SKRIPTA

OSNOVNI KONCEPTI PROGRAMSKOG JEZIKA F#

Jaša

Sadržaj

1. Uvod i pregled

F# je jezik koji podržava više programskih paradigmi (multi-paradigm programming language).[1] Prvenstveno je osmišljen kao funkcionalni jezik, ali podržava imperativnu i objektno orijentisanu paradigmu.

F# je strogo tipiziran jezik (strongly typed) gde se vrednost (value) vezuje sa tipom (bind) pre izvršavanja programa, što je karakteristika većine funkcionalnih jezika. Šta više, tendencija F#-a je da ne dozvoli programeru da vrednosti posmatra kao promenljive u nekim imperativnim i objektno-orijentsanim jezicima.

```
let broj = 5

val broj : int = 5
```

Vrednost se ne posmatra kao „slot“ ili kućica u koju se upisuje neka broj koja se menja pri izvršavanju programa.

```
broj = 6

val it : bool = false
```

```
broj = 5

val it : bool = true
```

```
broj <- 6

error FS0027: This value is not mutable
```

Ukoliko postoji potreba da vrednosti u F# jeziku (value) promeni „sadržaj“ posle deklaracije, takva namera se mora jasno naznačiti rezervisanom rečju mutable pri samom deklarisanju vrednosti! Tada vrednost počinje da liči na promenljive iz drugih programskih jezika.

```
let mutable promenljiviBroj = 7

val mutable promenljiviBroj : int = 7
```

Dodeljivanje nove vrednosti vrednosnoj promenljivoj vrši se posebnim operatorom (<-).

```
let mutable promenljiviBroj = 7
promenljiviBroj <- 10
promenljiviBroj + 1

val mutable promenljiviBroj : int = 10
val it : int = 11
```

Odlika ovog jezika je i „izvođenje tipova“ (*type inference*). To znači da je kompajler sposoban da u mnogim situacijama proceni kog je tipa vrednost, bez eksplicitnog deklarisanja tipa od strane programera.[2] Ova činjenica može navesti na pogrešan zaključak nekoga ko prvi put posmatra kod napisan ovim jezikom da se radi dinamičkoj tipizaciji.

```
let i = 1
let f = 1.0
let tekst = "Neki string"
let istina = true


val i : int = 1
val f : float = 1.0
val tekst : string = "Neki string"
val istina : bool = true
```

Osnovna jedinica izvršavanja programa je funkcija. Vezivanje funkcije za ime izražva se kao i kod prostih vrenosti (Simple values) ključnom rečju „let“. Sledeća funkcija prima broj, uvećava ga za 5 i vraća novi uvećan broj. Funkcije u F#-u uvek imaju povratnu vrednost.

```
let fja x = x + 5
fja 1

val fja : x:int -> int
val it : int = 6
```

Iz potpisa ove funkcije vidimo da ona prima jedan parametar x tipa int (x:int), kao i da vraća jedan int (-> int). Ista funkcija se može vezati za novo ime:

```
let dodaj5 = fja
dodaj5 1

val dodaj5 : (int -> int)
val it : int = 6
```

F# je jzik u kome se strogo pazi na poziciju i uvlačnje (indentation) koda. Od važnosti je da se skrene pažnja i na doseg promenljive u okviru finkcije. Sasvim očekivano, promenljiva deklarisana na višem nivou vidljiva je nižim nivoima:

```
let topLevel =
    let lowLevel1_1 = 1
    let lowLevel1_2 = 1
    let lowLevel1_3 =
        let lowerLevel2_1 = 21
        2 + 2 + lowLevel1_1 + lowerLevel2_1
    //ne bih mogao da napisem lowerLevel2_1 + 1 na kraju jer nije vidljiv visem nivou
    lowLevel1_3 + 1
//val topLevel : int = 27
```

Osobenost programa se ogleda u mogučnosti da se „pregazi“ postojeća promenljiva:

```
let nekaFunkcija x =
    let x = 6
    let y =
        let x = 7
        x + 1
    y

nekaFunkcija 2
```

Ukoliko bi sličan kod bio napisan u sklopu neke metoda nekog drugog jezika, na primer Java, kompajler bi javio grešku,u konkretnom slučaju: Duplicate local variable x.

```
	public static int vratiMe(int x){
		int x = 6;
	}
```

Rezultat izvršenja prethodne F# funkcije broj 8 bez obzira na uneti parametar. Kompajler je parametru x dodelio generički tip ('a)!

```
val nekaFunkcija : x:'a -> int
val it : int = 8
```

Kao u matematici, parametar jedne funkcije može biti druga funkcija (higher order function). Prvo je definisana funkcija (obradi4IDodaj2) koja kao argument prima drugu funkciju (funct), prosleđuje joj vrednost 4, a potom na rezultat dodaje 2. Ovo se jasno vidi i iz potpisa (`funct:(int -> int) -> int`) – parametar je funkcija koja preslikava vrednosti iz skupa int u isti taj skup, funkcija na kraju takođe vraća int vrednost. Druga funkcija (pomnoziTrojkom) primljeni parametar množi trojkom. Rezultat je 14, kao što je i predviđeno.

```
let obradi4IDodaj2 funct =
    funct 4 + 2

let pomnoziTrojkom x = x*3

obradi4IDodaj2 pomn0ziTrojkom

val obradi4IDodaj2 : funct:(int -> int) -> int
val pomnoziTrojkom : x:int -> int
val it : int = 14
```

F# omogućava zgodan način za prosleđivanje rezultata jedne funkcije drugoj, i za to postoje posebni operatori.

```
pomnziTrojkom
|> obradi4IDodaj2

val it : int = 14
```

Ovo je posebno značajno kada je potrebno „ulančati“ više funkcija. Pored imenovanih funkcija, u jeziku se javljaju anonimne, ili lamda funkcije. One se označavaju posebnom rečju „fun“, potom se navode parametri, strelica (->) i pravilo preslikavanja.

```
obradi4IDodaj2 (fun x -> x * 3)

val it : int = 14
```

U ovom primeru, već poznatoj funkciji prosleđena je lambda funkcija koja obavlja istu stvar kao i pomnoziTrojkom. Ukoliko postoji potreba za formiranjem neparametarske funkcije, to se radi na sledeći način:

```
let ispis () = printf "Neki ispis"

val ispis : unit -> unit
```

Poziv ove metode vrši se navođenjem zagrada takođe (ispis ()). Potpis ove metode je (`unit -> unit`). Svaka funkcija u jeziku mora imati svoj domen i kodomen, a unit tip zapravo ukazuje na odsustvo bilo kakve specifične vrednosti. Ovaj tip podseća na tip void u drugim programskim jezicima.

2. Kratak pregled tipova

Tipove podataka u F# jeziku možemo podeliti na one koji su karakteristični za .NET platformu, i one koji su specifični, ili su drugačije implementirani. Na primer:

```
let boolTip = true
let charTip = 'a'
let stringTip = "neki string"
let nekiInt = 45
let nekiInt64 = 45L
let nekiFloat = 4.0
let nekiFloat32 = 4.0f

val boolTip : bool = true
val charTip : char = 'a'
val stringTip : string = "neki string"
val nekiInt : int = 45
val nekiInt64 : int64 = 45L
val nekiFloat : float = 4.2
val nekiFloat32 : float32 = 4.0f
```

Kastovanje primitivnih tipova se vrši se na sličan način kao u C# jeziku.

```
let nekiFloatUInt = (int) nekiFloat
val nekiFloatUInt: int = 4

let charTipUFloat = (float) charTip
val charTipUFloat : float = 97.0
```

Očkivano, mogu se koristiti metode klase Convert:

```
let boolUInt = System.Convert.ToInt32(boolTip)
let nekiFloatUInt = System.Convert.ToInt32(nekiFloat)
```

String se prebacije u broj metodom Parse:

```
let decimalniBroj = System.Decimal.Parse("33.44")

val decimalniBroj : decimal = 33.44M
```

Liste su uređena nepromenljiva serija elemenata istog tipa. Liste u jeziku F# se razlikuju od lista u drugim jezicima .NET „frejmvorka“.

```
let nekaLista = [1;2;3;4]

val nekaLista : int list = [1; 2; 3; 4]
```

U prvom slučaju, sa dve tačke je označena želja programera da kreira listu brojeva od 3 do 5. U drugom, na tako kreiranu listu dodaje se novi element 2, a u trećem element 1. U četvrtom slučaju spajaju se dve liste [0;1] i listaDbaDoPet. U poslednjem primeru kreiramo listu koja sadrži cele brojeve od 0 do 10.

```
let listaTriDoPet = [3..5]
let listaDvaDoPet = 2 :: listaTriDoPet
let listaJedanDoPet = 1 :: listaDvaDoPet
let listaNulaDoPet = [0;1] @ listaDvaDoPet
let listaNulaDoDeset = listaNulaDoPet @ [6 ..10]

val listaTriDoPet : int list = [3; 4; 5]
val listaDvaDoPet : int list = [2; 3; 4; 5]
val listaJedanDoPet : int list = [1; 2; 3; 4; 5]
val listaNulaDoPet : int list = [0; 1; 2; 3; 4; 5]
val listaNulaDoDeset : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

List modul sadrži pregršt korisnih funkcija koje omogućavaju lakši rad sa listama. Pretpostavimo da je potrebno da se odredi suma kvadrata svih brojeva od 1 do 10.

```
[1 .. 10] |>
List.map (fun x -> x * x)
|> List.sum

val it : int = 385
```

U prvoj liniji kreirana je lista brojeva, koja se prosleđuje funkciji List.map koja prima dva parametra – funkciju koju će primeniti na svaki element liste (u ovom slučaju to je lambda funkcija `fun x -> x * x` koja svaki element množi samim sobom), drugi parametar je sama lista. Rezltat ove funkcije je nova lista elemenata koji su kvadrirani, potom se ona prosleđuje funkciji sum, koja vraća zbir svih elemenata liste.

Ukoliko pak treba izbaciti neke element iz liste, to je moguće uraditi ovako:

```
let vratiParne lista = List.filter (fun x -> x % 2 = 0) lista
vratiParne listaNulaDoDeset

val vratiParne : lista:int list -> int list
val it : int list = [0; 2; 4; 6; 8; 10]
```

Funkcija List.filter vraća novu kolekciju elemenata koji zadovoljavaju dati predikat[3], u ovom slučaju, to su brojevi deljivi dvojkom (`fun x -> x % 2 = 0`) – uzima int i vraća true ili false u zavisnosti od toga da li je uslov zadovoljen (x: int -> bool).

Tokom pisanja rada iskusio sam potrebu da pokušam da napišem svoje vrezije za neke od ovih funkcija. Kreiranje takvih pandana zahteva korišćenje rekurzije. Rekurzivne funkcije u F#-u moraju biti označene rečju „rec“. Takođe je bilo neophodno upoznati se sa idejom „match..with“ izraza koji omogućavaju „grananje“ i podsećaju na „switch..case“ izraze u drugim programskim jezicima.

```
let rec listMap list funct =
    match list with
    | [] -> []
    | glava :: rep ->
        (funct glava) :: (listMap rep funct)

val listMap : list:'a list -> funct:('a -> 'b) -> 'b list

listMap [0..10] (fun x -> x * 2)

val it : int list = [0; 2; 4; 6; 8; 10; 12; 14; 16; 18; 20
```

Dakle, prvi parametar je lista, drugi funkcija. U match izrazu isprva proveravamo da li lista sadrži elemente, i ukoliko je prazna, vraća se prazna lista. U drugom match slučaju (kada nije prazna) razdvajamo prvi element od ostatka liste, potom primenimo funkciju na taj element i spajamo sa ostatkom liste `(funct glava) ::`. Rakurzivno se prosleđuje ostatak liste `(listMap rep funct)`, gde se ponovo „odseca“ glava. Ceo postupak se ponavlja dok lista koja se prosleđuje ne ostane prazna. Potom sledi spajanje svih elemenata u novu listu.

Slična ideja je bila pri kreiranju pandana funkcije sum:

```
let rec sumaElemenata lista broj =
    match lista with
    | [] -> broj
    | glava::rep ->
        sumaElemenata rep (glava + broj)

sumaElemenata [1..3] 0

val sumaElemenata : lista:int list -> broj:int -> int
val it : int = 6
```

Funkcija uzima dva parametra, listu i početni zbir (koji je uvek 0 u ovoj verziji). Rekurzija se odvija na isti način kao u prethodnom primeru. Pri svakom novom pozivu, prozleđuje se odsečeni deo liste i broj uvećan za vrednost „glave“ – što je vrednost prvog elementa prosleđene liste.

Deklaracija novih tipova vrši se po jednostavnom šablonu. Navodi se teč „type“, posle koje sledi identifikator tipa, a zatim definicija.

```
type A = int * int
type B = {FirstName:string; LastName:string}
type C = Krug of int | Pravougaonik of int * int
type D = Dan | Mesec | Godina
type E<'a> = Choice1 of 'a | Choice2 of 'a * 'a

type A = int * int
type B =
  {FirstName: string;
   LastName: string;}
type C =
  | Krug of int
  | Pravougaonik of int * int
type D =
  | Dan
  | Mesec
  | Godina
type E<'a> =
```

```
  | Choice1 of 'a
  | Choice2 of 'a * 'a
```

Alijasi

```
type PreimenujString = string
type Naziv = string
let NekiNAziv:PreimenujString = "Neki naziv"

type PreimenujString = string
type Naziv = string
val NekiNAziv : PreimenujString = "Neki naziv"
```

N-torke (tples)

```
let par = 1,true,"str"
//val par : int * bool * string = (1, true, "str")
let par2 = (1,8)
//val par2 : int * int = (1, 8)
```

Zapisi (records)

```
type Fakultet ={NazivFakulteta:Naziv; GodinaOsnivanja:int}
(*type Fakultet =
  {NazivFakulteta: Naziv;
   GodinaOsnivanja: int;}*)
```

Diskriminisana unija (discriminated union)

```
type JedinicaMere = Cm | Inch | Mile
(*type JedinicaMere =
  | Cm
  | Inch
  | Mile*)

type Ime =
   | Nadimak of string
   | ImePrezime of string * string
//STABLA U F# E je kraj T je čvor
type Drvo<'a> =
    | E
    | T of Drvo<'a> * 'a * Drvo<'a>

let levoDete = T(E,"Marta",E)
//val levoDete : Drvo<string> = T (E,"Marta",E)

let desnoDete = T(E,"Jasha",E)
let tataIDeca = T(levoDete,"Goran",desnoDete)
//val tataIDeca : Drvo<string> = T (T (E,"Marta",E),"Goran",T (E,"Jasha",E))
```

Enum

```
type Pol = | Masculinum = 1 | Feminum = 2
(*type Pol =
  |  Masculinum  =  1
  |  Feminum  =  2*)
let g = Pol.Masculinum
//val g : Pol = Masculinum
```

Klasa

```
type Product (code:string, price:float) =
   let isFree = price=0.0
   new (code) = Product(code,0.0)
   member this.Code = code
   member this.IsFree = isFree
(*type Product =
  class
    new : code:string -> Product
    new : code:string * price:float -> Product
    member Code : string
    member IsFree : bool
  end*)

let p = Product("X123",9.99)
let p2 = Product("X123")
p2.IsFree //val it : bool = true
```

Struct

```
type Proizvod =
    struct
        val kod:string
        val cena:float
        new ( kod ) = {kod = kod; cena = 0.0 }
    end
(*type Proizvod =
  struct
    new : kod:string -> Proizvod
    val kod: string
    val cena: float
  end
*)

let proizvod = Proizvod("abc")
```

3. Primeri

N-torke (*Tuples*)

N – torka predstavlja uređenu listu elemenata, gde su precizirani tipovi kojima ti elementi pripadaju.

```
let par = 1,2
let trojka = (1, true, "tekst")

val par : int * int = (1, 2)
val trojka : int * bool * string = (1, true, "tekst")
```

Ukoliko int ne posmatramo kao apstraktan pojam, već kao uređen skup celih brojeva (ograničen), možemo reći da je par (1,2) samo jedan podskup dekartovog proizvoda skupa celih brojeva nad samim sobom.[4] Odatle se može izvući objašnjenje za ovakav potpis vrednosti par:

```
int * int
```

Primetno je da se radi o svojevrsnim „proizvodu“. Isto tako uređena trojka ima potpis:

```
int * bool * string
```

te je jasno da je prvi element iz skupa celih brojeva, drugi uzima vrednosti iz skupa {true, false} a treći je iz skupa svih mogućih stringova.

N – torke mogu sadržati i generičke tipove (’a).

```
let stampajElemente uredjenPar =
    let (a,b) = uredjenPar
    printfn "Prvi element je %A, drugi %A" a b

stampajElemente par

val stampajElemente : 'a * 'b -> unit
Prvi element je 1, drugi 2
val it : unit = ()
```

Funkcija stampajElemente ispisuje elemente na konzoli, ulazni argument je „proizvod“ generickog tipa, a izlani je tipa unit (printfn uvek vraća tip unit). Ovoj funkciji je prosleđen ranije definisan par (1,2).

Osim generika, elementi n-torki mogu biti druge n-torke, kao i tipovi definisani od strane korisnika. To oslikava sledeći primer. Koristim već definisane n-torke „par“ i „trojka“

```
let kompleksnaDvojka = par,trojka
let prviElement = fst kompleksnaDvojka

val kompleksnaDvojka : (int * int) * (int * bool * string) =
  ((1, 2), (1, true, "tekst"))
val prviElement : int * int = (1, 2)
```

U ovom primeru spojili smo par i trojku u novi par kompleksnaDvojka. Prvi element vrednosti kompleksnaDvojke (1,2) dobijen je primenom funkcije fst (Member of Microsoft.FSharp.Core.Operators ; Return the first element of a tuple, fst (a,b) = a). Dakle fst se može primenljivati samo na uređene liste dva elementa – to jest parove. Funkcija koja vraća drugi element je snd.

```
type Osoba = {Ime:string; Prezime:string}
type Smer = ISiT | OM | MEN | UK
type Index = int * Smer
type Student = Osoba * Index

let std1 = {Ime = "Jasha" ; Prezime = "Petrov"},(567,ISiT)


type Osoba =
  {Ime: string;
   Prezime: string;}
type Smer =
  | ISiT
  | OM
  | MEN
  | UK
type Index = int * Smer
type Student = Osoba * Index
val std1 : Osoba * (int * Smer) = ({Ime = "Jasha";
                                    Prezime = "Petrov";}, (567, ISiT))
```

U ovom primeru kreiran je tip Osoba (Record), Smer (Union), Student i Index koji su aberivacije(alias). Interesantno je da vrednist std1 u ovom slučaju nije tipa Student, upravo zbog korišćenja aberivacije!!! Naime, kompajler je ovo prepozno kao tip:

```
Osoba * (int * Smer)
```

Međutim, ako bismo naglasili:

```
let std1:Student = {Ime = "Jasha" ; Prezime = "Petrov"},(567,ISiT)
val std1 : Student = ({Ime = "Jasha";
                       Prezime = "Petrov";}, (567, ISiT))
```

Dekonstrukcija n – torke odvija se po sledećem šablonu:

```
let x1,x2,x3,x4 = (1,2,3.0,true)

val x4 : bool = true
val x3 : float = 3.0
val x2 : int = 2
val x1 : int = 1
```

Broj elemenata sa leve strane, odgovara dužini n – torke. U slučaju da neki elementi nisu od značaja, koristi se uobičajen znak „_“(wildcard).

```
let _,_,x3,x4 = (1,2,3.0,true)

val x4 : bool = true
val x3 : float = 3.0
```

Prepoznavanje obrazaca (Pattern matching) vrši se na jednostavan način.

```
let stdIndex1 = (444,UK)
match stdIndex1 with
    | broj,ISiT -> printfn "Ovo je smer ISiT i broj %i" broj
    | broj,OM -> printfn "Ovo je smer OM i broj %i"  broj
    | broj,MEN -> printfn "Ovo je smer MEN i broj %i"  broj
    | broj,UK -> printfn "Ovo je smer UK i broj %i" broj

val stdIndex1 : int * Smer = (444, UK)

Ovo je smer UK i broj 444
val it : unit = ()
```

U prethodnom primeru je kreiran par int*Smer koji zapravo predstavlja gore definisani Index, i kao takav se ispituje u match..with izrazu.

Sledeći primer oslikava korišćenje prepoznavanja obrazaca sa korišćenjem logičkih operatora. Prvi slučaj ispisuje drugi element para ako je prvi jednak 2, 3 ili 4.
Drugi slučaj je nemoguć jer par ne može istovremeno imati i 0 i 1 kao drugi element.
Izraz nakon znaka (->) u trećem redu odnosi se na sve slučajeve koji nisu obuhvaćeni u prva dva reda, dakle ispisuju se prvi i drugi element.

```
match (1,0) with
    | (2,x) | (3,x) | (4,x) -> printfn "x=%A" x
    | (_,0) & (_,1) -> printfn "Ovo se nece desiti"
    | (x,y) -> printfn "(x:%i,y:%i)" x y

(x:1,y:0)
val it : unit = ()
```

N – torke predstavljaju dobar način da se kao rezultat funkcije vrati više od jedne informacije što prikazuje sledeći složeniji primer.

Recimo da zelimo da prebrojimo pojavljivanja pomoćnog glagola jesam u skraćenom obliku (sam, si, je, smo, ste, su) u nekoj rečenici, a pritom želimo da vratimo i broj reči u istoj.

Rešio sam da problem rešim na složeniji način, pomoću dve funkcije. Jedna funkcija će primiti rečenicu, podeliti je i pozvati funkciji koja će se baviti prebrojavanjem pojavljivanja glagola „jesam“.

Dakle, funkcija *brojReciIGlagolaJesam* (druga definisana u sledećem primeru) prima rečenicu. Niz „nizReci“ dobijen odescanjem rečenice na razmacima (`nizReci = recenica.Split...`), brojReci je vrednost koja pokazuje dužinu niza.

```
let rec pojavaGlagolaJesam (niz:string array,brojac,pojavljivanja) =
    match brojac with
    | -1 -> pojavljivanja
    | _ -> match niz.[brojac] with
           | "sam" -> pojavaGlagolaJesam(niz,brojac-1,pojavljivanja + 1)
           | "si" -> pojavaGlagolaJesam(niz,brojac-1,pojavljivanja + 1)
           | "je" -> pojavaGlagolaJesam(niz,brojac-1,pojavljivanja + 1)
           | "smo" -> pojavaGlagolaJesam(niz,brojac-1,pojavljivanja + 1)
           | "ste" -> pojavaGlagolaJesam(niz,brojac-1,pojavljivanja + 1)
           | "su" -> pojavaGlagolaJesam(niz,brojac-1,pojavljivanja + 1)
           | _  -> pojavaGlagolaJesam(niz,brojac-1,pojavljivanja)
```

Tako „iseckana“ rečenica, prosleđuje se rekurzivnoj funkciji pojavaGlagolaJesam. Koja se kreće kroz niz od prve do poslednje reči uz pomoć brojača.

```
let brojReciIGlagolaJesam (recenica:string) =
    let nizReci = recenica.Split[|' '|]
    let brojReci = nizReci.Length
    let brojGalgola = pojavaGlagolaJesam(nizReci,brojReci - 1,0)
    (brojReci,brojGalgola)

brojReciIGlagolaJesam "Kruske su bile slatke, pojeli smo ih, jabuke ste pojeli vi, a tortu je smazao Grisa"

val pojavaGlagolaJesam :
  niz:string array * brojac:int * pojavljivanja:int -> int
val brojReciIGlagolaJesam : recenica:string -> int * int
val it : int * int = (16, 4)
```

Moguće je izvršiti i poređenja torki (Napomena: argumenti funkcije printfn se nisu pravilno iskopirali iz VS MS, potrebno je da se uvuku)

```
let poredjenjeTorki prvaTorka drugaTorka =
    if prvaTorka > drugaTorka then printfn "Prva torka %A je veća od druge %A"
prvaTorka drugaTorka
    if prvaTorka = drugaTorka then printfn "Prva torka %A je jednaka drugoj %A"
prvaTorka drugaTorka
    if prvaTorka < drugaTorka then printfn "Prva torka %A je manja od druge %A"
prvaTorka drugaTorka

poredjenjeTorki (1,2) (1,3)

val poredjenjeTorki :
  prvaTorka:'a -> drugaTorka:'a -> unit when 'a : comparison
Prva torka (1, 3) je veća od druge (1, 2)
val it : unit = ()
```

Poređenje može biti u najmanju ruku zabavno! U sledećem primeru upoređena su dva indeksa, izgleda da je indeks sa smera za operacioni menadžment važnij od indeksa sa smera IsiT! Isto tako mala slova stringa „ab“ u drugom slučaju neuporedivo su „veća“ od slova „AB“. Za prvu i drugu pojavu postoji valjano objašnjenje. Poređenje se vrši po redosledu elemenata, kako su prvi elementi ovih parova jednaki 2, prelazi se na sledeći element. Ako se vratimo na definiciju Smer, vidimo da je OM definisan posle smera IsiT, što mu daje veći indeks.

```
poredjenjeTorki (2,ISiT) (2,OM)
poredjenjeTorki (2,"ab") (2,"AB")
```

```
Prva torka (2, ISiT) je manja od druge (2, OM)
Prva torka (2, "ab") je veća od druge (2, "AB")
```

Zapisi (*Records*)

Zapis kao (Record) možemo definisati kao listu imenovanih tipova. Njihovo deklarisanje odvija se po sledećem šablonu:

```
type imeTipa = {
    [ mutable ] nazivPolja1 : tip1;
    [ mutable ] nazivPolja2: tip2;
    ...
}


type FakultetUB = { Naziv:string; BrojNastavnika:int}

type FakultetUB =
  {Naziv: string;
   BrojNastavnika: int;}
```

Kreiranje vrednosti se vrši na jednostavan način:

```
let FON = {Naziv = "Fakultet organizacionih nauka"; BrojNastavnika = 150}

val FON : FakultetUB = {Naziv = "Fakultet organizacionih nauka";
                        BrojNastavnika = 150;}
```

Kako se radi o zapisu, nismo sprečeni da unesemo potrebne podatke u proizvoljnom redosledu:

```
let MasF = {BrojNastavnika = 170; Naziv = "Masinski fakultet"}

val MasF : FakultetUB = {Naziv = "Masinski fakultet";
                         BrojNastavnika = 170;}
```

Kompajler ne dozvoljava definisanje dva tipa rekorda sa istim imenom, međutim do zabune najčešće dolazi ako se kreira novi tip zapisa koji ima istovetne parove `nazivPolja: tip`.

```
type VisokaSkola = { Naziv:string; BrojNastavnika:int}
let MATF = {Naziv="Matematicki fakultet"; BrojNastavnika = 145}

type VisokaSkola =
  {Naziv: string;
   BrojNastavnika: int;}
val MATF : VisokaSkola = {Naziv = "Matematicki fakultet";
                          BrojNastavnika = 145;}
```

Definisan je novi tip VisokaSkola koji takođe ima Naziv i BrojNastavnika. Pri unošenju novog fakulteta dešava se da ga kompajler poveže sa poslednjom odgovarajućom definicijom koja mu je pružena. Tako je MATF greškom protumačen kao tip visoke škole. Problem se rešava jednostavno, na više načina. Tip se može naglasiti dodavanjem imena tipa na barem jedan atribut.

```
let FPN = {FakultetUB.Naziv = "Fakultet politickih nauka"; BrojNastavnika = 112}

val FPN : FakultetUB = {Naziv = "Fakultet politickih nauka";
                        BrojNastavnika = 112;}
```

Ili pak krajnje eksplicitno:

```
let ETF:FakultetUB = {BrojNastavnika = 170; Naziv ="Elektrotehnički fakultet"}

val ETF : FakultetUB = {Naziv = "Elektrotehnički fakultet";
                        BrojNastavnika = 170;}
```

Pristup podacima u zapisu moguće je vršiti i putem „.dot“ notacije koja je korišćena za izračunavanje razlike u broju nastavnog osoblja i ispis u okviru funkcije čiji je zadatak da odredi na kom fakultetu ima više nastavnika i za koliko.

```
let viseNastavnika (faks1:FakultetUB, faks2:FakultetUB) =
    let razlikaUBroju = faks1.BrojNastavnika - faks2.BrojNastavnika
    match razlikaUBroju with
    | 0 -> printfn "Nema razlike"
    | _ when razlikaUBroju > 0 -> printfn "%s ima %i nastavnika vise nego %s"
                                          faks1.Naziv razlikaUBroju faks2.Naziv
    | _ when razlikaUBroju < 0 -> printfn "%s ima %i nastavnika manje nego %s"
                                          faks1.Naziv (razlikaUBroju*(-1)) faks2.Naziv
    | _ -> printfn "Ne mogu da odredim"

viseNastavnika (FON,ETF)

Fakultet organizacionih nauka ima 20 nastavnika manje nego Elektrotehnički fakultet

val viseNastavnika : faks1:FakultetUB * faks2:FakultetUB -> unit
val it : unit = ()
```

Isti zadatak može se obaviti i na uobičajeni način bez „.dot“ notacije – korišćenjem dekonstruktora (deconstruct):

```
let viseNastavnika2 faks1 faks2 =
    let {FakultetUB.Naziv = n1; BrojNastavnika = br1} = faks1
    let {FakultetUB.Naziv = n2; BrojNastavnika = br2} = faks2
    let razlikaUBroju = br1 - br2
    match razlikaUBroju with
    | 0 -> printfn "Nema razlike"
    | _ when razlikaUBroju > 0 -> printfn "%s ima %i nastavnika vise nego %s"
                                          n1 razlikaUBroju n2
    | _ when razlikaUBroju < 0 -> printfn "%s ima %i nastavnika manje nego %s"
                                          n1 (razlikaUBroju*(-1)) n2
    | _ -> printfn "Ne mogu da odredim"

viseNastavnika2 FON FPN

Fakultet organizacionih nauka ima 38 nastavnika vise nego Fakultet politickih nauka

val viseNastavnika2 : faks1:FakultetUB -> faks2:FakultetUB -> unit
val it : unit = ()
```

Formatiranje ispisa zapisa (record)

```
printfn "%A" FON
printfn "%O" FON
FON.ToString()

{Naziv = "Fakultet organizacionih nauka";
 BrojNastavnika = 150;}
FSI_0018+FakultetUB

val it : string = "FSI_0018+FakultetUB"
```

Razlika između %A i %O ispisa je što F# ima ugrađen mehanizam za ispisivanje osnovnih tipova, dok se navođenjem %O poziva metoda Object.ToString() koja ne daje očekivani ispis.[5]

Unije (Discriminated unions)

Ako se pri kreiranju n – torki (tuples) i zapisa (record) radilo o svojevrsnom proizvodu tipova:

```
let trojka = (1,2.0,true)

val trojka : int * float * bool = (1, 2.0, true)
```

onda se može reći da se kod unije javlja svojevrstan „zbir“.[6] Pretpostavimo da želimo da definišemo funkciju koja će primati samo argument tipa int ili argument tipa bool. Tada bi bilo najpodesnije da kreiramo novi tip IntOrBool koji će reprezentovati sve int-ove „plus“ bool - ove. Taj novi tip bi bio definisan na sledeći način:

```
type type-name =
    | case-identifier1 [of [ fieldname1 : ] type1 [ * [ fieldname2 : ]
type2 ...]
    | case-identifier2 [of [fieldname3 : ]type3 [ * [ fieldname4 : ]type4
...]
...
```

```
type IntOrBool =
    | IdentifikatorInta of int
    | IdentifikatorBoola of bool

type IntOrBool =
  | IdentifikatorInta of int
  | IdentifikatorBoola of bool
```

Treba napomenuti da je prva uspravna linija opciona, isto tako nije neophodno navođenje tipa posle identifikatora:

```
type PaoIliPolozio =
      Pao
    | Polozio of int

type PaoIliPolozio =
  | Pao
  | Polozio of int
```

Greška je ukoliko identifikator počinje malim slovom

```
type Oprez =
    | VelikiPocetak of int
    | niskiStart

Script1.fsx(752,7): error FS0053: Discriminated union cases and exception labels must
be uppercase identifiers
```

Kombinovanje tipova je naravno omogućeno što će slikovito biti objašnjeno na sledećem primeru. Pretpostavimo da stdent FON-a, Marko pravi letnju žurku početkom septembarskog roka. Nekim osobama je pristup strogo zabranjen, neke su dobrodošle, a neke zvanice su veoma bitne i sede na posebno za njih numerisanim mestima. Naravno, ako je osoba značajna, njeno prisustvo se ne dovodi u pitanje!

```
type PristupIliZnacajnost =
    | Pristup of bool
    | Znacaj of int

type NekaOsoba = {Ime:string; Godine:int; Pristup: PristupIliZnacajnost}
```

Marko nije bio zadovoljan ovim kodom, želeo je da omogući svojim zvanicama da pozovu još pratilaca. Marko je dodao kod:

```
type Gost<'NekaOsoba> =
    | BezPratnje of NekaOsoba
    | SaPratnjom of NekaOsoba*Gost<'NekaOsoba>

type Gost<'NekaOsoba> =
  | BezPratnje of NekaOsoba
  | SaPratnjom of NekaOsoba * Gost<'NekaOsoba>
```

Kada je napisao ovaj kod, uhvatio se za glavu shvativši da je kreirao rekurzivan tip. Gost može biti bez pratnje – dakle NekaOsoba (gost sam), ili sa pratnjim NekaOsoba*Gost (on sam, gost pratilac – koji takođe može povesti nekog!).

Odlučio je da napiše par metoda koje će služiti za proveru pridošlih gostiju. Matoda „provera“ kao ulazni parametar prima osobu tipa NekaOsoba, potom koristi „pattern matching“ izraz za tip PristupIliZnazajnost.

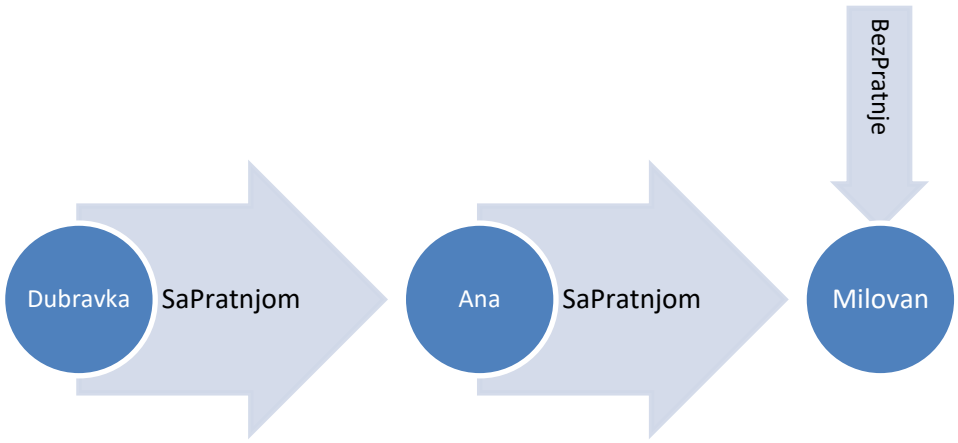
Vrednost dalliNe je tipa bool kao što je definisano unijom PristupIliZnazajnost, dok vredost mesto uzima int broj. Printfn u oba slučaja ispisuje ime osobe, a potom informaciju o pristupu ili mestu.

Svrha metode pregledaj goste je da se kreće kroz strukturu pridošlica. Neki gosti ne dovode nikoga, to jest dolaze sami, drugi pak mogu povesti još osoba. Ako Gost1 dolazi sa Gost2, i Gost2 ne vodi još osoba, onda se smatra da Gost2 dolazi bez pratnje, to jest, Gosta2 posmatramo samo kao pratnju Gosta1 (Linijska struktura). Na taj način ćemo se kretati kroz svojevrsnu listu gostiju. Ako gost dolazi pez pratnje, on se odmah proverava, ukoliko pak ima pratioca, prvo se proverava pratilac, a na kraju i sam gost.

```
let provera osoba =
    match osoba.Pristup with
    | Pristup daIliNe -> printf "Osoba %s ulazak: %b;" osoba.Ime daIliNe
    | Znacaj mesto -> printfn "%s je posebna i sedi na mestu %i" osoba.Ime mesto

let rec pregledajGoste gost =
    match gost with
    | BezPratnje onSam -> provera onSam
    | SaPratnjom (onSam,pratnja) -> pregledajGoste pratnja
                                    provera onSam
```

Marko je kreirao osobe. Gost1 je Milovan koji je rešio da ne vodi nikoga i nikako nije dobrodošao na žurku jer se nabacuje Markovoj devojci Dubravki. Gost2 - Ana, je ludo zaljubljena u Milovana, i pitala ga je da sa njom pođe na žurku, što je on oberučke prihvatio. Gost3 je Dubravka, Anina starija sestra, koja želi da sa sobom povede Anu, ne znajući da je ova pozvala i Milovana.



Dubravka je došla na zabavu sa sestrom Anom i njenim pratiocem Milovanom. Funkcija je vratila rezultat koji je naljutio Marka:

```
Osoba Milovan ulazak: false;Osoba Ana ulazak: true;Dubravka je posebna i sedi na mestu
3
```

Morće da razmisli kako da reši ovu situaciju.

```
let ana = {Ime = "Ana"; Godine = 20; Pristup = Pristup true}
let milovan = {Ime = "Milovan"; Godine = 24; Pristup = Pristup false}
let dubravka = {Ime = "Dubravka"; Godine = 22; Pristup = Znacaj 3}

let gost1 = BezPratnje milovan
let gost2 = SaPratnjom (ana,gost1)
let gost3 = SaPratnjom(dubravka,gost2)

pregledajGoste gost3

Osoba Milovan ulazak: false;Osoba Ana ulazak: true;Dubravka je posebna i sedi na mestu
3
```

Kao i kod prethodnih tipova, i kod unija može doći do zabune pri definisanju tipova koji imaju istovetne identifikatore. Novodefinisana vrednost će se povezati sa tipom koji je poslednji definisan.

```
type Tip1 =
    | A of int
    | B of bool
type Tip2 =
    | A of int
    | B of bool

let tip1 = A 2

type Tip1 =
  | A of int
  | B of bool
type Tip2 =
  | A of int
  | B of bool
val tip1 : Tip2 = A 2
```

Problem se rešava eksplicitnim navođenjem:

```
let tip1Eksp = Tip2.A 2

val tip1Eksp : Tip2 = A 2
```

Opcije (*Option*)

Opcija (Options) je specijalni podtip unije koji obuhvata svega dva slučaja i ima sledeću deklaraciju:

```
type Option<'a> =
   | Some of 'a
   | None
```

Opcije su korisne ukoliko je potrebno predstaviti neispravne ili nedostajuće vrednosti.

```
let okInt = Some 2
let neOkInt = None

val okInt : int option = Some 2
val neOkInt : 'a option
```

Pri definisanju tipa koji referencira na tip Opcija koriste se dva zapisa:

```
type NazivOsobe = option<string>
type NazivUstanove = string option
```

```
type NazivOsobe = string option
type NazivUstanove = string option
```

Upotreba i match izraz

Mnoge bibliotečke funkcije kao tip povrate vrednosti imaju upravo Opciju (eng. Option). Ovde je dat prikaz kako bi moglda da izgleda jedna takva finkcija. Pretpostavka je da string koji zapravo predstavlja ceo broj želimo da prebacimo u tip int korišćenjem funkcije Parse koja baca izuzetak ukoliko prosleđeni string nije broj ili sadrži neke nedozvoljene karaktere. Ovaj deo koda smešta se u try blok i u slučaju bilo koje greške (with _) vraća se opcija None. U slučaju uspeha pri parsiranju vraća se Some i.

```
let tryParseOption intStr =
   try
      let i = System.Int32.Parse intStr
      Some i
   with _ -> None
```

Funkcija vratiInt prima string kao parametar, poziva funkciju tryParseOption kojoj ga prosleđuje u okviru match izraza. Ukoliko je parsiranje završeno uspešno, vraća se broj i, u suprotnom slučaju -2147483648 što je najmanji broj iz int domena.

```
let vratiInt str =
    match tryParseOption str with
    | Some i -> i
    | None -> System.Int32.MinValue

vratiInt "789"
```

None i null

Kako je F# deo .NET platforme, prinuđen je da radi sa null konceptom. Napravljen je tip Nadimak koji je unija sa samo jednim slučajem – ImaIliNema koji je tipa Option. Kreirane su dve promenljive, jedna sa nadimkom (markoMarkovic) i jedna bez (cvetaCvetic).

```
type Nadimak =
    | ImaIliNema of Option<string>

let markoMarkovic = ImaIliNema (Some "Mare")
let cvetaCvetic = ImaIliNema (None)
```

Posle izvršenja u interaktivnom prozoru dobijen je sledeći ispis:

```
type Nadimak = | ImaIliNema of Option<string>
val markoMarkovic : Nadimak = ImaIliNema (Some "Mare")
val cvetaCvetic : Nadimak = ImaIliNema null
```

4. Zaključak

Kreiranje promenljivih na koje smo navikli u F# radije se posmatra kao odstupanje nego kao pravilo, vrednosti su nepromenljive - konstante. Velika prednost jezika ogleda se u tome što je deo .NET „frejmvorka“, što omogućava da se koriste mnogobrojne biblioteke i integracije sa drugim jezicima. Tipovi podataka karakteristični za F#, pre svega zapisi i unije, veštim kombinovanjem mogu biti dobri pandani klasama. Posebno interesantan i koristan koncept koji je ugrađen u jezik je „pattern-matching“. Ideja nepromenljivosti (immutability) primorava korisnike da izmene način razmišljanja. Ideje za rešavanje pojdeinih problema u F# mogu biti od koristi za optimizaciju i rešavanje sličnih problema u drugim jezicima.

5. Izvori

[1] https://fsharp.org/about/
[2] Andrew W. Appel. A Critique of Standard ML. Princeton University, revised version of CS-TR-364-92, 1992.
[3] https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/list.filter%5B't%5D-function-%5Bfsharp%5D
[4] http://fsharpforfunandprofit.com/posts/tuples/
[5] http://fsharpforfunandprofit.com/posts/records/
[6] http://fsharpforfunandprofit.com/posts/discriminated-unions/