń (słowo fun), schemat jest następujący:

```
fun arg ->  
match arg with
```

V1.1 – 16/ 52

Notatki

Notatki

Przykłady dla match

Deklaracja listy o pięciu elementach:

```
let list1 = [ 1; 4; 123; 150; 286 ]
```

Wykorzystanie predefiniowanych wielkości opisujących listę:

```
let rec printList listx =  
    match listx with  
    | head :: tail -> printf "%d " head; printList tail  
    | [] -> printfn ""  
printList list1
```

Wiele możliwości w jednej linii

```
let filter123 x =  
    match x with  
    | 1 | 2 | 3 -> printfn "Znaleziono 1, 2 lub 3"  
    | a -> printfn "%d" a
```

V1.1 – 17/ 52

tak samo jak wvżej ale z użyciem function

Instrukcja silnego sterowania

F# oferuje instrukcję sterującą ze silną kontrolą sterowania, a jest to konstrukcja `if ... then ... else`

```
let result =  
    if System.DateTime.Now.Second % 2 = 0 then  
        "tik"  
    else  
        "tak"  
printfn "%A" result
```

Jest to bezpośredni odpowiednik następującego przykładu z match:

```
let result =  
    match System.DateTime.Now.Second % 2 = 0 with  
    | true -> "tik"  
    | false -> "tak"
```

Notatki

Notatki

Aktywne wzorce

Aktywne wzorce pozwalają na podanie konstrukcji (funkcji) która będzie testować poprawność argumentu i po spełnieniu założeń wskazywać na rodzaj argumentu. Schemat wzorca aktywnego oraz częściowo aktywnego:

```
let (| identfier1 | identfier2 | ... | ) [ arguments ] = expression  
let (| identfier1 | identfier2 | ... | _ | ) [ arguments ] = expression
```

Przykład sprawdzający parzystość argumentu:

```
let (|Even|Odd|) input = if input % 2 = 0 then Even else Odd
```

Aktywny wzorec do sprawdzania parzystości argumentu:

```
let TestNum input =  
    match input with  
    | Even -> printfn "liczba %d parzysta" input  
    | Odd -> printfn "liczba %d nieparzysta" input
```

V1.1 – 19/ 52

Konstrukcje rekurencyjne

Definicja funkcji rekurencyjnej **fib** obliczającej liczby ciągu Fibbonaciego:

```
let rec fib x =  
    match x with  
    | 1 -> 1  
    | 2 -> 1  
    | x -> fib (x - 1) + fib (x - 2)
```

```
printfn "(fib 2) = %i" (fib 2)  
printfn "(fib 6) = %i" (fib 6)  
printfn "(fib 11) = %i" (fib 11)
```

i elementy ciągu Lucasa:

```
let rec luc x =  
    match x with  
    | x when x <= 0 -> failwith "wartość musi być większa niż zero"  
    | 1 -> 1  
    | 2 -> 3  
    | x -> luc (x - 1) + luc (--x - 2)
```

```
printfn "(luc 2) = %i" (luc 2)  
printfn "(luc 6) = %i" (luc 6)  
printfn "(luc 11) = %i" (luc 11)
```

V1.1 – 20/ 52

Notatki

Notatki

Pętla for ... do

Ogólny zapis instrukcji for

```
for identyfier = start [ to | downto ] finish do  
    body-expression
```

Przykład typowej pętli for:

```
let funexam1() =  
    for i = 1 to 100 do  
        printf "%d " i  
    printfn ""
```

Określenie wartości początkowej i końcowej:

```
let ex1 x y = x - 2*y  
let ex2 x y = x + 2*y  
  
let funexam2 x y =  
    for i = (ex1 x y) to (ex2 x y) do  
        printf "%d " i  
    printfn ""
```

V1.1 - 21 / 52

Pętla for ... in

Ogólny zapis instrukcji for ... in dla przeliczalnego wyrażenia

```
for pattern in enumerable-expression do  
    body-expression
```

Przeglądnięcie listy:

```
let list1 = [ 1; 2; 4; 8; 16 ]  
for i in list1 do  
    printfn "%d" i
```

Utworzenie i przeglądnięcie sekwencji uporządkowanych par:

```
let seq1 = seq { for i in 1 .. 10 -> (i, i*i) }  
for (i, isqr) in seq1 do  
    printfn "%d podniesione do kwadratu to %d" i isqr
```

V1.1 - 22 / 52

Notatki

Notatki

Pętla for ... in

Wyświetlenie, co drugiego elementu:

```
let funexam3() =  
  for i in 1 .. 2 .. 100 do  
    printf "%d " i  
  printfn ""  
funexam3()
```

Konwersja liczby całkowitej na postać binarną:

```
let BitsNum = 32 ;;  
let binary_of_int n =  
  [ for i in BitsNum - 1 .. -1 .. 0 ->  
    if (n >>> i) % 2 = 0 then "0" else "1" ]  
  |> String.concat "" ;;  
let x1 = binary_of_int 1431 ;;  
let x2 = binary_of_int ( -1123 ) ;;
```

Operator |> to tzw. pipe-forward operator stosujący lewy argument to prawego np.:

0.5 |> Math.Sin. Operator " _ " pozwala na zignorowanie zmiennej/wzorca w konstrukcji for: [V1.1 - 23/ 52](#)

Notatki

Konstrukcja pętli while

Konstrukcja pętli while ma następującą postać:

```
while test-expression do  
  body-expression
```

Typowy iteracyjny przykład zastosowania pętli while:

```
open System  
  
let szukajWartości val maxVal =  
  let mutable nextIter = true  
  let randomNumberGenerator = new Random()  
  while nextIter do  
    let rand = randomNumberGenerator.Next( maxVal )  
    printfn "%d " rand  
    if rand = val then  
      printfn "\nOdgadnięto wartość %d!" value  
      nextIter <- false
```

szukajWartości 10 30

Notatki

Listy

Lista to podstawowy typ (odpowiada tablicy w przypadku języków imperatywnych) w przypadku F# i jest on typem wbudowany w języku F#. Wszystkie elementy listy muszą być tego samego typu. Podstawowe konstrukcje – "[]" – lista pusta, operator ":@" łączy dwa elementy w listę:

```
let emptyList = []  
let lista1 = "jeden" :: []  
let lista2 = "jeden" :: "dwa" :: []  
let lista3 = ["jeden"; "dwa"]
```

Operator "@" łączy dwie listy w jedną listę:

```
let lista4 = ["jeden"; "dwa"] @ ["trzy "; "cztery"; "pięć"]
```

Wykorzystanie „pudełkowania” do utworzenia listy z elementami różnych typów:

V1.1 – 25 / 52

```
let lista5 = [box 1; box 2.0; box "trzy"]
```

Proste przykłady dla list

Odwrócenie kolejności elementów na liście:

```
let revlista = List.rev lista
```

Łącznie list w jedną listę:

```
let listaList = [[2; 3; 5]; [7; 11; 13]; [17; 19; 23; 29]]  
let rec łącznieList l =  
    match l with  
    | głowa :: ogon -> głowa @ (łącznieList ogon)  
    | [] -> []  
let liczbyPierwsze = łącznieList listaList  
printfn "%A" liczbyPierwsze
```

Zastosowanie operacji do każdego elementu na liście:

```
let rec listmap func list =  
    match list with  
    | head :: rest ->  
        func head :: listmap func rest
```

V1.1 – 26 / 52

Notatki

Notatki

Suma elementów listy

Zadanie polega na obliczeniu wartości ciągu np.: od 1 do 100. Podejście iteracyjne:

```
let forSum n =  
    let mutable r = 0  
    for i = 1 to n do  
        r <- r + i  
    r
```

Wykorzystanie sekwencji:

```
let sequenceSum data = data |> Seq.fold (fun acc a -> acc + a) 0 ;;
```

Podejście bezpośrednie (naiwne, ale bardziej funkcyjne):

```
let rec simpleListSum L =  
    match L with  
    | h::t -> h + simpleListSum t  
    | [] -> 0 ;;
```

V1.1 - 27 / 52

Notatki

Suma elementów listy

Obliczanie sumy elementów listy z akumulatorem:

```
let rec ListSumAcc L accu =  
    match L with  
    | h::t -> ListSumAcc t (h + accu)  
    | [] -> accu + 0 ;;
```

```
let ListSum L = ListSumAcc L 0 ;;
```

Można zastosować **reduce** do realizacji tego zadania:

```
let ReduceSum L = List.reduce (fun x y -> x + y) L ;;
```

Wyświetlenie wyników poszczególnych technik obliczania sumy elementów listy:

```
printfn "The imperativeSum from 1 to 100 = %A" (forSum 100)  
printfn "The sequenceSum from 1 to 100 = %A" (sequenceSum [1 .. 100])  
printfn "The naiveListSum from 1 to 100 = %A" (simpleListSum [1 .. 100])
```

V1.1 - 28 / 52

Notatki

Sortowanie szybkie

Sortowanie szybkie, to modelowy przykład stosowany do prezentacji złożośći programowania funkcyjnego:

```
let rec qsort1 L =  
    match L with  
    | [] -> []  
    | x::xs ->  
        let smaller = [for i in xs when i <= x -> i] in  
        let larger = [for i in xs when i > x -> i] in  
        qsort1 smaller @ [x] @ qsort1 larger;;
```

Ta wersja będzie funkcjonować poprawie tylko w (naj)starszych wersjach języka F#.

V1.1 – 29 / 52

Notatki

Sortowanie szybkie

Wersja sortowania szybkiego dla nowych wydań F#.

```
let rec qsort1 L =  
    match L with  
    | [] -> []  
    | x::xs ->  
        let smaller = [for i in xs do if i <= x then yield i] in  
        let larger = [for i in xs do if i > x then yield i] in  
        qsort1 smaller @ [x] @ qsort1 larger;;
```

Można to zapisać krócej bez jawnego powoływania list **smaller** oraz **larger**:

```
let rec qsort2 = function  
    [] -> []  
    | x::xs ->  
        qsort2 [for a in xs do if a < x then yield a] @ x ::  
        qsort2 [for a in xs do if a >= x then yield a]
```

V1.1 – 30 / 52

Notatki

Sortowanie szybkie

Wersja sortowania szybkiego z operacją filtrowania listy:

```
let rec quicksort l =  
  match l with  
  | [] -> []  
  | h::t -> quicksort (List.filter (fun x -> x < h) t) @ h ::  
    quicksort (List.filter (fun x -> x >= h) t)
```

V1.1 – 31/ 52

Notatki

Krotki

Krotki (ang. tuples), to konstrukcja pozwalająca na tworzenie uporządkowanych wartości o różnych typach:

```
( element1 , element2, ... , elementN )
```

Przykłady krotek:

```
( 1, 2 )  
( "one", "two", "three" )  
( a, b )  
( "one", "1", "2.0" )  
( a + 1, b + 1 )  
let (a, b) = (1, 2)  
let c = fst (1, 2)  
let d = snd (1, 2)  
let third (_, _, c) = c
```

Odczytanie pierwszego i drugiego elementu:

V1.1 – 32/ 52

Notatki

Zbiory

Typ zbiorowy posiada następujące własności:

- ▶ są niezienne (ang. immutable),
- ▶ są zawsze posortowane,
- ▶ nie mają powielonych elementów,
- ▶ szybkie operacje wstawiania, usuwania, badania przynależności, określania podzbiatów, a także podstawowe operacje jak suma, różnica, iloczyn (część wspólna),
- ▶ nie ma dostępu losowego .

Biblioteka standardowa F# oferuje typ zbiorowy który jest implementowany jako zbalansowane drzewo binarne. Pozwala to aby dowolny pojedynczy element był dodawany lub usuwany ze zbioru o n elementach w czasie $O(\log_2 n)$.

V1.1 – 33/ 52

Notatki

Zbiory – przykłady operacji

Tworzenie zbiorów:

```
let s1 = Set.empty  
let s2 = Set.singleton 3  
let s3 = set [1;2;3;4;5]
```

Dołączanie elementu, konwersja na listę:

```
let s4 = Set.add 5 s2  
let s5 = s2, s3  
let l1 = Set.toList s4
```

Podstawowe operacje na zbiorach (suma, część wspólna, różnica, podzbiór):

```
printfn "%A" (Set.union (set [1; 3; 5]) (set [3; 5; 7]))  
printfn "%A" (Set.intersect (set [1; 3; 5]) (set [3; 5; 7]))  
printfn "%A" (Set.difference (set [1; 3; 5]) (set [3; 5; 7]))  
printfn "%A" (Set.isSubset (set [3; 5]) (set [3; 5; 7]))
```

V1.1 – 34/ 52

Notatki

Tablice

Tablice o struktura o ustalonej wielkości, o elementach takiego samego typu, sposoby tworzenia są następujące:

```
let a1 = [| 1; 2; 3 |]  
let a2 = [| for i in 1 .. 10 -> i * i |]  
let a3 : int array = Array.zeroCreate 10
```

Dostęp do elementów jest możliwy dzięki operatorowi `.[]`:

```
a2.[0..2]  
a2[..2]  
a2.[2..]
```

Przetwarzanie elementów tablicy za pomocą operatora `"|>"`:

```
[| 1 .. 100 |]  
|> Array.filter (fun elem -> elem % 3 = 0) V1.1 - 35/ 52  
|> Array.choose (fun elem -> if (elem <> 7) then Some(elem*elem) else None)  
|> Array.rev  
|> printfn "%A"
```

Definicja typów

Do tworzenia typów (a dokładniej klasa, ale można go też traktować jako rekord bądź strukturę, jednakże F# posiada typ strukturalny) stosowane jest słowo kluczowe **type** o następującej syntaktyce:

```
[ attributes ]  
type [accessibility-modifier] typename = {  
  [ mutable ] label1 : type1;  
  [ mutable ] label2 : type2;  
  ...  
}  
member-list
```

Przykładowe deklaracje

```
type Point3D = { x : float; y : float; z : float; }  
type Klient = { Imię : string; Nazwisko : string; ID : uint32; NumerKonta : uint32; }
```

Rozpoznanie typu odbywa się po etykietach pól:

Notatki

Notatki

Definicja typów

Wyrażenie match oraz przykładowy typ:

```
let zróbCośZPunktem (point: Point3D) =  
    match point with  
    | { x = 0.0; y = 0.0; z = 0.0 } -> printfn "centrum"  
    | { x = xVal; y = 0.0; z = 0.0 } -> printfn "Punkt leży na osi X. Wartość %f." xVal  
    | { x = 0.0; y = yVal; z = 0.0 } -> printfn "Punkt leży na osi Y. Wartość %f." yVal  
    | { x = 0.0; y = 0.0; z = zVal } -> printfn "Punkt leży na osi Z. Wartość %f." zVal  
    | { x = xVal; y = yVal; z = zVal } -> printfn "Punkt (%f, %f, %f)." xVal yVal zVal
```

Odszukanie imienia osoby o określonym nazwisku:

```
type Osoba = { imię : string ; nazwisko : string }  
let listOsób =  
    [ { imię = "Imie1" ; nazwisko = "Nazwisko1" } ;  
      { imię = "Imie2" ; nazwisko = "Nazwisko2" } ;  
      { imię = "Imie3" ; nazwisko = "Nazwisko3" } ;  
      { imię = "Imie4" ; nazwisko = "Nazwisko4" } ]  
  
let rec odszukajImię l n =  
    match l with  
    | { imię = x ; nazwisko = nazwisko } :: ogon when nazwisko = n -> x  
    | { imię = _ ; nazwisko = _ } :: ogon -> (odszukajImię ogon n)  
    | [] -> failwith "Nie znaleziono osób"  
  
printfn "%A" (odszukajImię listOsób "Nazwisko2")
```

V1.1 – 37 / 52

Notatki

Unie dyskryminowane

W definicji typów można również stosować notację wzorca:

```
type BinaryTree =  
    | Node of int * BinaryTree * BinaryTree  
    | Empty
```

Funkcja wypisujące elementy w porządku „in-order”:

```
let rec printInOrder tree =  
    match tree with  
    | Node (data, left, right)  
      -> printInOrder left  
          printfn "Wartość %d" data  
          printInOrder right  
    | Empty  
      -> ()
```

Definicja drzewa i wyświetlenie elementów:

```
let binTree =
```

V1.1 – 38 / 52

Notatki

Leniwe obliczenia

Leniwe obliczenia, to typ obliczeń w których określone wyrażenia są obliczane na żądanie, bardzo często technika ta jest łączona z sekwencją generującą nieskończoną listę danych:

```
let fibs =  
    Seq.unfold  
        (fun (n0, n1) ->  
            Some(n0, (n1, n0 + n1)))  
        (1I, 1I)  
  
let f20 = Seq.take 20 fibs  
  
printfn "%A" f20  
for i in f20 do printfn "%A" i
```

Notatki

Klasy w F#

Słowo kluczowe type jest wykorzystywane także do tworzenia klas, gdzie obiekty są tworzone za pomocą new. Wszystkie możliwości klas dostępne w ramach platformy .NET zostały przeniesione do F#:

```
type Base() =  
    member x.GetFirstState() = 10  
  
type Sub() =  
    inherit Base()  
    member x.GetSecondState() = 20  
  
let myObject = new Sub()  
  
printfn  
    "myObject.state = %i, \nmyObject.otherState = %i"  
    (myObject.GetFirstState())  
    (myObject.GetSecondState())
```

Notatki

Przekładanie operatorów

Język F# posiada bardzo szerokie możliwości przekładania operatorów:

```
type Vector(x: float, y : float) =  
    member this.x = x  
    member this.y = y  
    static member (~-) (v : Vector) =  
        Vector(-1.0 * v.x, -1.0 * v.y)  
    static member (*) (v : Vector, a) =  
        Vector(a * v.x, a * v.y)  
    static member (*) (a, v: Vector) =  
        Vector(a * v.x, a * v.y)  
    override this.ToString() =  
        this.x.ToString() + " " + this.y.ToString()  
  
let v1 = Vector(1.0, 2.0)  
let v2 = v1 * 2.0  
let v3 = 2.0 * v1  
let v4 = - v2  
printfn "%s" (v1.ToString())
```

Wyrażenie (~-) oznacza iż operator - jest operatorem unarnym.

V1.1 - 41/ 52

Notatki

Przekładanie operatorów globalnych

Można też przekładać operatory globalne, a nawet tworzyć specjalne własne operatory:

```
let inline (+?) (x: int) (y: int) = x + 2*y  
printfn "%d" (10 +? 1)
```

Inny przykład bez specjalizacji typów:

```
let inline (+@) x y = x + x * y  
printfn "%d" (1 +@ 1)  
printfn "%f" (1.0 +@ 0.5)
```

V1.1 - 42/ 52

Notatki

Aplikacja w Windows.Forms

Aplikacja w Windows.Form z przyciskiem po którym wyświetlone zostanie okno typu MessageBox:

```
open System.Windows.Forms

let form = new Form(Text = "Okno typu -- WinForms")

let button = new Button(Text="Kliknij Mnie!", Dock=DockStyle.Fill)

button.Click.Add( fun _ -> MessageBox.Show("Hello, World!", "Cześć!") |> ignore )

form.Controls.Add(button)
form.Show()

Application.Run(form)
```

V1.1 – 43/ 52

Notatki

Drzewa i kontrolka TreeView

Typ dla drzewa przechowującego dowolne typy oznaczone przez 'a:

```
type 'a Tree =
  | Node of 'a Tree * 'a Tree
  | Leaf of 'a

let tree =
  Node(
    Node(
      Leaf "one",
      Node(Leaf "two", Leaf "three")),
    Node(
      Node(Leaf "four", Leaf "five"),
      Leaf "six"))
```

V1.1 – 44/ 52

Notatki

Drzewa i kontrolka TreeView

Przepisanie zawartości drzewa o typie 'a Tree do kontrolki TreeView:

```
let mapTreeToTreeNode t =  
  let rec mapTreeToTreeNodeInner t (node : TreeNode) =  
    match t with  
    | Node (l, r) ->  
      let newNode = new TreeNode("Node")  
      node.Nodes.Add(newNode) |> ignore  
      mapTreeToTreeNodeInner l newNode  
      mapTreeToTreeNodeInner r newNode  
    | Leaf x ->  
      node.Nodes.Add(new TreeNode(sprintf "%A" x)) |> ignore  
  let root = new TreeNode("Root")  
  mapTreeToTreeNodeInner t root  
  root
```

V1.1 – 45 / 52

Notatki

Drzewa i kontrolka TreeView

Utworzenie formularza i uruchomienie aplikacji:

```
let form =  
  let temp = new Form()  
  let treeView = new TreeView(Dock = DockStyle.Fill)  
  treeView.Nodes.Add(mapTreeToTreeNode tree) |> ignore  
  treeView.ExpandAll()  
  temp.Controls.Add(treeView)  
  temp  
  
Application.Run(form)
```

V1.1 – 46 / 52

Notatki

Liczby Fibonacciego i przetwarzanie w tle

Włączenie niezbędnych podzespółów oraz utworzenie typu i „leniwej” listy z liczbami Fibonacciego:

```
open System
open System.ComponentModel
open System.Windows.Forms
open System.Numerics

type Result =
    { Input: int;
      Fibonacci: BigInteger; }

let fibs =
    (1I,1I) |> Seq.unfold
        (fun (n0, n1) ->
            Some(n0, (n1, n0 + n1)))

let fib n = Seq.nth n fibs
```

V1.1 – 47 / 52

Notatki

Liczby Fibonacciego i przetwarzanie w tle

Utworzenie formularza oraz obiektu przeprowadzającego obliczenia w tle:

```
let form =
    let form = new Form()
    let input = new TextBox()
    let button = new Button(Left = input.Right + 10, Text = "Go")
    let results = new BindingList<Result>()
    let output = new DataGridView(Top = input.Bottom + 10,
        Width = form.Width, Height = form.Height - input.Bottom + 10,
        Anchor = (AnchorStyles.Top ||| AnchorStyles.Left ||| AnchorStyles.Right ||| AnchorStyles.Bottom),
        DataSource = results)

    let runWorker() =
        let background = new BackgroundWorker()
        let input = Int32.Parse(input.Text)
        background.DoWork.Add(fun ea ->
            ea.Result <- (input, fib input))
        background.RunWorkerCompleted.Add(fun ea ->
            let input, result = ea.Result :?> (int * BigInteger)
            results.Add({ Input = input; Fibonacci = result; })
            background.RunWorkerAsync())

    button.Click.Add(fun _ -> runWorker())
    let dc c = c :> Control
    form.Controls.AddRange([|dc input; dc button; dc output |])
    form

do Application.Run(form)
```

V1.1 – 48 / 52

Operator :?> jest operatorem konwersji/rzutu, natomiast <- jest operatorem przypisania.

Notatki

Analiza wyrażeń arytmetycznych – biblioteka FParsec

Język F#, a ogólniej rodzina języków ML/Haskell/Ocaml dzięki swoim własnościom jest predysponowana do budowy analizatorów syntaktyczno-leksykalnych. Jedną z bibliotek, która dodatkowo ułatwia to zadanie jest biblioteka o nazwie "FParsec".
Utworzenie obiektów pomocniczych:

```
open System
open FParsec.Primitives
open FParsec.CharParsers
open FParsec.OperatorPrecedenceParser

let ws = spaces // skips any whitespace

let ch c = skipChar c >>. ws

let number = pfloat .>> ws

let opp = new OperatorPrecedenceParser<_,_>()
let expr = opp.ExpressionParser
opp.TermParser <- number <|> between (ch '(') (ch ')')
```

V1.1 – 49/ 52

Notatki

Analiza wyrażeń arytmetycznych – biblioteka FParsec

Określenie operatorów oraz pomocniczych funkcji jak exp, log, sin:

```
opp.AddOperator(InfixOp("+", ws, 1, Assoc.Left, fun x y -> x + y))
opp.AddOperator(InfixOp("-", ws, 1, Assoc.Left, fun x y -> x - y))

opp.AddOperator(InfixOp("*", ws, 2, Assoc.Left, fun x y -> x * y))
opp.AddOperator(InfixOp("/", ws, 2, Assoc.Left, fun x y -> x / y))

opp.AddOperator(InfixOp("^", ws, 3, Assoc.Right, fun x y -> System.Math.Pow(x, y)))

opp.AddOperator(PrefixOp("-", ws, 4, true, fun x -> -x))

let ws1 = notFollowedBy letter >>. ws

opp.AddOperator(PrefixOp("log", ws1, 4, true, fun x -> System.Math.Log(x)))
opp.AddOperator(PrefixOp("exp", ws1, 4, true, fun x -> System.Math.Exp(x)))
opp.AddOperator(PrefixOp("sin", ws1, 4, true, fun x -> System.Math.Sin(x)))

let completeExpression = ws >>. expr .>> eof

let calculate s = run completeExpression s
```

V1.1 – 50/ 52

Notatki

Analiza wyrażeń arytmetycznych – biblioteka FParsec

Analiza wyrażenia i odczytanie wyniku:

```
let printResult r =  
    match r with  
    | Success (v, a, b) -> printfn "wynik to %A" v  
    | Failure (msg, err, _) -> printf "%s" msg; failwith msg  
  
(calculate "10.5 + 123.25 + 877") |> printResult
```

Inne narzędzia do analizy syntaktyczno-leksykalnej, to aplikacje fslex i fsyacc dostępne w pakiecie „FSharpPowerPack”.

W następnym tygodniu między innymi

1. kolekcje, uogólnione kolekcje,
2. współpraca z plikami,
3. dostęp do baz danych (wspierane serwery danych),
4. architektura ADO.NET,
5. kontrolki Windows.Forms i WPF, dostępu do danych.

Proponowane tematy prac pisemnych:

1. analiza paradygmatów programowania,
2. pojęcie funkcji wyższego rzędu w F#,
3. prosty język interpretowany oparty o podzbiór języka Pascal bądź C opracowany za pomocą F#.

Dziękuję za uwagę!!!

Notatki

Notatki
