Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.9
DOI: https://doi.org/10.24867/01BE19Zaric

FUNKCIONALNO PROGRAMIRANJE I PROGRAMSKI JEZIK *F#*

FUNCTIONAL PROGRAMMING AND F# PROGRAMMING LANGUAGE

Jovica Zarić, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *U radu su opisani osnovni koncepti funkcionalne programske paradigme i njeno poređenje sa objektno-orijentisanom odnosno proceduralnom paradigmom. Opisan je programski jezik F# kroz svoje najvažnije osobine.*

Ključne reči: *funkcionalno programiranje, objektno-orijentisano programiranje, proceduralno programiranje, F#*

Abstract – *This paper describes key functional programming concepts and compares functional programming with objected-oriented and procedural programming. It presents key features of F# programming language.*

Keywords: *functional programming, object-oriented programming, procedural programming, F#*

1. UVOD

Programiranje je stvaranje računarskih programa (aplikacija) za obavljanje zadataka kroz niz definisanih, automatskih ili manuelnih, operacija. Računari su prisutni u modernom društvu u različitim oblicima: *PC* računari, mobilni i *tablet* uređaji, pametni satovi, računarski sistemi u prevoznim sredstvima, kućnim aparatima, poljoprivrednim mašinama, fabrikama.

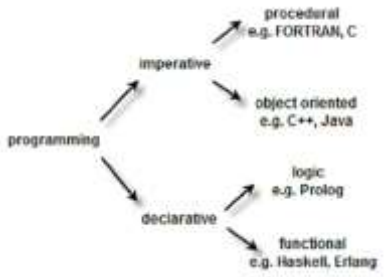
Pored velikog broja računarskih platformi, postoji i veliki broj programskih jezika, načina i alata za razvoj programa. Neki jezici su prilagođeni samo određenim platformama, neki se mogu koristiti za više njih. Razvijeno je i nekoliko programskih tehnika (paradigmi) za pisanje programa. Programska paradigma predstavlja pogled koji programeri imaju nad programom i njegovim izvršavanjem.

Paradigmom se definiše način pisanja, organizacije i izvršavanja koda kako bi isti bio pogodan i za čovjeka koji ga piše, čita i treba za razumije, tako i za računar koji ga izvršava. Slika 1.1 predstavlja osnovnu podjelu programskih paradigmi, podjelu na imperativnu i deklarativnu grupu.

Imperativna u koju spadaju proceduralno programiranje [1] i objektno-orijentisano programiranje (OOP) [2], akcenat stavlja na način na koji će se zadatak obaviti, dok deklarativno (funkcionalno i logičko) na sam zadatak, šta se treba obaviti, a ne i kako.

NAPOMENA:

Ovaj rad proistekao je iz master rada čiji mentor je bio dr Aleksandar Kupusinac, vanr. prof.



Slika 1.1 *Podjela programskih paradigmi*

2. UVOD U FUNKCIONALNO PROGRAMIRANJE

Na samom početku ere računara i programskih jezika (1940-tih godina), programski jezici su razvijani u skladu za arhitekturom računara. Jedan od prvih pristupa u razvoju programa je bio imperativni jer je konceptualno blizak hardveru računara. Izvršavanje programa predstavlja izvršavanje instrukcija programa od strane procesora koje mijenjaju sadržaj dodjeljene memorije i tok izvršavanja programa.

Funkcionalno programiranje [3] je nastalo primjenom matematičkih principa u programiranju. Razvojem matematičkog stila programiranja dolazi do udaljavanja od konkretne računarske arhitekture koja izvršava program što za posljedicu ima slabije performanse u odnosu na jezike imperativne paradigme. Funkcionalno programiranje posmatra pojedinačne operacije kao izračunavanja metamatičkih funkcija. Spada u deklarativne programske paradigme jer akcenat stavlja na to šta posmatrani kod radi, a ne i kako radi. Podržava funkcije koje koriste samo svoje ulazne parametre tako da pozivanje funkcije sa istim ulaznim argumentima uvijek vraća isti rezultat. Ovakav pristup omogućava lakše razumijevanje ponašavanja aplikacije, eliminaciju izmjena vrijednosti rezultata prethodnih izračunavanja (*mutable data*) i stanja aplikacije odnosno izazivanje *side* efekata. Programski jezici *C#, F#, Common Lisp, Clojure, Erlang, Haskell, Java, Javascript, Kotlin* pripadaju *impure* grupi funkcionalnih jezika jer pored funkcionalnog koncepta podržavaju i objektno-orijentisani i imperativni. *Pure* funckionalni jezici, npr. *Clean, Curry, Elm, Mercury,* podržavaju samo funkcionalno programiranje, ali su mnogo manje zastupljeni u razvoju komercijalnih softverskih aplikacija.

Lamdba račun [4], nastao u 1930-tim godina od strane američkog matematičara Alonza Čerča, predstavalja osnovu funkcionalnog programiranja. Lambda račun uvodi neimenovane, anonimne, funkcije koje opisuju neophodne ulazne parametre i način izračunavanja rezultata (listing 2.1).

square_num(x, y) = $x^2 + y^2$
(x, y) -› $x^2 + y^2$

Listing 2.1 *Koncept anonimne funkcije*

Lambda račun podržava samo funkcije sa jednim ulaznim parametrom koristeći *curring*. Funkcija sa dva parametra, može se predstaviti kao funkcija koja prima jedan parametar i vraća novu funkciju koja, takođe, prima jedan parametar. Pomoću *curring*-a, funkcija sa više parametara se predstavlja kao niz funkcija od kojih svaka prima samo po jedan parametar. Lambda račun posmatra funkcije kao vrijednosti prve klase (*first-class citizen*), te se funkcije mogu koristiti kao ulazni parametri ili biti vraćene kao rezultat drugih funkcija.

3. OSNOVNI KONCEPTI FUNKCIONALNOG PROGRAMIRANJA

Primjeri koda u ovom dijelu su pisani korišćenjem *Javascript* programskog jezika jer isti posjedu i funkcionalne elemente.

3.1 *Pure* (čiste) funkcije

Pure funkcijom se smatra jednostavna funkcija koja zadovoljava sljedeće kriterijume:

- Prima najmanje jedan ulazni parametar
- Pristupa vrijednostima samo svojih ulaznih parametara
- Uvijek vraća rezultat
- Za iste ulazne argumente uvijek vraća isti rezultat

Pure funkcija koristi vrijednosti samo svojih ulaznih parametara i ne pristupa ili mijenja vrijednost drugih promjenljivih. Ovim principom se izbjegavaju nepoželjni *side* efekti u programiranju jer utiču na izvršavanje drugih funkcija, unose *bug*-ove u kod i otežavaju otkrivanje istih. Funkcija koja ne prima nijedan ulazni parametar i ne može da mijenja vrijednosti promjeljivih van svog *scope*-a je beskorisna u programiranju jer može da vraća samo konstantnu vrijednost. Takođe, funkcija mora da vrati vrijednost kako bi manifestovala svoju namjenu i izvršavanje. *Pure* funkcija za iste ulazne paramatre uvijek vraća isti rezultat, te je neophodno da i funkcije koje ona poziva budu, takođe, *pure*. Listing 3.1.1 predstavlja primjer *pure* funkcije.

```
function Sum(a, b) {
return a + b;
}
```

Listing 3.1.1 *Pure* funkcija

Pure funkcije same po sebi nisu dovoljne za razvoj modernih softverskih aplikacija. Aplikacije podrazumijevaju upis i čitanje iz baze podataka ili fajla, otvaranje konekcije ka udaljenim serverima, te pojedine funkcije rade mnogo više od same obrade ulaznih parametara. Izvršavanje ovakvih funkcija može dovesti do pojave *side* efekata ili *exception*-a (izuzetaka) čime njihovo izvršavanje postaje nepredvidivo i one postaju *impure*. Funkcionalno programiranje ne može eliminisati *impure* funkcije, ali teži da njihovu upotrebu ograniči na ona mjesta gdje su zaista neophodne.

3.2 Immutability

U imperativnom programiranju, moguće je varijabli promijeniti prethodno dodjeljenu vrijednost (*mutable data*). Listing 3.2.1 predstavlja dozvoljenu operaciju u ovoj programskoj paradigmi.

```
var a = 3;
a = a + 1;
```

Listing 3.2.1 *Mutable* varijabla

Funkcionalno programiranje podržava *immutability* koncept koji zabranjuje izmjenu vrijednosti varijabli odnosno dozvoljava samo preuzimanje vrijednosti varijable. Samim tim koncept varijable ne postoji, ali se ipak naziv varijabla (promjenljiva) koristi iz istorijskih razloga. Funkcionalno programiranje čuva vrijednosti u konstantama. Ukoliko je potrebno izmijeniti vrijednost, kreira se nova konstanta sa novom vrijednošću. Ovim pristupom se piše jednostavniji i sigurniji kod, eliminišu se čak i slučajne izmjene vrijednosti.

3.3 Rekurzija

Immutability koncept onemogućava upotrebu *for*, *while* i *do while* petlji jer koriste brojačke promjeljive i promjeljive čijim bi se mijenjanjem vrijednosti postigli uslovi za izlazak iz petlje. Iz ovih razloga, koncept rekurzije (*recursion*) je veoma zastupljen u funkcionalnom programiranju. U svakom koraku rekruzije, funkcija vraća konkretnu vrijednost ili poziva samu sebe sa novim, izračunatim, vrijednostima argumenata poziva. Listing 3.3.1 predstavlja primjer rekurzivne funkcije za sabiranje niza cijelih brojeva.

```
function sum(first, last, currentSum) {
	if (first > last) {
		return currentSum;
	}
	return sum(first + 1, last,
	currentSum + first);
}
```

Listing 3.3.1 Primjer rekurzije

Osnovni elementi rekurzivne funkcije su:

- *Base case* – uslov za završetak rekurzije
- *Recursive call* – pozivanje funkcije u samoj sebi

3.4 Funkcije višeg reda

Za funkciju u funkcionalnom programiranju se kaže da je *first-class citizen* jer se funkcija tretira kao vrijednosti (slično kao *string* ili *int*). Ovo omogućava uvođenje koncepta funkcija višeg reda (*higher-order functions*) koje mogu da primaju funkciju kao argument ili vraćaju funkciju kao povratnu vrijednost. Takođe, definicija funkcije se može dodijeliti varijabli ili smjestiti u strukturu podataka. Funkcionalno programiranje podržava i *closure* princip koji povratnoj anonimnoj funkciji omogućava pristup svim lokalnim variablama funkcije koja ju je kreirala. Listing 3.4.1 predstavlja primjer anonimne funkcije kao povratne vrijednosti.

```
function makeAdd (increment) {
	return function (x) {
		return x + increment;
	};
}
```

Listing 3.4.1 *Anonimna funkcija kao povratna vrijednost*

3.5 *Curring*

Curring predstavlja razbijanje funkcije koja prima više parametara na kompoziciju više funkcija od kojih svaka prima samo jedan parametar.

Primjenom ovog principa, funkcija sa dva parametra se može predstaviti kao funkcija koja prima jedan parametar i vraća drugu funkciju koja prima drugi parametar. Izvršavanjem druge funkcije se dobija željeni rezultat. Jezici koji podržavaju ovaj koncept nude i sintaksu za lakše pisanje i čitanje koda.

Na listingu 3.5.1 se nalazi primjer funkcije napisan u *Javascript*-u upotrebom *arrow function* notacije.

```
var makeAdd = x => y => x + y;
var addFive = makeAdd(5);
var result = addFive(3);
```

Listing 3.5.1 Primjer *curring*-a

4. RAZLIKE U ODNOSU NA DRUGE PROGRAMSKE PARADIGME

4.1 Funkcionalno i objektno-orijentisano programiranje

Objektno-orijentisano programiranje (OOP) je programska paradigma, zasnovana na konceptu objekta kao gradivne jedinice aplikacije, koja omogućava definiciju klase kao šablona (*template*-a) za kreiranje i ponašanje objekata.

Klasa definiše podatke (atribute) koji predstavljaju stanje i metode (funkcije) koje predstavljaju ponašanje objekta kao i konstruktore koji kreiraju objekat date klase. Izvršavanje OOP aplikacije se svodi na kreiranje objekata i njihovu međusobnu interakciju odnosno mijenjanje stanja objekata. Tabela 4.1.1 predstavlja poređenje funkcionalnog i objektno-orijentisanog programiranja.

Tabela 4.1.1 *Poređenje funkcionalnog i objektno-orijentisanog programiranja*

Funkcionalno programiranje	**Objektno-orijentisano programiranje**
Koristi *immutable* podatke (zahtjeva više programske memorije)	Koristi *mutable* podatke (štedi memoriju)
Deklarativan pristup	Imperativan pristup
Izuzetno pogodno za paralelno izvršavanje	Nije pogodno za paralelno izvršavanje
Nema *side* efekte	Veoma prisutni *side* efekti
Redoslijed operacija nije toliko bitan	Redoslijed operacija je izuzetno bitan
Koristi rekurziju	Koristi petlje

4.2 Funkcionalno i proceduralno programiranje

Proceduralno programiranje (često se naziva i imperativno iako predstavlja njegovu podgrupu) posmatra program kao niz naredbi koji opisuje šta se treba uraditi. Zasnovano je na procedurama (*routines*) koje definišu instrukcije za pojedinačne korake čijim izvršavanjem se postiže željeni cilj. Prednost ovakvog pristupa je bliskost sa čovjekovim razmišljanjem pri rješavanju problema i sa arhitekturom računara što za posljedicu ima dobre performanse izvršavanja.

Mane proceduralnog programiranja u odnosu na funkcionalno su što je redoslijed izvršavanja procedura veoma značajan jer rezultat procedura direktno zavisi od trenutnog stanja programa. Veoma su prisutni *side* efekti i *mutable* podaci.

5. PROGRAMSKI JEZIK *F#*

F# [5] se pojavio u svijetu programskih jezika 2005. godine. Razvijen je od strane Majkrosofta [6] kao *strongly-typed* jezik koji podržava više programskih paradigmi. Primarno je funkcionalni, ali omogućava i imperativni i objektno-orijentisani stil programiranja. *F#* je *cross-platform* jezik te se može izvršavati na *Windows*-u, *Linux*-u ili *Mac OS*-u. Zasnovan je na *.NET* platformi tako da se lako može integrisati sa *C#*-om ili *Viusal Basic*-om i može koristiti sve *.NET* biblioteke.

Razvojna okruženja *Visual Studio* i *Xamarin* nude potpunu podršku za rad sa *F#*-om. Razvoj *F#* aplikacija moguć je još i u *Mono*, *MonoDeveloper* i *WebSharper* razvojnim okruženjima ili pomoću *Ionide* proširenja u *Atom*-u i *Visual Studio Code*-u.

F# nudi sintaksna pojednostavljenja u odnosu na *C#* i *Java* jezike. Izraze nije potrebno završavati sa tačka-zarezom (;). Kreiranje bloka izraza se ne postiže vitičastim zagradama, {}, nego indentacijom (uvlačenjem) izraza, te se izrazi u istom indentacionom nivou nalaze u istom bloku. Jednolinijski komentari počinju sa dvostrukim *slash* (/) karakterom, dok višelinijski komentari počinju sekvencom (*, a završavaju se sa *).

Podržava cjelobrojne tipove podataka, *signed* i *unsigned integer*, sa veličinom od 8, 16, 32 i 64 bita, kao i *bigInt* tip kojim se može predstaviti bilo koji cijeli broj. Od tipova podataka sa pokretnim zarezom podržani su 32-bitni i 64-bitni *float*, 128-bitni *decimal* tipovi. *Char* i *string* predstavljaju dostupne tekstualne tipove podataka, dok *bool* predstavlja logički tip. *F#* podržava osnovne binarne i unarne aritmetičke operatore primjenljive na *integer* i *float* tipove, zatim operatore poređenja, logičke i *bitwise* operatore.

Varijabla se definiše ključnom riječju *let* i = operatorom za dodjelu vrijednosti i podrazumijevano je da je *immutable*. Varijabla se može definisati i kao *mutable* sa *let mutable*, da bi joj se vrijednost mogla promijeniti korišćenjem ‹- operatora. Tip se može navesti nakon naziva praćenog dvotačkom (:). Ukoliko se tip ne navede,

F# kompajler će na osnovu dodjeljene vrijednosti dodijeliti tip varijabli.

F# nudi *if/elif/else* strukture za odlučivanje koje vraćaju vrijednost (suprotno *if*-u u *C#* i *Java* jeziku). Svaka grana vraća vrijednost svog posljednjeg izraza, te je neophodno da sve grane vraćaju vrijednosti istog tipa. Grana može da ne vrati vrijednost, odnosno da implicitno vrati *unit*, slično *void*-u u *C#*-u.

For...to i *for...downto* petlje omogućavaju eksplicitno navođenje broja iteracija. Pomoću *for...in* se može prolaziti kroz kolekcije. U *for* petljama nije moguće preskakati iteracije (*continue*) ili nasilno prekidati petlju (*break*). Podržana je još i *while...do* petlja.

Funkcija se definiše upotrebom ključne riječi *let* i navođenjem naziva i liste parametara bez tipa razdvojenih razmacima i tijela funkcije kao na listingu 5.1. Opciono, par naziv parametra : tip se može staviti u ( ) kako bi se specificirao tip parametra. Specificiranje tipa povratne vrijednosti (*return-type*) nakon dvotačke je opciono, jer kompajler može da odredi tip povratne vrijednosti na osnovu tipa vrijednost posljednjeg izraza u tijelu funkcije.

```
let function-name parameter-list [ :
return-type ] = function-body
```

Listing 5.1 Definicija funkcije

Rekurzivna funkcija se definiše korišćenjem *let rec* ključnih riječi. Operatori >> (*forward composition*) i << (*backward composition*) omogućavaju kompoziciju dvije funkcije u novu funkciju. Ulazni parametar naredne funkcije u kompoziciji mora po tipu odgovarati povratnoj vrijednosti prethodne funkcije. *Pipelining* (|> - *forward pipeline*, <| - *backward pipeline*) omogućava ulančavanje više funkcija kao uzastopnih operacija nad datom vrijednošću.

F# ima ugrađeni *option* tip za označavanje vrijednosti varijable koja može da postoji, a može i da ne postoji i ima dvije moguće vrijednosti *Some(x)* i *None*. *Option* nudi alternativu *null*-u u *C#-u* ili *Java* jeziku.

Dostupne strukture podataka su *tuple*, lista, niz, sekvenca, set, mapa, *record*.

F# podržava *discriminated* unije za definisanje skupa mogućih slučajeva u kojima varijabla može postojati. Korisne su za situaciju kada se jedan podatak može javiti u više oblika odnosno tipova. Slično ponašanje bi se u objektno-orijentisanom programiranju moglo postići definicijom bazne klase i više izvedenih što je znatno komplikovanije. Nedostatak *switch-case* strukture je nadoknađen *pattern matching*-om koji pored poređenja podataka omogućava vraćanje rezultata i dekompoziciju ulaznih podataka. Naročito je pogodan za korišćenje sa *discriminated* unijama.

F# u potpunosti omogućava objektno-orijentisano programiranje, te isti podržava klase, mehanizam nasljeđivanja i *interface*-e. Jezik još omogućava generičnost (*generics*), rad sa delegatima, mehanizme za rukovanje izuzecima, kao i ogranizaciju koda u module i *namespace*-eve.

F# je jezik opšte namjene. Može se koristiti za razvoj *Windows desktop* i *console*-nih aplikacija, za pisanje skript (.fsx fajlovi). Postoji nekoliko *framework*-a (*Suave, ASP .NET Core, Fable, WebSharper, Girrafe, Freya*) za razvoj veb aplikacija pomoću *F#*-a. Pogodan je za metematičke aplikacije jer se matamatičko razmišljanje može lako preslikati u *F#* kod.

6. ZAKLJUČAK

Funkcionalno programiranje predstavlja sasvim drugačiji pogled na organizaciju i izvršavanje programskog koda u odnosu na imperativni pristup.Većini programera imperativna odnosno objektno-orijentisana paradigma je osnovna ili jedina poznata prije svega zbog veće zastupljenosti OOP na računarskim i tehničkim fakultetima.

Razlog ovoga se može potražiti i u manjoj primjeni funkcionalne paradigme u razvoju modernih softverskih aplikacija prije svega zbog slabijih performansi funckionalnih jezika. Primjenom *mutability* koncepta i eliminacijom *side* efekata, funkcionalno programiranje omogućava pisanje koda sa manje *bug*-ova, koda u kojem se lakše i bezbjednije mogu praviti izmjene.

Takođe, testiranje i paralelizacija izvršavanja funkcionalnog koda je mnogo lakša. Zbog ovih prednosti, za očekivati je sve veću prisutnost funkcionalnog programiranja kako u formalnom računarskom obrazovanju tako i u razvoju komercijalnih softverskih rješenja.

7. LITERATURA

[1] Procedural programming, https://en.wikipedia.org/wiki/Procedural_programming

[2] Object-oriented programming, https://en.wikipedia.org/wiki/Object-oriented_programming

[3] Functional programming, https://en.wikipedia.org/wiki/Functional_programming

[4] Lambda calculus, https://en.wikipedia.org/wiki/Lambda_calculus

[5] FSharp, https://fsharp.org/

[6] Microsoft, https://www.microsoft.com

Kratka biografija:

Jovica Zarić rođen je 03.06.1992. godine u Bijeljini. Osnovnu školu „Jovan Dučić“ u Bijeljini završio je 2007. godine. U istom gradu je završio opšti smjer Gimnazije „Filip Višnjić“ 2011. godine. Zvanje diplomirani inženjer elektrotehnike i računarstva stekao je 2015. godine na Fakultetu tehničkih nauka u Novom Sadu.