FUNKCIONALNO PROGRAMIRANJE U JEZIKU JavaScript

PREDGOVOR

Funkcionalno programiranje (FP) je stil programiranja sa dugom istorijom i sve intenzivnijom primenom u poslednje vreme.

Postojeći literaturni izvori funkcionalno programiranje prikazuju najčešće na sledeće načine:

- Kroz programske jezike koji dosledno implementiraju koncepte funkcionalnog programiranja i zovu se čisti funkcionalni jezici.
- Kroz široko primenjivane programske jezike koji ne spadaju u grupu čistih funkcionalnih jezika, ali u određenoj meri podržavaju koncepte i tehnike funkcionalnog programiranja.
- Kroz matematički instrumentarijum koji je teorijska osnova funkcionalnog programiranja i obuhvata dve glavne oblasti, Lambda račun i Teoriju kategorija.

Autori ovog rukopisa su se opredelili da kombinuju ove pristupe na način koji će čitaocu pomoći da ovlada funkcionalnim stilom programiranja tako što će: (1) u potpunosti da razume ideju i osnovne koncepte funkcionalnog programiranja i da spozna njegove prednosti i nedostatke pri razvoju tipičnih savremenih softverskih aplikacija, (2) da ovlada programskim jezikom JavaScript koji ima široku praktičnu primenu i podržava osnovne zahteve paradigme funkcionalnog programiranja i da nauči da ga koristi primenjujući funkcionalnu paradigmu, i (3) da se upozna sa matematičkim fundamentima funkcionalnog programiranja i prednostima koje proizilaze iz njegove stroge matematičke zasnovanosti.

Rukopis je organizovan u četiri dela.

Prvi deo nosi naslov O RAZVOJU SOFTVERA I PROGRAMIRANJU i sastoji se od dva poglavlja. Prvo poglavlje bavi se ulogom kompozicije i apstrakcije u razvoju softvera, a drugo poglavlje stilovima programiranja sa naglaskom na funkcionalnom stilu i njegovim osnovnim obeležjima.

Drugi deo rukopisa ima naslov OSNOVE PROGRAMSKOG JEZIKA JAVASCRIPT. Sastoji se od pet poglavlja. Prvo poglavlje izlaže osnovne konstrukte jezika sa naglaskom na sintaksi. Drugo poglavlje bavi se neprimitivnim tipom `object`. Teće poglavlje sadrži osnovno o funkcijama u jeziku JavaScript. Četvrto pglavlje bavi se asinhronim programiranjem u Jeziku JavaScript. Poslednje, peto poglavlje daje sažet prikaz JavaScript okruženja.

Treći i četvrti deo rukopisa posvećeni su funkcionalnom programiranju pri čemu se jezik JavaScript koristi kao tehnološka platforma

Treći deo koji nosi naslov FUNKCIONALNO PROGRAMIRANJE U JEZIKU JAVASCRIPT je centralni deo rukopisa. Sastoji se od tri poglavlja. Prvo od njih bavi se motivacijom za korišćenje funkcionalnog programiranja i izlaže osnovne koncepte i principe ove programske paradigme. Drugo poglavlje objašnjava ulogu i način korišćenja funkcje u funkcionalnom programiranju. Treće poglavlje posvećeno je rekurziji.

Četvrti deo rukopisa ima naslov FUNDAMENTI FUNKCIONALNOG PROGRAMIRANJA. U njemu su sažeto izložena dva teorijska fundamenta funkcionalnog programiranja: Lambda račun i teorija kategorija. Pored toga, izložena su i tri karakteristična primera primene teorije kategorija u funkcionalnom programiranju: funktori, aplikativni funktori i monade.

DEO I: O RAZVOJU SOFTVERA I PROGRAMIRANJU

Poglavlje 1: RAZVOJ SOFTVERA JE KOMPOZICIJA I APSTRAKCIJA

Započećemo ovu knjigu navođenjem stanovišta da su računarski programi namenjeni, pre svega, za čitanje ljudima, a tek onda za izvršavanje računarima. Posledica ovakvog stanovišta je potreba za programskim jezicima koji imaju izražajne moći i sintaksu pogodne za stil programiranja koji je blizak načinu na koji ljudi rešavaju probleme. U malo različitim formulacijama iskazali su ga različiti autori.

Fišer Blek (FischerBlack) je pojam stila programiranja artikulisao još 1964. godine u svom radu "Styles of Programming in LISP" u zborniku radova "The Programming Language LISP: Its Operations and Applications", urednika E. Berkeley i D. Bobrow koji je dostupan na adresi https://www.softwarepreservation.org/projects/LISP/book/III_LispBook_Apr66.pdf.

Drugi je Harold Abelson[1] jedan od autora čuvene knjige "Structure and Interpretation of Computer Programs" iz 1996, kome se pripisuje izjava: **"Programi moraju biti napisani da bi ih ljudi mogli čitati, a samo slučajno da bi ih izvršile mašine."**. Gotovo ista formulacija pripisuje e i Donaldu Ervinu Knutu (Donald Erwin Knuth)[2]: **"Programi su namenjeni da ih čitaju ljudi i sasvim slučajno da ih izvršavaju računari. "**

Krajnji rezultat razvoja softvera je računarski program koji će (još uvek) čitati ljudi i izvršavati računari.

Kada se govori o razvoju softvera u današnjim uslovima postoje divergencije u stavovima i mišljenjima po velikom broju pitanja, od kojih su neka čak i fundamentalnog značaja. I to nije ništa neobično - razvoj softvera je komplikovan posao i nerealno je očekivati potpunu saglasnost po pitanjima njegovog razvoja. Ipak, oduvek postoji opšta saglasnost o dve stvari: (1) Razvoj softvera predstavlja razlaganje složenih problema na jednostavnije koji se pojedinačno rešavaju i, zatim, međusobno povezuju - komponuju - u celovito rešenje; (2) Razvoj softvera podrazumeva primenu apstrakcije da bi se moglo izvršiti prethodno pomenuto razlaganje, formulisati rešenja za jednostavnije probleme, i ta rešenja komponovati u funkcionalnu celinu .

U ovom Poglavlju bavićemo se kompozicijom i apstrakcijom u razvoju softvera.

1.1. Komponovanje je suština razvoja softvera

Termin komponovanje, kompozicija koristi se da označi nešto sa čime se mi stalno susrećemo u svakodnevnom životu. U suštini, gotovo sve što radimo je komponovanje, svaka složenija aktivnost sastoji se od pojedinačnih akcija „složenih“ na odgovarajući način.

Ako želite da pojedete sendvič, morate ga „sklopiti“ od više pojedinačnih delova: hleb, namaz (ako to volite, a i ne morate), malo nekog mesa (ako niste vegetarijanac), malo povrća (ako niste isključivi „mesožder“). A ako želite da napišete čak i najjednostavniji računarski program, morate da ga „sklopiti“ od više pojedinačnih naredbi nekog programskog jezika.

[1] Harold Abelson (1947), američki naučnik I profesor iz oblasti računarstva (https://en.wikipedia.org/wiki/Hal_Abelson)

[2] Donald Ervin Knuth (1938), američki naučnik iz oblasti računarstva I matematičar, (https://en.wikipedia.org/wiki/Donald_Knuth)

1.1.1. Komponovanje softvera

Proces razvoja softvera je razlaganje velikih problema na manje probleme, izrada komponenti koje te manje probleme rešavaju i, zatim, komponovanje tih komponenti u celovitu aplikaciju.

Dakle, svaki razvoj softvera podrazumeva definisanje nekih komponenti (funkcije i strukture podataka, jer softver samo izvršava neke funkcije nad nekim podacima) i njihovo komponovanje u celinu koja obezbeđuje nameravanu funkcionalnost.

U jednom svom predavanju Rúnar Bjarnason, jedan od autora vrlo popularne knjige o funkcionalnom programiranju [1] kaže sledeće.

"Modularnost je svojstvo da se Vaš softver može razdvojiti u svoje sastavne delove i da se ti delovi mogu ponovo koristiti nezavisno od celine, na nove načine koje niste predvideli kada ste te delove kreirali."

"Kompozicionalnost je odlika da Vaš softver može da se razume kao celina kroz razumevanje njegovih delova i pravila koja upravljaju kompozicijom."

Pri razvoju softvera kompozicija se ne može izbeći i ona uvek postoji, čak i kada je oni koji softver razvijaju nisu svesni. Bez obzira na programski jezik i paradigmu, ne može se pobeći od komponovanja funkcija i struktura podataka, jer se na kraju sve svede na to. Kompozicija može samo da se radi nesvesno i da loš rezultat bude gotovo izvestan, ili svesno pa da verovatnoća dobrog rezultata bude značajno uvećana.

Dobra kompozicija je najvažnija odlika dobrog softvera.

Oni koji softver razvijaju moraju da razumeju kompoziciju, jer je kompozicija jedini način da se napravi dobar softver.

1.1.2. Mehanizmi za komponovanje softvera

Kompozicija softvera nastoji da obezbedi mehanizme za sistematičnu konstrukciju baziranu na dobro definisanim jedinicama softvera. U literaturi su predlagani različiti mehanizmi za različite vrste jedinica softvera. U radu [2] autora Kung-Kiu Lau i Tauseef Rana dat je dobar pregled glavnih mehanizama za kompoziciju softvera klasifikovanih u tri grupe prema pogledima na kompoziciju softvera. Ta tri pogleda su: programerski, konstrukcioni, i komponentno-baziran razvoj.

Programerski pogled smatra svaki legitimni konstrukt programskog jezika jedinicom kompozicije, a sama kompozicija je prosto združivanje tih konstrukata u druge konstrukte definisane u programskom jeziku. Smislene jedinice kompozicije u programerskom pogledu su, na primer, funkcije u funkcionalnim jezicima, procedure u imperativnim jezicima, klase u objektnim jezicima, aspekti u aspektno-orijentisanim jezicima, itd.

Konstrukcioni pogled je pogled u kome se jedinice kompozicije nazivaju komponentama, pri čemu su komponente labavo definisane jedinice softvera sa priključcima koji predstavljaju tačke interakcije ili povezivanja. Shodno tome, komponente mogu da budu proizvoljne jedinice softvera koje se mogu povezati nekim mehanizmima kao što su moduli povezani jezikom interakcije modula, ili Java Bean-ovi povezani korišćenjem jezika Piccola, itd.

Komponentno-bazirani razvoj je pogled u kome su jedinice kompozicije precizno definisane putem komponentnog modela. Komponentni model definiše šta su komponente (njihovu sintaksu i semantiku) i koji se operatori kompozicije mogu koristiti za njihovo komponovanje. Jedna definicija softverske komponente je: “element softvera koji je saglasan sa komponentnim modelom i može se nezavisno razvijati i komponovati bez modifikacije u skladu sa standardima kompozicije”. Svoju

primenu je ovaj pristup našao u servisno-orijentisanim arhitekturama softvera poput web servisa i operatora koreografije i orkestracije.

Isti izvor mehanizme kompozicije iz sve tri navedene kategorije klasifikuje u četiri opšte kategorije: (i) sadržavanje (eng. containment); (ii) proširivanje (engl. extension); (iii) povezivanje (engl. connection); i (iv) koordinacija (engl. coordination).

Za nas je od interesa pristup koji komponente posmatra kao (matematičke) funkcije koje primaju ulazne vrednosti i vraćaju izlazne vrednosti, što odgovara mehanizmu funkcija višeg reda iz kategorije koordinacije. U čistom funkcionalnom programiranju ovo je inherentni mehanizam komponovanja.

1.2. Apstrakcija i softver

Započećemo ovaj odeljak sledećim tekstom.

Ljudska vrsta i svi *produkti njene delatnosti* odlikuju se jednom karakteristikom: ograničenim mogućnostima. Mi smo, kao vrsta, ograničeni svojim psiho-fizičkim odlikama, a veštački artifakti koje proizvodimo takođe su karakterisani različitim ograničenjima po vrsti i obimu. Za *temu kojom se bavimo* u ovoj knjizi - razvoj softvera - *od primarnog značaja su naša ograničenja koja se odnose na sposobnosti manipulisanja podacima, informacijama i znanjem* sa krajnjim ciljem rešavanja nekog zadatka u kome učestvuju pripadnici ljudske vrste i jedan proizvod delatnosti ljudske vrste - računar. Krajnji *cilj je računarski program koji računar može da izvrši* i čijim se izvršavanjem ostvaruje rezultat koji je opservabilan za čoveka ili/i neki drugi veštački artifakt koji je proizvela ljudska vrsta.

Šta smo mi to, u stvari, uradili u tekstu prethodnog paragrafa? Prvo smo odredili ***temu kojom ćemo da se bavimo*** - razvoj softvera. Zatim smo izdvojili, kao predmete našeg interesovanja, ***pripadnike ljudske vrste*** i jedan produkt njene delatnosti - ***računar*** kao nešto što nam je relevantno u kontekstu a nismo se bavili zmijama, bakterijama, lavovima, niti biciklima, tanjirima, stolicama ili cipelama. Takođe, nisu nam bila relevantna ograničenja naše vrste koja se odnose na brzinu kretanja ili muzičku nadarenost, već su nam bila relevantna ograničenja koja imamo u pogledu sposobnosti da manipulišemo podacima, informacijama i znanjem.

Postupak kojim smo generisali tekst iz prvog paragrafa ovoga poglavlja je tipičan primer ***apstrakcije***. Kako kaže Džon Gutag (John Vogel Guttag)[3]: "Suština apstrakcija je očuvanje informacija koje su relevantne u zadatom kontekstu i zaboravljanje informacija koje su u tom kontekstu nerelevantne."

Mi smo, dakle, ***odredili kontekst*** - ***razvoj softvera***, i ***očuvali smo informacije koje smo smatrali relevantnim u zadatom kontekstu*** - naše sposobnosti da manipulišemo podacima, informacijama i znanjem i računar kao uređaj. ***Nerelevantne informacije*** o našoj sposobnosti da brzo trčimo ili da baš ne ne umemo da pevamo, informacije o biciklima, cipelama, tanjirima i stolicama ***smo izostavili***.

1.2.1. Softver je apstrakcija

Kada se radi o razvoju softvera, apstrakcija ima posebnu težinu jer se, sa puno prava, može kazati da je sam softver, u stvari, apstrakcija.

Kao prvo, softver je nematerijalna stvar pa je time apstrahovan od naše mogućnosti da ga svojim čulima direktno percipiramo.

[3] https://en.wikipedia.org/wiki/John_Guttag

Kao drugo, softver je uvek namenjen rešavanju klase problema, što podrazumeva određivanje klase problema, izdvajanje i očuvavanje relevantnih, odnosno odbacivanje nerelevantnih informacija za tu klasu.

U softverskom inženjerstvu se apstrakcija posmatra kao[4]:

- Uklanjanje fizičkih, prostornih ili vremenskih detalja ili atributa u proučavanju objekata ili sistema da bi se pažnja usmerila na detalje veće važnosti – **proces apstrakcije**;
- **Apstraktni koncepti-objekti** koji odslikavaju zajednička svojstava ili atributa različitih ne-apstraktnih objekata ili sistema - ***rezultat apstrakcije***.

Proces apstrakcije ima dve glavne komponente:

- **Generalizacija** je pronalaženje ***sličnosti*** (očiglednih) koje se ponovljaju u obrascima i sakrivanje sličnost iza apstrakcije.
- **Specijalizacija** je izdvajanje *onoga* ***što je različito*** (smisleno različito) u svakom od slučajeva koji poseduju međusobne sličnosti.

Zanimljivo razmišljanje o terminu apstrakcija u kontekstu softvera dato je u izvoru [1]. Autor Erik Eliot ovde se bavi terminom *apstrakcija* iz aspekta koji se može smatrati etimološkim, poredeći reči koje odgovaraju pojmu apstrakcije u dva jezika - Latinskom i Jidišu. U tom razmišljanju, on iznosi zapažanje da i u jednom i u drugom jeziku termin možemo povezati na neki način sa *udaljavanjem ili otuđivanjem od konkretnog*. Zanimljiv je zaključak koji autor izvodi, najvećim delom na bazi značenja koji on nalazi u jeziku Jidiš. Naime, Erik Eliot, prevodeći reč "abstraction" sa engleskog na Jidiš i obrnuto, dobija kao rezultat englesku reč "absentminded" koja se u rečnicima prevodi na srpski jezik kao "rasejan" a koju bih ja u ovoj situaciji na srpski jezik prevela kao "misaono odsutan". Eliotu se to dopada jer on smatra da *misaono odsutna* osoba "vozi na autopilotu", bez aktivnog razmišljanja o onome što radi, prosto to radi. I apstrakcija nam, po njemu, omogućuje da bezbedno vozimo "na autopilotu". Po njemu, softver je automatizacija koja nam omogućuje da mnogo brže i bez razmišljanja o postupku koji sprovodimo (u toku sprovođenja postupka) uradimo ono što bismo, suštinski, mogli da uradimo i bez korišćenja softvera. Softver je, dakle, APSTRAKCIJA KOJA SAKRIVA SVE DETALJE TEŠKOG RADA uz istovremeno ostvarivanje dobrobiti.

Pored toga, ceo naš svakodnevni život obiluje ponavljanjima određenih postupaka i akcija što se pojavljuje i u softveru. Mi ne učimo da hodamo svaki put kada nam se pojavi potreba za hodanjem. Naučimo to jednom i ponavljamo u svakoj situaciji kada nam je to potrebno. To vodi do jednog od osnovnih principa arhitekture softvera - PONOVNE UPOTREBLJIVOSTI (engl. reusability): Softverska rešenja treba da budu takva da se MOGU RAZLOŽITI NA KOMPONENTNE DELOVE I REKOMPONOVATI U NOVA REŠENJA BEZ PROMENE INTERNIH DETALJA IMPLEMENTACIJE KOMPONENTI.

Konačno, apstrakcija može da DOPRINESE I POJEDNOSTAVLJENJU PROBLEMA tako što će se problem sagledati kroz proces izdvajanja suštine i koncepata. Kada sagledamo suštinu problema, često možemo da zaključimo da već postoje rešenja suštine problema koja mogu da se primenu i na naš konkretan problem, a najčešće i mnoge druge konkretne probleme. Uostalom, mi veliki broj problema (ako imamo sreće) možemo da svedemo na rezultate iz matematike. U oblasti softvera ističe se i pristup koji se zove modelom-vođen razvoj softvera koji se u velikoj meri oslanja na apstrakciju. To se, naravno, kasnije proširuje i na kodiranje čime se značajno smanjuje potreba za izradom novog koda.

1.2.2. Mehanizmi apstrakcije u softveru

Pored koncepta algoritma kao osnovne apstrakcije u razvoju softvera, glavni oblik apstrakcije u računarstvu, pa i razvoju softvera, je jezička apstrakcija. U oblasti su prisutni različiti jezici poput

[4] https://en.wikipedia.org/wiki/Abstraction_(computer_science)

jezika za modelovanje i programskih jezika različitih nivoa, a (programski) jezici omogućuju krieranje novih apstrakcija poput tipova podataka, funkcija, modula, itd. Među najvažnijima mehnizmima apstrakcije softvera su:

- Jezici;
- Tipovi podataka i strukture podataka;
- Procedure, funkcije;
- Moduli;
- Klase;
- Interfejsi;
- Razvojna okruženja;
- .....

David Šmit (David A. Schmidt) iskazuje princip apstrakcije kao zahtev programskog jezika u formulaciji iz knjige "The structureoftypedprogramminglanguages" [3] na sledeći način: *"Fraza bilo koje semantički smislene semantičke klase treba da bude imenovana."*

Kao preporuku programerima, Bendžamin Pirs (Benjamin C. Pierce) u knjizi " Types and Programming Languages" [4], iskazuje princip apstrakcije na sledeći način: *"Svaki značajan deo funkcionalnosti u programu trebao bi da bude implementiran samo na jednom mestu u izvornom kodu. Kada se slične funkcije izvršavaju različitim delovima koda, generalno je delotvorno kombinovati ih (delove koda) u jedno tako što će se promenljivi delovi apstrahovati."*

Funkcije su izuzetan mehanizam za apstrakcije zbog toga što poseduju svojstva koja su od suštinskog značaja za dobru apstrakciju:

- ***Identitet*** - sposobnost dodele imena i ponovnog korišćenja u različitim kontekstima, što odgovara Šmitovoj formulaciji principa apstrakcije.
- ***Kompozibilnost*** - sposobnost komponovanja jednostavnih funkcija u složenije funkcije, što odgovara Pirsovoj formulaciji principa apstrakcije.

Zbog toga što je predmet našeg interesovanja funkcionalno programiranje, mislim da je prikladno ovde citirati Džona Karmaka (John Carmack[5]) koji kaže: "Ponekad, elegantna implementacija je samo funkcija. Ne metoda. Ne klasa. Ne okruženje. Samo funkcija."

Konačno, kada se govori o apstrakciji softvera, valjalo bi razjasniti odnos apstrakcije i životnog ciklusa softvera.

Stav autora je da je sam ŽIVOTNI CIKLUS SOFTVERA APSTRAKCIJA NAJVIŠEG NIVOA u oblasti softvera, utoliko što se ona odnosi na svaki softver u svakoj fazi njegovog postojanja/nepostojanja, bez obzira na njegovu namenu, način specifikacije i implementacije, ili neki drugi aspekt posmatranja.

Dalje, svaka faza u životnom ciklusu softvera je apstrakcija softvera u toj, određenoj fazi koja se izgrađuje uzimajući u obzir aspekte softvera relevantne za tu fazu. Na primer, za kreiranje zahteva softvera u obzir se uzimaju zahtevi pojedinačnih (klasa) korisnika, iz njih se izvode zajednički elementi a specijalizuju se zahtevi koji se razlikuju od korisnika do korisnika. U fazi kodiranja, piše se kod komponenti koje će biti upotrebljive u različitim aplikacijama tako što se obezbeđuju interfejsi za kombinovanje komponenti.

Realizacija faza životnog ciklusa softvera podrazumeva postojanje odgovarajućih mehanizama za apstrakciju u softveru. Ovih mehanizama ima, naravno, mnogo i oni nisu međusobno isključivi niti

[5]**John D. Carmack II** (1970) je američki programer, posebno poznat po rezultatima iz oblasti video igara (https://en.wikipedia.org/wiki/John_Carmack).

striktno primenljivi na samo određenu fazu. U jednoj fazi primenjuje se više mehanizama apstrakcije a ima i nekih mehanizama koji se mogu primenjivati i u više faza.

1.3. Sažetak

U ovom poglavlju razmotrili smo ulogu kompozicije i apstrakcije u razvoju softvera. Objašnjeni su termini kompozicija i apstrakcija i, posebno, njihovo značenje i značaj u razvoju softvera.

Proces razvoja softvera je razlaganje velikih problema na manje probleme, izrada komponenti koje te manje probleme rešavaju i, zatim, komponovanje tih komponenti u celovitu aplikaciju. Dobra kompozicija je najvažnija odlika dobrog softvera. Postoje različiti mehanizmi za kompoziciju softvera među kojima se posebno ističe kompozicija funkcija.

Apstrakcija u softverskom inženjerstvu sastoji se iz ***procesa apstrakcije*** i ***apstraktnih koncepata***. Pored koncepta ***algoritma*** kao osnovne apstrakcije u razvoju softvera, glavni oblik apstrakcije u računarstvu, pa i razvoju softvera, je ***jezička apstrakcija***.

Literatura uz Poglavlje 1

[1] Paul Chiusano, Rúnar Bjarnason, Functional Programming in Scala, ManningPublicationsCo, 2015.

[2] K. -K. Lau and T. Rana, "A Taxonomy of Software Composition Mechanisms," 2010 36th EUROMICRO Conference on Software Engineering and AdvancedApplications, Lille, France, 2010, pp. 102-110, doi: 10.1109/SEAA.2010.36.

[3] David A. Schmidt, The structure of typed programming languages, MIT Press, 1994.

[4] Benjamin C. Pierce, Types and Programming Languages, MIT Press, 2002

Poglavlje 2: PARADIGME PROGRAMIRANJA I OSNOVNO O OBJEKTNOM PROGRAMIRANJU

Ovo poglavlje ima za cilj da čitaocu pruži kratak prikaz istorije, koncepata i korišćenja programske paradigme (stila programiranja) koja se naziva *funkcionalno programiranje*.

U njemu će, prvo, biti objašnjeno značenje pojma paradigma programiranja, navedene i vrlo sažeto opisane osnovne paradigme programiranja.

Zatim će biti vrlo sažeto dato izlaganje o funkcionalnom programiranju sa ciljem da čitaoca uvede u materiju koja će biti izložena u poglavljima koja slede. Obuhvaćeni su kratak prikaz istorije i objašnjenje suštine funkcionalnog programiranja i navedeni su najvažniji koncepti funkcionalnog programiranja sa kratkim objašnjenjima njihovog značenja.

2.1. Paradigme programiranja

Programski jezik sastoji se od *sintakse* i *modela izvršavanja* (engl. *execution model*).

U računarskoj nauci, sintaksa programskog jezika su pravila koja definišu kombinacije simbola za koje se smatra da su ispravno strukturirani iskazi ili izrazi na tom jeziku, elementi tog jezika. Dakle, sintaksa jezika definiše pravila koja se moraju poštovati pri izražavanju putem programskog jezika baš kao što u prirodnim jezicima postoje reči, fraze, rečenice, interpunkcijski znakovi i alfabet (pismo) koje koristimo da ono što želimo (i možemo) izrazimo tim jezikom.

Svaki programski jezik ima model izvršavanja koji određuje način na koji se jedinice rada (elementi jezika koji su naznačeni sintaksom programa) rasoporeduju na izvršenje. Model izvršavanja specificira ponašanje elemenata jezika. Primenom modela izvršavanja može se izvesti ponašanje programa pisanog u odgovarajućem programskom jeziku. Kada čovek čita kod pisan u nekom programskom jeziku, on u svojoj glavi primenjuje model izvršavanja i na taj način razumeva šta će da se desi izvršavanjem tog koda: kada se naiđe na naredbu dodele, zna se da će vrednost sa desne strane znaka jednakosti biti dodeljena varijabli sa leve strane znaka jednakosti; kada se nađe na do `while`, zna se da će se kod između linije u kojoj je ključna reč do i linije u kojoj je ključna reč `while` izvršavati više puta, itd. Ponašanje programa koji se izvršava mora da odgovara ponašanju izvedenom iz modela izvršavanja - program mora da se izvršava po propisanom modelu. Dakle, model izvršavanja predstavlja opis semantike programskog jezika.

Model izvršavanja obuhvata stvari kao što su nedeljiva jedinica izvršavanja i ograničenja koja opisuju redosled kojim se dešavaju te jedinice izvršavanja. Na primer, u programskom jeziku C jedinice izvršavanja su *naredbe* (engl. *statements*) koje se sintaktički označavaju terminalnim simbolom ";". U mnogim jezicima operacija sabiranja se tretira kao nedeljiva, a u sekvencijalnim jezicima se više operacija sabiranja odigravaju jedna za drugom. Taj redosled može da se bira unapred ili se, u nekim situacijama, može određivati dinamički prema toku izvršavanja. Većina modela izvršavanja dozvoljava oba ova načina u određenoj meri.

Model izvršavanja implementira se putem kompajlera ili interpretera i često uključuje i sistem izvršavanja (engl. *Runtime system*). Kompajler implementira unapred definisan redosled izvršavanja a sistem izvršavanja dinamički redosled izvršavanja. Interpreter koji predstavlja kombinaciju translatora i sistema izvršavanja, omogućuje kombinovanje statičkog i dinamičkog određivanja redosleda.

Modeli izvršavanja mogu da postoje i nezavisno od/izvan programskih jezika, a da se pri tome omogući pozivanje tog eksternog modela izvršavanja korišćenjem regularne sintakse jezika u kome je

program pisan. Karakteristični primeri su modeli izvršavanja u paralelnom računarstvu koji moraju da izlože i svojstva hardvera da bi se ostvarila visoka performansa u računanju. Najpoznatije konkretne implementacije su POSIX Threads biblioteka[6] i Hadoop Map-Reduce model programiranja[7].

Paradigma programiranja je termin koji se koristi da označi klasifikaciju programskih jezika baziranu na svojstvima jezika. Klase koje se najčešće koriste nisu disjunktne, odnosno isti programski jezik može da se klasifikuje u više klasa. Neke pardigme bave se pretežno implikacijama na izvršni model jezika kao što su bočni efekti[8] ili pitanje da li je sekvenca operacija definisana izvršnim modelom. Druge se bave načinom organizacije koda (poput grupisanja koda) u jedinice, zajedno sa stanjem koje taj kod modifikuje. Treće se bave sintaktičkim stilom i gramatikom. Iako postoje izvori koje prepoznaju čak 26 kategorija[9] najvišeg nivoa i na desetine podkategorija, uobičajena podela na najvišem nivou programske jezike i danas klasifikuju u dve paradigme: (1) imperativna i (2) deklarativna. Naravno, moguće je i finije klasifikovanje kroz podklase koje svaka od ovih paradigmi ima pa se imperativni jezici dalje klasifikuju na proceduralne i objektno-orijentisane, a deklarativni jezici se klasifikuju na funkcionalne, logičke i matematičke.

Tema paradigme programiranja je jedno od centralnih pitanja razvoja softvera i kao takva zaslužuje posebnu pažnju. U kontekstu ove knjige koja se bavi funkcionalnim programiranjem, od značaja je da se razume odnos funkcionalnog programiranja prema dvema osnovnim paradigmama, imperativnoj i funkcionalnoj. Stoga ćemo se u nastavku pozabaviti ovim dvema osnovnim paradigmama iz aspekta funkcionalnog programiranja.

Pitanje programske paradigme je vrlo obuhvatno i na način razumljiv programerima objašnjeno u tekstu "Programming Paradigms for Dummies: What Every Programmer Should Know" autora Petera Van Roja (Peter Van Roy) koji je dostupan na adresi https://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf.

2.1.1. Imperativno programiranje

Hardverska implementacija računara koje mi danas koristimo je imperativna. Računarski hardver je dizajniran da izvršava *mašinski kod* koji je nativan za hardver računara (instrukcije koje hardver može da izvrši) i koji je imperativan – mašinski kod tačno specificira SVAKU INSTRUKCIJU KOJU RAČUNAR TREBA DA IZVRŠI I TAČAN REDOSLED IZVRŠAVANJA. Računarski program skladišti podatke (vrednosti) u memorijske lokacije u računaru. Sadržaj tih memorijskih lokacija u datom vremenskim trenutku izvršavanja programa naziva se *stanje programa*. Iz ove perspektive stanje programa je definisano sadržajem memorije, a naredbe u kodu programa su instrukcije nativnog mašinskog jezika računara čijim izvršavanjem se to stanje menja.

Imperativno programiranje je stil programiranja koji opisuje računanje putem naredbi koje **menjaju stanje programa**. Na način koji je veoma sličan izražavanju komandi u imperativnom govoru prirodnog jezika, imperativni program se sastoji od komandi koje računar treba da izvrši. Imperativno programiranje usredsređuje se na opisivanje KAKO program radi.

---

[6]**POSIX Threads** (**pthreads**) je model izvršavanja koji postoji nezavisno od jezika i istovremeno je model paralelnog izvršavanja. Omogućuje programu da upravlja različitim tokovima rada koji se vremenski preklapaju. (https://en.wikipedia.org/wiki/POSIX_Threads)

[7]**MapReduce** je model programiranja i odgovarajuća implementacija za obradu i generisanje skupova velikih podataka (engl. big data sets) sa paralelnim, distribuiranim algoritmom nad klasterom.

[8]U računarskoj nauci se kaže da operacija, funkcija ili izraz ima **bočni efekat** ako modifikuje neke vrednosti varijabli stanja izvan svog lokalnog okruženja, odnosno ako proizvodi neki opservabilan efekat izuzev vraćanja vrednosti onome ko je ispostavio poziv.

[9]https://en.wikipedia.org/wiki/Programming_paradigm

Imperativni programski jezici višeg nivoa koriste koncept *varijable* za skladištenje vrednosti i složenije naredbe ali slede isti stil. I oni imaju naredbe koje operišu nad sadržajem memorije kome se pristupa putem varijabli. Pored toga, omogućuju da se jednom naredbom pozovu delovi programa (internih ili eksternih) putem podprograma i procedura čije izvršavanje menja stanje programa. Upravo ovo svojstvo je osnova za otklon imperativnih programa ka deklarativnim programima što se ogleda kroz podtipove imperativnih programa kao što su proceduralno, modularno i objektno programiranje. Pri tome, deklarativnost se ovde odnosi na pogled programera, a ne na model izvršavanja. Iako je moguće rezonovati o programu samo na osnovu imena, argumenata, i tipova procedura, sam izvorni kod i dalje fiksira redosled izvršavanja naredbi.

Posebno, 80-te godine prošlog veka odlikuju se rapidnim porastom interesovanja za *objektno-orijentisano programiranje*. Jezici ovoga pristupa su po stilu imperativni, uz dodatak podrške za koncept *objekta* i svođenje izvršavanja svakog objektnog računarskog programa na interakciju objekata putem razmene poruka. Razvijen je veliki broj jezika iz ove podgrupe imperativnih jezika (na primer, Simula, Smalltalk-80, C++, Perl, Wolfram jezik, itd.), a trend je dostigao vrhunac u programskom jeziku Java koji je lansiran 1995. godine.

2.1.2. Osnovno o objektnom programiranju

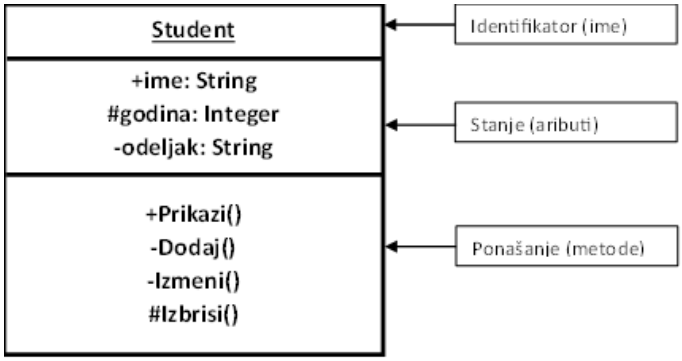
Ovaj odeljak posvetićemo vrlo sažetom izlaganju paradigme objektnog programiranja. Motivacija za ovakvu odluku je što programski jezik JavaScript koji smo mi odabrali za tehnološku osnovu, nije čist funkcionalni jezik, nego je to **objektno-orijentisan jezik sa mogućnostima funkcionalnog programiranja** koji je i u delu podrške funkcionalnom programiranju u velikoj meri oslonjen na koncept objekta. Ovde ćemo izložiti osnovne koncepte objektno-orijentisanog programiranja koji su potrebni za potpunije razumevanje funkcionalnog programiranja u programskom jeziku JavaScript.

Kako je UML (Unified Modeling Language)[10] najšire korišćen jezik za grafičko predstavljanje objektnog softvera, u nastavku ćemo koristit UML notaciju za objašnjavanje objektnog programiranja.

2.1.2.1. Suština objektno-orijentisanog programiranja

Prema izvoru [https://en.wikipedia.org/wiki/Object-oriented_programming#See_also], objektno orijentisano programiranje (OOP) je programska paradigma zasnovana na konceptu ***objekt*** i ***prosleđivanju poruka među objektima***. U objektnom programiranju, objekat je entitet sa ***identitetom*** (jednoznačno ime koje ga razlikuje od drugih objekata) koji se sastoji od ***stanja*** (podaci) i ***ponašanja*** kojima se može uticati na stanje u objektu. Shodno tome, objekat se sastoji od ***atributa*** (nazivaju ih i svojstva) u kojima se skladište podaci/stanja, i ***metoda*** u koje se smešta kod koji implementira ponašanja i može se izvršavati nad sadržajima atributa. Na taj način, objekat u sebi ***enkapsulira*** **(zatvara)** *stanje* i *sve mehanizme kojima se na to stanje može uticati*. UML notacija za objekat prikazana je na slici 2.1.

[10] https://en.wikipedia.org/wiki/Unified_Modeling_Language



Slika 2.1 Reprezentacija objekta u UML notaciji

UML simbol za predstavljanje objekta je pravougaonik podeljen na tri sekcije.

U prvoj sekciji navodi ime objekta koje je obavezno podcrtano (ovde je ime objekta **Student**).

U drugoj sekciji je lista imena atributa - promenljive stanja (ovde su to **ime**, **godina** i **odeljak**).

U trećoj sekciji je lista imena metoda objekta (ovde su to **Prikazi**, **Dodaj**, **Izmeni** i **Izbrisi**).

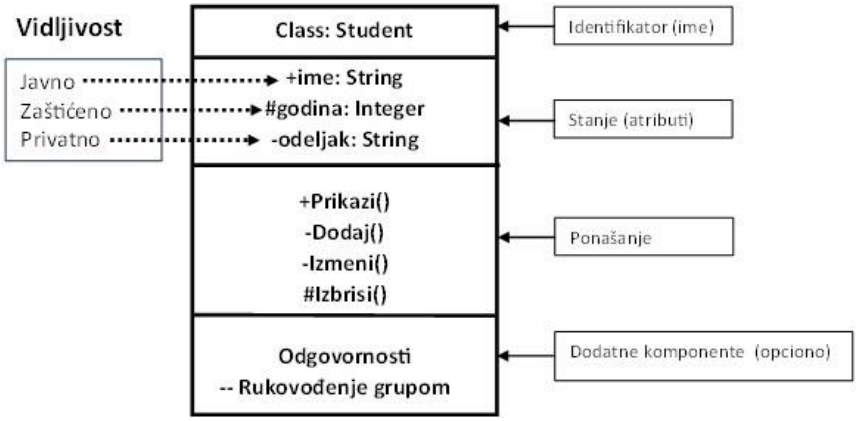
Ispred imena atributa i imena metode navodi se simbol koji **određuje njihovu "vidljivost"**, odnosno definiše **pravo pristupa atributu/metodi od strane klasa i objekata**. Njihovo značenje biće objašnjeno kada budemo govorili o klasi jer je vidljivost u tesnoj sprezi sa klasnom hijerarhijom.

U objektnoj paradigmi programiranja realizacija funkcionalnosti programa ostvaruje se putem prosleđivanja poruka objektima. Jedini način da se utiče na stanje nekog objekta je da se slanjem poruke traži od tog objekta da svoje stanje promeni. Objekat svoje stanje menja izvršavanjem svojih metoda. Dakle, promene stanja se kontrolišu na lokalnom nivou objekta, odnosno stanje nije izloženo nekontrolisanom pristupu.

Objektno-orijentisani programski jezici su raznovrsni, ali je većina najpopularnijih klasno-bazirana što znači da su objekti instance klasa koje određuju tipove objekata. Ovde ćemo ukratko objasniti klasno-bazirano objektno programiranje a kada "dublje uronimo" u JavaScript, govorićemo o drugom modelu objektnog programiranja oslonjenom na koncept ***prototipa objekta***.

Programiranje zasnovano na klasama je stil objektno orijentisanog programiranja (OOP) u kojem je klasa osnovni način apstrakcije. Putem koncepta klase se vrši apstrakcija koja omogućuje da se predstavi grupa objekata (njihova stanja i ponašanja) i njihova struktura - međusobne veze.

Strukturu i ponašanje grupe objekata (određeni tip objekta) **definiše klasa**, odnosno klasa je definicija tipa. Klasa je, jednostavno rečeno, proširiv programski obrazac za kreiranje objekata koji obezbeđuje početne vrednosti za stanje (atributi, varijable stanja) i implementaciju ponašanja (metode, funkcije). UML notacija za klasu prikazana je na slici 2.2.



Slika 2.2 Reprezentacija klase u UML notaciji

U klasno-baziranom pristupu objekat se mora eksplicitno kreirati na osnovu klase i tako kreiran objekat se naziva **instancom te klase**.

Klasa ima svoj identitet – ime. Pošto treba da obezbedi obrazac za kreiranje drugih klasa ili objekata, klasa u sebi sadrži obrasce za ono što će da sadrži neka druga klasa ili objekat: obrazac stanja (atributa) i obrazac ponašanja (metode). Opciono može da sadrži i dodatne komponente. Kao što smo već spomenuli kada smo objašnjavali pojam objekta, ispred imena atributa i imena metode navodi se simbol koji **određuje njihovu "vidljivost"**, odnosno definiše **pravo pristupa atributu/metodi od strane klasa i objekata**. Važe sledeće oznake:

- Simbol plus (**+**) je oznaka za *javno* (engl. Public) i znači da podatku/metodi **mogu da pristupe sve klase, odnosno objekti koji su instance bilo koje klase**.
- Simbol minus (**–**) je oznaka za *privatno* (engl. Private) i znači da podatku/metodi **može da pristupi samo klasa u kojoj je podatak/metoda definisan, odnosno objekti koji su instance te klase**.
- Konačno, simbol heš (**#**) je oznaka za *zaštićeno* i znači da podatku/metodi **može da pristupi klasa u kojoj je podatak/metoda definisan ili podklasa te klase, odnosno objekti koji su instance tih klasa**.

Enkapsulacija je jedan od ključnih koncepata objektnog programiranja koji omogućuje objedinjavanje podataka i operacija u jednom objektu. Ona sprečava da korisnik naruši invarijante klase, što je korisno jer omogućava da se implementacija klase objekata izmeni za aspekte koji nisu izloženi u interfejsu, a da te izmene ne zahtevaju da se interveniše na kodu koji klasu koristi. Definicije enkapsulacije u objektnom programiranju se dominantno fokusiraju na koheziju (grupisanje i pakovanje povezanih informacija).

2.1.2.2. Nasleđivanje u OOP-u

Nasleđivanje u objektno orijentisanom programiranju je mehanizam zasnivanja objekta ili klase na drugom objektu ili klasi, uz zadržavanje slične implementacije. Takođe se definiše i kao izvođenje novih klasa (podklasa) iz postojećih klasa (super klasa, osnovna/bazna klasa), a zatim njihovo formiranje u hijerarhiju klasa. Klasa koja (se) nasleđuje naziva se ***podklasa*** a klasa od koje se nasleđuje naziva se ***roditeljska klasa*** ili ***superklasa***. U većini objektno orijentisanih jezika, „podređeni objekat"/ „objekat-dete" (objekat kreiran nasleđivanjem) stiče sva svojstva i ponašanja „nadređenog objekta" /„roditeljskog objekta", sa izuzetkom konstruktora, destruktora i preopterećenih operatora.

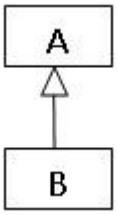
Nasleđivanje omogućava da se kreiraju klase koje su izgrađene na postojećim klasama, da se specificiraju nove implementacije uz zadržavanje istog ponašanja (ostvarivanje interfejsa), da se ponovo koristi kod i nezavisno proširuje softver putem javnih klasa i interfejsa.

Postoje različite vrste nasleđivanja zavisno od paradigme nasleđivanja i specifičnog programskog jezika. Razlikuju se sledeće vrste nasleđivanja:

- Jednostruko nasleđivanje;
- Višestruko nasleđivanje;
- Višenivovsko nasleđivanje;
- Hijerarhijsko nasleđivanje;
- Hibridno nasleđivanje.

2.1.2.2.1. Jednostruko nasleđivanje

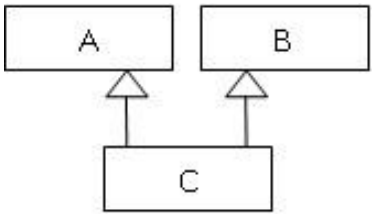
Jednostruko nasleđivanje je nasleđivanje gde naslednička klasa nasleđuje (atribute i metode) SAMO OD JEDNE klase (roditeljska klasa, superklasa) kao u što je ilustrovano slikom 2.3. gde naslednička klasa B nasleđuje od superklase A.



Slika 2.3. Jednostruko nasleđivanje

2.1.2.2.2. Višestruko nasleđivanje

Kod višestrukog nasleđivanja, naslednička klasa nasleđuje atribute i metode OD VIŠE NADREĐENIH KLASA (superklasa) što je ilustrovano slikom 2.4 gde klasa C nasleđuje od superklase A i superklase B.

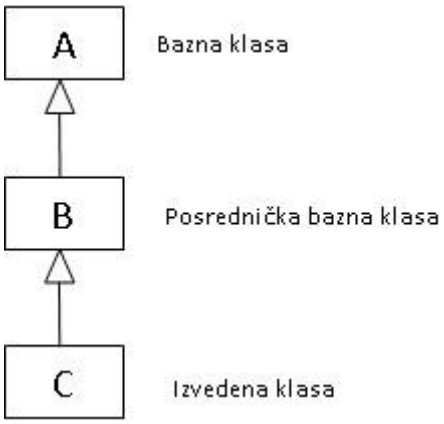


Slika 2.4 Višestruko nasleđivanje

Višestruko nasleđe je kontroverzno pitanje već dugi niz godina. Razlog tome je njegova povećana složenost i dvosmislenost u situacijama gde više od jedne roditeljske klase implementira isto svojstvo pa se postavlja pitanje od koje roditeljske klase se to svojstvo nasleđuje.

2.1.2.2.3. Višenivovsko nasleđivanje

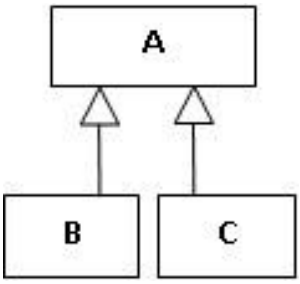
Višenivovsko nsleđivanje je vrsta nasleđivnja gde se ***podklasa* nasleđuje od *neke druge podklase***, odnosno gde se klasa izvodi iz izvedene klase. Ovakv odnos među klasma nije neuobičajen u OOP-u. Situacija višenivovskog nasleđivanja je ilustrovana na slici 2.5 gde je **klasa A koja nije izvedena iz druge klase** bazna klasa iz koje se izvodi klasa B. Zatim se iz klase B koja je izvedena klasa izvodi klasa C. Kako klasa B nasleđuje klasu A i klasa C nasleđuje klasu B, postoji veza nasleđivanja između klase A i klase C koja se ostvaruje posredno putem klase B. Klasa B se naziva se ***posrednička bazna klasa*** jer obezbeđuje vezu za nasleđivanje između klase A i klase C. Lanac ABC zove se ***putanja nasleđivanja***.



Slika 2.5 Višenivovsko nasleđivanje

2.1.2.2.4. Hijerarhijsko nasleđivanje

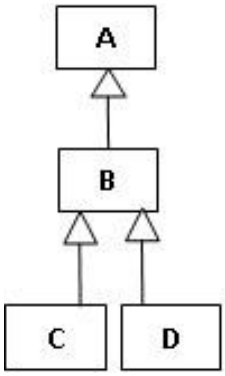
Ovo je vrsta nasleđivanja u kojoj JEDNA KLASA služi kao superklasa za više podklasa. Slika 2.6 ilustruje hijerarhijsko nasleđivanje gde roditeljska klasa A ima dve podklase B i C. Ovde i podklasa B i podklasa C imaju istu roditeljsku klasu A, ali su B i C dve odvojene podklase.



Slika 2.6 Hijerarhijsko nasleđivanje

2.1.2.2.5. Hibridno nasleđivanje

Hibridno nasleđivanje je mešavina dve ili više pethodno navedenih vrsta nasleđivanja. Na primer, situacija u kojoj klasa B nasleđuje od klase A i klase C i D nasleđuju od klase B, odnosno A ima podklasu B koja ima dve podklase C i D što je ilustrovano slikom 2.7. Ovde je u pitanju mešavina višenivovskog nasleđivanja i hijerarhijskog nasleđivajnja. Pri tome postoje dva hijerarhijska lanca lanac ABC i lanac ABD, dok je hijerarhijsko nasleđivanje ono u kome klase C i D nasleđuju od klase B koja nasleđuje od klase A .



Slika 2.7 Hibridno (višenivovsko i hijerarhijsko) nasleđivanje

U svojoj knjizi [1] Erik Eliot parafrazira viđenje suštine OOP-a pionira u oblasti objektnog programiranja Alana Keja (Alan Kay[11]). Prema Keju, osnovni sastavni delovi objektnog programiranja su:

- Prosleđivanje poruka;
- Enkapsulacija;
- Dinamičko vezivanje.

Kombinovanje prosleđivanja poruka i enkapsulacije služi nekim vrlo važnim svrhama.

- **Enkapsuliranje stanja izolovanjem drugih objekata od lokalnih promena stanja**. Jedini način da se utiče na stanje drugog objekta je da se slanjem poruke traži (ne naređuje) od tog objekta da svoje stanje promeni. Promene stanja se kontrolišu na lokalnom, ćelijskom nivou, a ne izložene zajedničkom pristupu.
- **Međusobno razdvajanje objekata**. Pošiljalac poruke je samo labavo povezan sa primaocem poruka, preko API-ja za razmenu poruka.
- **Adaptibilnost izvršavanja putem kasnog vezivanja**. Adaptibilnost izvršavanja pruža mnoge velike prednosti koje Alan Kej se smatra suštinskim za OOP.

Ono što je ovde upadljivo u odnosu na važeće raširenо shvatanje objektnog programiranja koje je reflektovano u najvećem broju objektnih programskih jezika, jeste da se **nasleđivanje** i **pod-klasni polimorfizam** NE SMATRAJU suštinom objektnog programiranja, odnosno da u suštinu objektnog programiranja NE SPADAJU:

- Klase;
- Klasno nasleđivanje;
- Specijalan tretman objekata/funkcija/podataka;
- Ključna reč new;
- Polimorfizam;
- Statički tipovi;
- Prepoznavanje klase kao tipa.

2.1.3. Deklarativno programiranje

Deklarativno programiranje u računarskoj nauci je stil programiranja koji izgrađuje strukturu i elemente računarskog programa tako da oni izražavaju logiku sračunavanja, bez opisivanja toka sračunavanja. Za razliku od imperativnog stila koji eksplicitno implementira algoritme opisujući precizno KAKO se zadatak rešava (korake pri rešavanju), deklarativni stil opisuje ŠTA program treba da uradi, dok se deo koji opisuje KAKO se to radi prepušta implementaciji jezika.

Deklarativno programiranje je stil programiranja gde programi opisuje svoje željene rezultate bez eksplicitnog navođenja komandi ili koraka koji moraju da se izvrše.

Deklarativno programiranje često programe tretira kao teorije formalne logike, a sračunavanje kao dedukciju u tom logičkom prostoru.

Jedna od prvih grupa deklarativnih programskih jezika su upitni jezici za baze podataka (n.pr., SQL[12], XQuery). Važna grupa su i markerski jezici (HTML, XML) a u tu grupu spadaju i regularni izrazi, logičko programiranje, sistemi za upravljanje konfiguracijama, ontološki jezici, itd.

Među deklarativnim jezicima danas se posebno izdvaja grupa jezika funkcionalnog programiranja koji se dalje dele na čiste i nečiste funkcionalne jezike. Osnovna karakteristika čistih funkcionalnih

[11] https://en.wikipedia.org/wiki/Alan_Kay

[12] SQL se može smatrati arhetipskim deklarativnim jezikom.

jezika je striktna kontrola bočnih efekata[13] u funkcijama i reprezentacija promena stanja programa samo putem funkcija koje menjaju stanje. Ipak, i čisti funkcionalni jezici često imaju mogućnost mešanja proceduralnog i funkcionalnog programiranja. Takođe, i neki jezici za logičko programiranje i upitni jezici podržavaju proceduralni stil programiranja.

Termin deklarativno programiranje je, kao što se iz prethodnog izlaganja može videti, *krovni termin* koji obuhvata veliki broj stilova programiranja od kojih ćemo ovde navesti i ukratko objasniti one koji se najčešće pojavljuju. Pri tome, nećemo navoditi jezike funkcionalnog programiranja jer će oni biti posebno detaljno obrađivani u svim ostalim poglavljima ove knjige.

Jezici za programiranje ograničenja iskazuju relacije između varijabli u obliku ograničenja koja specificiraju svojstva ciljnog rešenja. Skup ograničenja se rešava tako što se svakoj varijabli dodeljuju vrednosti tako da rešenje zadovoljava maksimalan broj postavljenih ograničena. Ovaj stil programiranja je često dopuna drugim stilovima kao što su funkcionalno, logičko, ili čak i imperativno programiranje.

Domenski specifični jezici (Domain-SpecificLanguages, DSL) su jezici koji ne moraju nužno da budu Tjuring-kompletni[14] što olakšava da jezik bude čisto deklarativan. U ovu grupu spadaju brojni markerski jezici (HTML, XML, MXML, XAML, XSLT, itd.), jezici za generisanje parsera (Yacc/Lex), jezici za specificiranje bildova i upravljanje konfiguracijama (Make, Puppet jezik), jezici regularnih izraza (POSIX standard, Perl jezik). U poslednjoj dekadi prošlog veka pojavili su se softverski sistemi koji kombinuju tradicionalne markerske jezika za korisnički interfejs (n.pr. HTML) sa deklarativnim markiranjem koje definiše ŠTA (ali ne i KAKO) treba da radi serverska strana da bi podržala deklarisani interfejs. Ovde se obično koristi domenski specifičan XML prostor imena a mogu se uključiti i apstrakcije sintakse SQL baze ili parametrizovanipoziviveb servisa korišćenjem REST-a[15] i SOAP-a[16].

Jezici logičkog programiranja poput jezika Prolog iskazuju relacije i upite nad njima. Specifikacija načina KAKO će se odgovoriti na te upite nije stvar izvornog programa, već stvar implementacije jezika i dokazivača teorema i tipično predstavlja neki oblik unifikacije[17]. Kao i kod funkcionalnog programiranja, mnogi logički jezici dozvoljavaju bočne efekte i, shodno tome, nisu striktno deklarativni.

Hibridni jezici su jezici koji imaju deo koda za specifikaciju zavisnosti u deklarativnom maniru, ali i deo koda koji predstavlja imperativnu listu akcija. Na primer, Yacc deklarativno specificira kontekstno slobodnu gramatiku jezika (za koji generiše parser), ali uključuje i snipete koda jezika-domaćina koji je obično imperativan jezik (n.pr. C).

Modeli, odnosno matematičke reprezentacije fizičkih sistema mogu se na računaru implementirati deklarativnim kodom. Takav kod sadrži „jednačine“ koji ne predstavljaju konstrukt dodele iz imperativnog jezika, već opisuju relacije ponašanja sistema. Kada se model izrazi ovakvim formalizmom, on predstavlja ulaz u program koji, koristeći agebarske manipulacije, formuliše najbolji

---

[13] Bočni efekat funkcije je svaka promena stanja koja se može opaziti izvan funkcije, izuzev povratne vrednosti funkcije.

[14] U teoriji sračunljivosti kaže se za sistem pravila manipulisanja podacima (n.pr. skup instrukcija računara, programski jezik ili ćelijski automat) da je **Tjuring-kompletan** ili **računski univerzalan** ako se pomoću njega može simulirati svaka **Tjuringova mašina**.

[15] **Representational state transfer** (**REST**) je arhitekturalni stil softvera koji definiše skup ograničenja za kreiranje Vebs ervisa(https://en.wikipedia.org/wiki/Representational_state_transfer).

[16] **SOAP** (skraćenica od **Simple Object Access Protocol**) je specifikacija protokola za razmenu poruka namenjenog razmeni strukturiranih informacija za implmentaciju veb servisa u računarskim mrežama (https://en.wikipedia.org/wiki/SOAP).

[17] U logici i računarskoj nauci, **unifikacija** je algoritamski proces rešavanja jednačina simboličkih izraza (https://en.wikipedia.org/wiki/Unification_(computer_science)).

algoritam za rešavanje. Ovi jezici su najčešće deo okruženja za vizuelno modelovanje a primeri su Analytica[18], Modelica[19], Simile[20], itd.

2.1.4. Funkcionalno programiranje

Funkcionalno programiranje je stil programiranja koji za rešavanje zadataka koristi *funkcije* i *kompoziciju funkcija*. Kako se operacija kompozicije ovde konceptualno svodi na jednu jedinu operaciju - izvršavanje funkcije nad rezultatom druge funkcijom - nema potrebe da se detaljnije opisuje kako program radi pa stoga jezici funkcionalnog programiranja ne spadaju u grupu imperativnih jezika. To ne znači da jezici koji se mogu klasifikovati kao imperativni ne podržavaju elemente funkcionalnog programiranja. U ovoj knjizi kao jednu tehnologiju koja implementira koncept funkcionalnog programiranja koristićemo jezik JavaScript koji se klasifikuje u imperativne, objektno-orijentisane jezike, a ipak pruža vrlo upotrebljive mogućnosti funkcionalnog programiranja.

U ovom odeljku prvo ćemo prikazati istoriju funkcionalnog programiranja. Nakon toga ćemo pokušati da kroz jednu metaforičnu priču objasnimo suštinu funkcionalnog programiranja i na kraju ćemo pobrojati i sažeto objasniti osnovne koncepte funkcionalnog programiranja.

2.1.4.1. Kratka istorija funkcionalnog programiranja

U ovom odeljku daćemo vrlo sažet opis okolnosti pod kojima se dešavao i dešava razvoj jednog od postojećih, u praksi sve više korišćenih stilova programiranja - funkcionalnog programiranja.

Prema izvoru [1], funkcionalno programiranje prošlo je do sada kroz dve faze razvoja, fazu uspona i fazu pada i trenutno se nalazi u drugoj fazi uspona. Pri tome, prva faza uspona funkcionalnog programiranja počinje 30-tih godina prošlog veka (dakle, pre nešto manje od jednog veka) pojavom radova dva izuzetno značajna stvaraoca u oblasti računarstva, Alana Tjuringa (Alan Mathison Turing)[21] i Alonza Čerča (Alonzo Church)[22]. Njih dvojica su predložila dva različita univerzalna modela računanja za koje je Čerč dokazao da su ekvivalentni.

Alan Tjuring je predložio model zvani *Tjuringova mašina* koji definiše apstraktnu mašinu sposobnu da manipuliše sekvencom simbola iz konačnog alfabeta primenjujući tabelu pravila nad beskonačnom memorijom[23]. Iako vrlo jednostavan, ovaj model je u stanju da implementira bilo koji algoritam. Tjuringova mašina je teorijska osnova imperativnog programiranja.

Alonzo Čerč je smislio *Lambda račun* (λ-račun)[24] koji je, takođe, univerzalni model računanja. Lambda račun je formalni sistem u matematičkoj logici za izražavanje računanja zasnovanog na apstraktnom konceptu funkcije i primeni funkcije korišćenjem povezivanja i zamene promenljivih. To je univerzalni model računanja koji se može koristiti za simulaciju bilo koje Tjuringove mašine. Lambda račun je teorijska osnova funkcionalnog programiranja.

Lambda račun je značajno uticao na razvoj softvera. Pre 80-tih godina prošlog veka, mnoge uticajne ličnosti iz domena računarske nauke razvijale su softver korišćenjem kompozicije funkcija. Period od druge polovine 50-tih godina do 70-tih karakterisan je razvojem niza programskih jezika oslonjenih

[18] https://docs.analytica.com/index.php/Analytica_for_programmers

[19] https://modelica.org/modelicalanguage.html

[20] https://www.simulistics.com/overview.htm

[21] https://en.wikipedia.org/wiki/Alan_Turing

[22] https://en.wikipedia.org/wiki/Alonzo_Church

[23] https://en.wikipedia.org/wiki/Turing_machine

[24] https://en.wikipedia.org/wiki/Lambda_calculus

na kompoziciju funkcija. Najuticajniji od tih jezika bio je LISP (danas Lisp), čiji je je razvoj godine 1958. inicirao John McCarty[25]. Lisp je dominantno oslonjen na Lambda račun i još uvek je u upotrebi u obliku različitih dijalekata (najpoznatiji su Closure, CommonLisp i Scheme), kao i u okviru drugih aplikacija (n.pr. popularni softver AutoCAD i druge aplikacije iz CAD oblasti koje podržavaju AutoLISP). Pored Lisp-a pojavili su se i sledeći jezici: Information Processing Language (IPL) iz 1956 koji se smatra prvim računarskim programskim jezikom funkcionalnog programiranja, jezici APL, J i K iz 60-tih godina prošlog veka, jezici iz 70-tih godina prošlog veka ML, SASL i NPL. John Backus je 1977. godine prikazao jezik FP gde definiše funkcionalne programe kao programe koji slede *princip kompozicionalnosti*[26]. Konstruktivistička teorija tipa[27], koju je 80-tih godina prošlog veka uveo Per Martin-Löf, uticala je na funkcionalne programske jezike, što se prvenstveno odnosi i na jezik Haskell koji je i otvoreni standard za funkcionalne jezike od 1990-tih.

Period između 1970 i 1980 obeležen je pristupom razvoju softvera koji je učinio otklon od jednostavne algebre ka proceduralnim jezicima poput jezika C (1978) i objektnim jezicima kao što su Smalltalk (1972), C++ i Java (od 1980 pa sve do 2010. godine).

Od 2010. godine ponovo se pažnja softverske industrije usmerava na funkcionalno programiranje. Nastaju brojni novi jezici a i postojeći se modifikuju sa ciljem da uključe funkcionalno programiranje. Danas se oni dele na dve velike grupe: (1) čiste (engl. *pure*) funkcionalne jezike (primeri su Haskell, Mercury, PureScript, itd.) i (2) nečiste (engl. *impure*) funkcionalne jezike (n.pr., JavaScript, Python, Clojure, Elm, Scala, Ruby, itd). Lista jezika klasifikovanih u ove dve grupe dostupna je na adresi https://en.wikipedia.org/wiki/List_of_programming_languages_by_type#Functional_languages.

Valja napomenuti da se nečisti funkcionalni jezici mnogo šire primenjuju, što je posledica jednostavne činjenice da je programiranje složen posao i da ni jedan postojeći stil programiranja, pa ni funkcionalno programiranje, nije univerzalno rešenje. Naravno, ne mogu se zanemariti ni stanje u industriji a ni navike i kompetencije programera koji još uvek nisu usvojili funkcionalni stil programiranja u meri koju on zaslužuje.

2.1.4.2. Suština funkcionalnog programiranja

Najbolji način da se objasni suština NEČEGA jeste da se napravi definicija toga NEČEGA. Definicija je odlično sredstvo za objašnjavanje suštine ako je definiciju moguće artikulisati tako da potpuno, precizno, tačno i razumljivo opisuje ono što definiše. Za razliku od matematike, gde je definicija jedan od temelja, u drugim oblastima ovakva striktna artikulacija nije jednostavna, a često nije ni moguća.

Funkcionalno programiranje, iako ima puno dodira sa matematikom, NIJE MATEMATIKA pa otuda i različiti pokušaji da se definiše funkcionalno programiranje. Ovde ćemo navesti tri karakteristična pokušaja više da ilustrujemo stanje, bez ambicije da usvojimo neki od njih.

Prvi, vrlo koncizan pokušaj[28], kaže: " Funkcionalno programiranje (FP) je stil razvoja softvera koji naglašava funkcije koje na zavise od stanja programa. "

Drugi pokušaj[29] dodaje dosta detalja: "U računarskoj nauci, **funkcionalno programiranje** je programska paradigma - stil izgradnje strukture i elemenata računarskih programa - koja računanje

[25] https://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)

[26] **Princip kompozicionalnosti** kaže da je značenje složenog izraza određeno značenjem njegovih konstituentnih izraza i pravila koja su korišćena u njihovom kombinovanju.

[27] https://en.wikipedia.org/wiki/Intuitionistic_type_theory#Martin-L%C3%B6f_type_theories

[28] https://www.manning.com/books/functional-programming-in-scala

[29] https://en.wikiversity.org/wiki/Theory_of_Programming_Languages/Functional_Programming#cite_note-1

tretira kao evaluaciju (određivanje vrednosti) matematičkih funkcija i izbegava promenljiva stanja i promenljive podatke. To je deklarativan stil programiranja u kome se programiranje vrši putem izraza ili deklaracija, umesto pomoću naredbi. Funkcionalni kod je idempotentan[30]: vrednost koju funkcija vraća zavisi samo od njenih argumenata, odnosno pozivanje funkcije sa istim argumentom uvek proizvodi isti rezultat."

Treći pokušaj[31], čiji je autor Brajan Lonsdorf, potencira raznovrsnost pogleda na funkcionalno programiranje i nije baš konvencionalan: "Funkcionalno programiranje (FP) ima više različitih definicija. Definicija Lisp programera je u mnogome različita od Haskell perspektive. OCaml FP nosi vrlo malo sličnosti sa paradigmom viđenom u Erlang-u. Čak ćete naići na nekoliko konkurentskih definicija u JavaScript-u. Ipak, postoji spojnica koja to povezuje -- nekakva zamućena 'znam-šta-bi-trebalo-da-bude-kada-ga-vidim' definicija, više kao opscenost (zaista, neki nalaze da je FP opsceno!)."

Zbog toga mi nećemo ni pokušavati da formulišemo eksplicitnu definiciju funkcionalnog programiranja, nego ćemo pokušati da suštinu funkcionalnog programiranja objasnimo tako što ćemo navesti temeljne koncepte funkcionalnog programiranja sa njihovim sažetim opisima iz izvora [1].

Prema izvoru [1], temeljni stubovi funkcionalnog programiranja su: *nepromenljivost* (*Immutability*), *separacija* (*Separation*), *kompozicija* (*Composition*) i tok (*Flow*). Izvor [1] ih objašnjava kroz priču koja izgleda ovako.

"Lutao sam kroz arhive stare biblioteke i našao mračan tunel koji je vodio do sobe iz računarstva. Tamo sam našao svitak koji je, izgleda, pao na pod i bio zaboravljen. Svitak je bio upakovan u prašnjavu čeličnu cev na kojoj je bila oznaka: 'From the archives of The Church of Lambda' ('Iz arhiva Crkve Lambda'/ 'Iz arhiva Čerča od Lambde')[32]. Bio je obavijen majušnim listom papira na kome je pisalo:

Majstor programer i njegov šegrt sedeli su u Tjuringovoj meditaciji, razmišljajući o Lambdi. Šegrt je pogledao majstora i upitao: 'Majstore, Vi mi kažete da pojednostavim, ali programi su složeni. Radni okviri olakšavaju kreiranje otklanjanjem teških izbora. Šta je bolje, klasa ili radni okvir?'

Majstor programer pogledao je svog šegrta i upitao: 'Zar ti nisi čitao učenja mudrog majstora Karmaka koji kaže ... 'Ponekad, elegantna implementacija je samo funkcija. Ne metod. Ne klasa. Ne radni okvir. Samo funkcija.'*[33]*

'Ali majstore,...' započeo je šegrt, a majstor ga je prekinuo pitanjem: 'Zar nije tačno da termin **nije funkcionalan** ima značenje reči **nefunkcionalan**?'

I tada je šegrt razumeo."

Ono što je ovde, možda, potrebno napomenuti je opet u vezi sa prevodom sa engleskog jezika (originalni tekst) na srpski jezik. Termini upotrebljeni respektivno u originalnom tekstu su ***nonfunctional*** i ***dysfunctional***. Oni se na srpski jezik prevode u istu reč ***nefunkcionalan***, što ovde nije prihvatljivo jer se termin ***nonfunctional*** ovde koristi kao karakterizacija stila programiranja - stil

---

[30]**Idempotentnost** je osobina određenih operacija u matematici I računarskoj nauci da ih je moguće primeniti više puta bez promene rezultata koji je dobijen prvom primenom (n.pr. možete da množite brojem 1 koliko god puta hoćete, a rezultat će uvek da bude isti).

[31]https://snipcart.com/blog/functional-programming-paradigm-concepts

[32]Ovaj alternativni naslov je posledica igre reči. Naime, *Alonzo Church* (Church je na engleskom crkva) je izumitelj teorijskog fundamenta računarskog, posebno funkcionalnog programiranja koji se zove *Lambda račun* (na engleskom Lambda Calculus). Toliko o jednoznačnosti prirodnog jezika.

[33]https://twitter.com/id_aa_carmack/status/53512300451201024

programiranja koji nije funkcionalni stil, a reč ***dysfunctional*** kao karakterizacija ishoda programiranja - programa koji se odlikuje radom koji nije normalan ili odgovarajući/očekivani. Tako da je stav majstora potpuno jasan: program koji nije pisan funkcionalnim stilom je nefunkcionalan.

Autor pomenutog izvora [1] dalje "čita" svitak na kome je i indeks sa listom knjiga koje se nalaze u toj sobi. Pokušava da čita neke od njih, ali ne razume njihov sadržaj pa se vraća čitanju komentara ispisanih na marginama liste referenci svitka. A u tim komentarima piše, u suštini, sledeće.
"Nepromenljivost: ***Jedina istinita konstanta je promena. Mutacija[34] sakriva promenu. Sakrivena promena ispoljava haos. Zato, mudri uzimaju u obzir istoriju.*****"** Ovo objašnjenje se može ilustrovati jednostavnim primerima poput sledećeg: ako dete ima jednu čokoladu i dobije još jednu čokoladu, imaće dve čokolade. Ali to ne menja činjenicu da je, pre no što je dobilo drugu čokoladu, već imalo jednu čokoladu što nećemo znati ako to nekako ne "zapamtimo". Mutacija pokušava da izbriše istoriju, ali se istorija ne može izbrisati bez posledica, jer prošlost može da se vrati kao utvara koja se u softveru manifestuje kao greška.

"Separacija: ***Logika je mišljenje. Efekti se postižu delovanjem. Zbog toga, mudar razmišlja pre no što deluje i deluje tek nakon što je razmislio. Pri pokušaj da se istovremeno deluje i misli, delovanje može da dovede do skrivenih pratećih efekata koji uzrokuju greške u logici.*****"** I opet primer: Dolaze Vam gosti i Vi želite da pripremite sendviče za posluženje. Pri tome, svi gosti jako vole sendviče sa majonezom, ali neki zbog zdravlja ne smeju da konzumiraju so, dok drugi baš vole da majonez bude malo više slan. Takođe, Vi ste nabavili samo neslan majonez i nemate vremena da ponovo idete u kupovinu. Da bi Vam svi gosti bili zadovoljni, morate da napravite sendviče koji sadrže neslan majonez i one koji sadrže slan majonez. A da biste to uradili, treba prvo da pravite sendviče sa neslanim majonezom pa da zatim posolite majonez i da pravite sendviče sa slanim majonezom. U funkcionalnom programiranju (i ne samo funkcionalnom) to odgovara dekomponovanju problema na manje probleme koji pojedinačno mogu lakše da se reše.

"Kompozicija: ***U prirodi stvari rade u harmoniji. Drvo ne može da raste bez vode, ptica ne može da leti bez vazduha. Zbog toga, mudri mešaju sastojke da naprave ukusniju supu. Kompoziciju treba planirati. Praviti funkcije čiji će izlazi prirodno funkcionisati kao ulazi u više drugih funkcije. Oznake funkcija (signature) treba da budu što jednostavnije.*****"** A primer je: funkcija koja samo sabira skup brojeva može da obezbedi izlaz koji će koristiti svaka druga funkcija kojoj je potreban zbir nekog skupa brojeva. Funkcija koja sabira treba da kao ulaz prima samo potrebne argumente (brojeve koje treba da sabere) i da ima prikladno ime (recimo, **`sum`**a ne **`x406Abc`**). U programiranju to odgovara rešavanju složenih problema kombinovanjem rešenja jednostavnijih problema.

"Konzervacija: ***Vreme je dragoceno, a posao zahteva vreme. Zbog toga, mudri višekratno koriste svoje alate koliko god je to moguće. Budale kreiraju specijalne alate za svaku novu kreaciju.*****"** A primer je: ako ste jednom napravili funkciju **`sum`**, nećete je ponovo praviti kad god Vam zatreba sabiranje. Međutim, broj elemenata koje sabirate može da bude različit, sabirci i rezultat mogu da budu celobrojni ili realni brojevi, ili nešto treće. Funkcije koje su tipski specifične ne mogu se koristiti za podatke različitih tipova. Mudar programer pravi funkcije koje mogu da rade nad različitim tipovima podataka ili podatke „pakuje" tako da izgledaju kao ono što funkcija očekuje. Univerzalna apstrakcija za ovo su liste i elementi. U programiranju, to je princip ponovne upotrebljivosti.

"Tok: ***Stajaće vode su nebezbedne za piće. Ostavljena hrana će istrunuti. Mudri prosperiraju zato što puštaju stvari da teku. ... Napomena uređivača: jedina ilustracija na svitku ispod ovog teksta bila je niz različitih pataka koje plutaju niz tok potpisana naslovom: Lista izražena u vremenu je struja. ... A u odgovarajućoj napomeni je pisalo: Deljeni objekti i fiksture podataka[35] mogu da dovedu do interferencije (mešanja) funkcija, niti (thread) koji se nadmeću za iste resurse mogu***

[34]Mutacija je trajna promena nečega (najčešće u biologiji – DNA, geni, ...)

[35]Fikstura podataka (*data fixture*) predstavlja fiksirane podatke, najčešće u svrhu testiranja softvera.

jedna drugu da 'sapletu'. O programu se može rasuđivati i ishodi se mogu predvideti samo ako podaci slobodno teku kroz čiste[36] funkcije." U programiranju ovo odgovara problemu asinhronog ponašanja ome mi, ljudi nismo baš vični pa nam je teško i da ga razumemo i da ga implementiramo u kod.

Svitak se završavao ovim poslednjim iskazom bez komentara:

"Mudrost: Mudar programer razume put pre no što njime prođe. Zato je mudar programer *funkcionalan*, dok se nemudar gubi u *džungli.*"

Prethodna priča tačno i vrlo slikovito iskazuje osnovnu ideju funkcionalnog programiranja - razumeti put pre no što se njime prođe što se u funkcionalnom programiranju sublimiše u *deklarativnom stilu* - ne mora se znati KAKO neki kod radi, važno je da se zna ŠTA treba da uradi.

2.1.4.3. Koncepti funkcionalnog programiranja

Ima koncepata specifičnih za funkcionalno programiranje koji su baš strani imperativnom programiranju uključujući i objektno programiranje a, naravno, ima i koncepata u kojima se ovi stilovi podudaraju. Međutim, realnost je da su programski jezici multiparadigmatski bar u nekoj meri, pa se koncepti svojstveni funkcionalnom programiranju javljaju i u imperativnim jezicima. Ovde ćemo ukratko izložiti osnovne koncepte funkcionalnog programiranja koje ćemo u nastavku detaljnije nalizirati. Koncepti koje ćemo izložiti su sledeći:

- Funkcije i čiste funkcije;
- Referencijalna transparentnost;
- Rekurzija;
- Striktna i ne-striktna evaluacija;
- Sistem tipiziranja,
- Imutabilnost,
- Funkcionalne strukture podataka.

2.1.4.3.1. Funkcije i čiste funkcije

Sam naziv stila programiranja kojim se ovde bavimo ističe koncept funkcije u prvi plan. Zato ćemo se, prvo, podsetiti značenja termina **funkcija** u širem smislu i značenje istog termina u matematici. Pri tome, objašnjenje će biti dato sa isključivim ciljem da se kasnije objasni funkcionalno programiranje i nećemo insistirati na rigoroznom matematičkom formalizmu pri objašnjavanju funkcije u matematici, niti na drugim značenjima ovoga termina.

2.1.4.3.1.1. O funkcijama

U širem smislu funkcija je:

- Sposobnost nekoga/nečega da izvrši neku akciju za koju je posebno prilagođen ili korišćenje za izvršenje te akcije
    - Na primer, "lekar vrši funkciju lečenja ljudi" što znači da onaj koji leči ljude mora da bude za to osposobljen (da je lekar).
- Svrha postojanja nekoga/nečega
    - Na primer, "povrće postoji da bi ljudi imali neophodnu hranu za preživljavanje " što znači da je svrha postojanja povrća da se ljudima obezbedi izvor sastojaka neophodnih za održanje života.

[36]Čista funkcija je vrlo važan koncept u funkcionalnom programiranju i, u osnovi, predstavlja funkciju koja za iste ulaze uvek daje isti izlaz i ne proizvodi nikakve bočne efekte.

U matematici, funkcije su u osnovi opisi načina na koji promena jedne veličine zavisi od druge veličine, n.pr.: obim kruga od poluprečnika, položaj planete od vremenskog trenutka, itd. Veličina od koje zavisi neka druga veličina se zove **argument** funkcije, a zavisna veličina se zove **vrednost** funkcije.

U matematici, funkcija je **relacija između skupova** koja **svakom elementu jednog skupa** (**oblast/domen/ skup dozvoljenih vrednosti argumenata funkcije**) ***pridružuje*** **tačno jedan element drugog skupa** (**oblast/ kodomen/ skup vrednosti funkcije**)[37]. Tipični primeri su, recimo, funkcije koje elementima nekog skupa brojeva (n.pr., celi, realni) dodeljuju elemente tog istog skupa.

Na osnovu objašnjenja matematičke funkcije, možemo da kažemo da je **funkcija proces koji prihvata ulaze zvane *argumenti* i daje izlazni rezultat zvani *povratna vrednost***. Ovaj proces pokreće se ***pozivanjem funkcije***.

Funkcije se u programiranju koriste na dva osnovna načina:

- **Procedura: Izvršava proizvoljnu kolekciju funkcionalnosti. Ta kolekcija se naziva *procedura* a stil programiranja se zove *proceduralno programiranje*. Procedura može, a ne mora da ima argumente i može a ne mora da ima povratnu vrednost.**
- **Mapiranje:** Proizvodi neke izlaze na bazi zadatih ulaza tako što **mapira** (preslikava) ulazne vrednosti na izlazne vrednosti. Mapiranje **može da ima argumente** i **mora da ima povratnu vrednost**.

U prvom načinu korišćenja, oslanjamo se na opštiju definiciju funkcije koja potencira svrhu, odnosno ne insistira na argumentima i povratnoj vrednosti. Drugi način korišćenja funkcije je direktna primena prethodno navedene matematičke definicije - funkcija preslikava elemente skupa oblasti definisanosti (argumenti) na elemente skupa vrednosti funkcije (povratna vrednost).

U funkcionalnom programiranju, **fokus je na mapiranju** odnosno **korišćenju koncepta funkcije kakav se javlja u matematici**.

Ono što ćemo još pomenuti su osnovne stvari o načinu na koji se funkcija definiše i načinu na koji funkcija komunicira svoje ulazne podatke i povratnu vrednost.

Funkcija se definiše svojim opisom koji, u opštem slučaju, obuhvata njenu identifikaciju, opis njenih ulaznih podataka, opis procesa kojim se ulazni podaci transformišu u povratnu vrednost, i instrukciju za vraćanje povratne vrednosti i završavanje izvršavanja poziva funkcije.

Kada govorimo o načinu na koji funkcija prima ulazne podatke, javljaju se dva termina: *argument* i *parametar*. Ponekad, ova dva termina se (neopravdano) tretiraju kao sinonimi.

Tačna interpretacija je da termin *argument* označava vrednost koja se prosleđuje funkciji, a termin *parametar* označava imenovani entitet/varijablu unutar funkcije koji prihvata prosleđenu vrednost - argument.

Na primer, u sledećem fragmentu koda `x` i `y` su parametri, dok su `a` i `a*2` (t.j. vrednosti 2 i 6 respektivno) argumenti:

```
function imeFunkcije(x,y) { // Deklaracija funkcije, x i y su
parametri

/* Ovde dolazi telo funkcije - kod koji se izvršava pri pozivu
funkcije */
```

[37]Jednom elementu domena može se pridružiti SAMO JEDAN element kodomena. Sa drugestrane, VIŠE RAZLIČITIH elemenata domena može se mapirati na ISTI elemement kodomena.

```
}


let a = 3;
imeFunkcije(a, a*2); // Ovo je poziv funkcije sa argumentima a i a*2
```

Broj argumenata koje funkcija očekuje određen je brojem parametara[38]. Ovaj broj se naziva ***arnost*** funkcije. U poglavlju koje se posebno bavi funkcijama u JavaScript-u će biti dato još detalja koji se odnose na parametre i argumente u programskom jeziku JavaScript.

Sledeća napomena odnosi se na povratnu vrednost funkcije. Da bi se maksimalno ostalo u duhu funkcionalnog programiranja, što znači korišćenje funkcija a ne procedura, funkcije bi uvek trebale da imaju povratnu vrednost. Funkcija može da vrati maksimalno jednu vrednost pa je, u slučaju potrebe da se vrati više vrednosti, opcija da se one organizuju u složenu vrednost, poput niza ili objekta. U nastavku će biti još detalja koji se odnose na povratnu vrednost funkcije u jeziku JavaScript.

U funkcionalnim programskim jezicima funkcije su "građani prvog reda" (engl. first class citizens). To znači da se sa njima, pored pozivanja, može raditi sve ono što se radi sa bilo kojim jezičkim entitetom prve klase (kao što je, na primer, brojčana vrednost): mogu se dodeljivati varijablama, mogu se prosleđivati kao argumenti drugim funkcijama, mogu da budu povratne vrednosti drugih funkcija.

Funkcija koja prima jednu ili više drugih funkcija i/ili vraća funkciju ima posebnu ulogu u funkcionalnom programiranju i posebno ime - *funkcija višeg reda*.

2.1.4.3.1.2. Čista funkcija

U funkcionalnom programiranju od posebnog značaja je koncept **čiste funkcije**. Čista funkcija je funkcija koja:

- Za **isti ulaz** uvek **vraća isti izlaz**.
- Ne proizvodi bočne efekte.

Jasan znak da je funkcija nečista je ako ima smisla da se ona pozove a da se ne koristi njena povratna vrednost. Za čiste funkcije to nije dozvoljeno.

Čiste funkcije, zbog mnogih svojih prednosti, predstavljaju temelj funkcionalnog programiranja. One su potpuno nezavisne od spoljašnjeg stanja i zbog toga su imune na česte greške u softveru koje su posledica deljenih stanja. Nezavisna priroda čini ih izuzetno pogodnim za računanja na višeprocesorskim arhitekturama i za premeštanje u okviru koda, refaktorizovanje i reorganizovanje a njihova primena u programima čini te programe fleksibilnijim i otvorenijim za buduće promene. U nastavku će još biti reči o prirodi i primeni čistih funkcija i detaljno ćemo objasniti šta znači termin ***bočni efekat***.

2.1.4.3.2. Rekurzija

Jednostavno rečeno, ***rekurzivne funkcije*** su one **funkcije koje same sebe pozivaju** uz **ponavljanje operacije pozivanja** dok se **ne dostigne *bazni slučaj***. ***Rekurzija*** je, naravno, korišćenje rekurzivnih funkcija. Rekurzija (pozivanje iste funkcije) je vrlo bliska konceptu imperativne petlje gde se takođe „ista stvar“ računa više puta u cikličkom maniru.

U opštem slučaju, rekurzija zahteva održavanje steka koji konzumira memoriju linearno proporcionalno dubini rekurzije[39] što rekurziju čini skupom za primenu umesto imperativnih petlji.

[38] Broj parametara i broj argumenata u opštem slučaju ne moraju da budu jednaki.

[39] ***Dubina rekurzije*** je maksimalan broj ugnježdenih poziva rekurzivne funkcije, uključujući i prvi poziv.

Međutim, kompajler može da prepozna poseban oblik rekurzije zvani *terminalna rekurzija* i da ga iskoristi za optimizaciju koda na način da implementacija rekurzije zahteva manju količinu memorije. Na taj način rekurzija postaje upotrebljiva za implementaciju iteracija u funkcionalnim jezicima. U svakom slučaju, ovo nije lak zadatak i različiti jezici ga rešavaju na različite načine.

Većina funkcionalnih jezika opšte namene dozvoljava neograničenu rekurziju i istovremeno su Tjuring-kompletni. To čini problem zaustavljanja neodlučivim, može da dovede do nekonzistentnosti u jednačinskom rezonovanju i generalno zahteva uvođenje nekonzistentnosti u logiku koja se izražava sistemom tipiziranja u jeziku. Postoje i namenski funkcionalni jezici (n.pr. Coq) koji dopuštaju samo *dobro utemeljenu* (engl. *well-founded*) rekurziju. Posledica je njihova Tjuring-nekompletnost[40] i nemogućnost izražavanja određenih funkcija. Funkcionalno programiranje ograničeno na dobro utemeljenu rekurziju sa još nekoliko ograničenja (n.pr. da svaka funkcija mora da ima definiciju za sve unutar svog domena) zove se *totalno funkcionalno programiranje*.

Koncept rekurzije je veoma značajan u funkcionalnom programiranju pa će njemu biti posvećeno posebno poglavlje u drugom delu ove knjige.

2.1.4.3.3. Striktna i ne-striktna evaluacija

Evaluacija je postupak kojim se određuje vrednost izraza; na primer, ako u izrazu $2 * b$ "opšti broj" b zamenite sa 2 izraz će biti evaluiran na vrednost 4.

Koncept se odnosi na način obrade argumenata pri evaluaciji izraza.

Funkcionalni jezici mogu se kategorisati po strategiji evaluacije. Po ovom kriterijumu klasifikuju se u dve grupe: jezici sa striktnom/pohlepnom (engl. *eager*) ili nestriktnom/lenjom (engl. *lazy*) evaluacijom.

Pri striktnoj evaluaciji vrši se obrada argumenata "unapred", bez obzira da li je evaluirana vrednost potrebna u trenutnoj fazi izvršavanja.

Pri lenjoj evaluaciji obrada se vrši u trenutku kada je evaluirana vrednost potrebna za izvršavanje programa.

Tehnička razlika je u denotacionoj[41] semantici izraza koji sadrže računanja koja mogu da otkažu (proizvedu grešku). Pri striktnoj evaluaciji, evaluacija svakog termina koji sadrži podtermin koji otkazuje takođe otkazuje. Na primer, naredba koja treba da izda dužinu niza:

```
print length([2+1, 3*2, 1/0, 5-4])
```

otkazuje u slučaju striktne evaluacije zbog deljenja sa nulom u trećem članu niza (`1/0`), a ne otkazuje pri lenjoj evaluaciji jer za određivanje dužine niza nije potrebno sračunavati vrednosti članova niza.

Većina jezika primenjuje striktnu evaluaciju kao podrazumevanu, ali pruža i mogućnost lenje evaluacije.

2.1.4.3.4. Tipiziranje i sistem tipova u funkcionalnom programiranju

U računarskom programiranju, sistem tipova je logički sistem koji se sastoji od skupa pravila koja svakom terminu (reči, frazi ili drugom skupu simbola) dodeljuju osobinu koja se zove tip (na primer, ceo broj, string, logički tip).

Sistem tipiziranja i tipovi u programiranju su mehanizam kojim se proverava prisustvo grešaka u kodu (i one se eventualno uklanjaju) do kojih bi moglo da dođe ako se ta pravila ne poštuju. Najjednostavniji primer je, recimo, pokušaj da saberete tri vrednosti od kojih je jedna numeričkog

[40] Programski jezik je Tjuring kompletan ako je u stanju da simulira svaku Tjuringovu mašinu.

[41] U računarskoj nauci, denotaciona semantika je pristup formalizovanja značenja programskih jezika konstruisanjem matematičkih objekata (denotacija) koji opisuju značenja izraza iz jezika.

tipa (recimo broj 2), druga je logičkog tipa (recimo vrednost `false`), a treća je tipa string (recimo vrednost 'A' . Šta bi ovde mogao da bude smislen rezultat? Da li bi rezultat bio broj ili logička vrednost ili nekakav string?

Posebno od pojave Hindley-Milner sistema tipiziranja[42], u funkcionalnim jezicima prisutna je tendencija korišćenja tipiziranog Lambda računa[43]. Ovaj pristup tipiziranju **odbacuje sve programe koji se u fazi kompilacije klasifikuju kao nevalidni**, dok pristup koji koristi netipizirani Lambda račun **prihvata sve programe koji se u fazi kompilacije klasifikuju kao validni**.

Prvi pristup nosi rizik lažnih pozitivnih grešaka (da se nešto što će se u fazi izvršenja pokazati kao nevalidno proglasi validnim), dok drugi pristup nosi rizik lažnih negativnih grešaka (da se nešto što će se u fazi izvršenja pokazati kao validno proglasi nevalidnim).

Naravno, konačan rezultat se manifestuje u fazi izvršavanja kada se odbacuju svi nevalidni programi i kada postoji dovoljno informacija da se ne odbace validni programi.

Korišćenje algebarskih tipova podataka[44], što je praksa u funkcionalnom programiranju, pogoduje manipulisanju složenim strukturama podataka, a postojanje stroge provere tipa u fazi kompilacije čini programe pouzdanijim. Takođe, mogućnost zaključivanja tipa u većini slučajeva oslobađa programera obaveze da ručno deklariše tipove kompajleru.

Postoji i koncept ***zavisnih tipova***[45](engl. *dependent types*). Ovi sistemi tipiziranja nemaju odlučivo zaključivanje tipa i teži su za razumevanje i programiranje. Prednost im je što mogu da izraze proizvoljne iskaze predikatske logike. Uz oslonac na Kari-Hauardov (Curry-Howard) izomorfizam[46], dobro-tipizirani programi u jezicima koji podržavaju zavisne tipove (n.pr., Coq, Agda, Cayenne, i Epigram) su sredstvo za pisanje formalnih matematičkih dokaza iz kojih kompajler može da generiše ***sertifikovan kod***[47]. Iako su ovi jezici za sada od primarnog interesa za istraživački auditorijum, počinju da se koriste i u praksi[48]. *Generalizovani algebarski tipovi podataka* (engl. *Generalized Algebraic Data Types*, GADT)[49], koji su ograničen oblik zavisnih tipova, mogu se implementirati tako da obezbede neke pogodnosti zavisno tipiziranog programiranja, uz izbegavanje većine nepogodnosti. Raspoloživi su za Glasgow Haskell Compiler, OCaml (od verzije 4.00) i za jezik Scala, a postoje predlozi i za druge jezike (n.pr. Java i C#).

2.1.4.3.5. Referencijalna transparentnost

Idealni funkcionalni programi ne bi trebali da imaju naredbe dodele, odnosno vrednost varijable u funkcionalnom programu trebala bi da bude nepromenljiva - kada se jednom definiše, ne bi smela da može da se menja. Na taj način se eliminiše svaka mogućnost bočnih efekata jer se svaka varijabla može zameniti svojom vrednošću u svakoj tački izvršavanja. Zato su idealni funkcionalni programi *referencijalno transparentni*.

Posmatrajmo naredbu dodele:

---

[42] https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system

[43] https://en.wikipedia.org/wiki/Typed_lambda_calculus

[44] Algebarski tip podataka (engl.Algebraic Data Type,ADT) je vrstakompozitnogtipa – tipa koji se dobijakombinovanjemdrugihtipova. Jedan od najuobičajenijih tipova ove vrste je jednostruko povezana lista.

[45] U računarskojnauci i logici **zavisni tip** je tip čijadefinicijazavisiodvrednosti.

[46] https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence

[47] Kod za koji je dokazano da zadovoljava formalnu specifikaciju svogp onašanja - verifikacijakorektnostiprograma

[48] Na primer, Compcert je kompjaler za podskupjezika C koji je pisan u jeziku Coq i formalnoverifikovan.

[49] U funkcionalnom programiranju, generalisani algebarski tip podataka (engl. Generalized Algebraic DataType, GADT) je generalizacija parametarskih algebarskih tipova podataka.

```
x = x * 10
```

Ova naredba menja vrednost dodeljenu varijabli x. Pretpostavimo da je inicijalna vrednost dodeljena varijabli x bila 1, i da su zatim dve uzastopne evaluacije varijable x dale vrednosti 10 i 100 respektivno. Jasno je da zamena `x = x * 10` sa 10 ili 100 daje programe sa različitim značenjem – taj izraz se ne može u programu zameniti svojom izračunatom vrednošću. Zato ovaj izraz *nije referencijalno transparentan*. U stvari, naredbe dodele *nikada nisu referencijalno transparentne*.

Nasuprot tome, funkcija

```
function plusone(x) {return x + 1;}
```

jeste referencijalno transparentna zato što za isti ulaz uvek vraća isti izlaz i u kodu se može zameniti svojom izračunatom vrednošću a da to ne utiče na program. U funkcionalnom programiranju nastoji se da se u najvećoj mogućoj meri koristi ovaj tip funkcije. Međutim, valja napomenuti da je referencijalna transparentnost, iako ima određenu praktičnu vrednost, ipak više od teorijskog značaja.

2.1.4.3.6. Imutabilnost

Imutabilnost je važan princip funkcionalnog programiranja. Odnosi se na svojstvo entiteta/elemenata programa koje ih karakteriše kao ***nepromenljive***. Nepromenljivost ovde znači da se entitet, kada se jednom kreira, ne može menjati do kraja svog postojanja. Tehnički gledano, može se reći da je imutabilan onaj entitet u programu za koji se zna svako stanje u kome se entitet nalazio u toku svog postojanja.

U najvećem broju programskih jezika primitivni tipovi su podrazumevano imutabilni.

U objektno orijentisanom i funkcionalnom programiranju, nepromenljivi objekat je objekat čije stanje ne može da se menja nakon što je kreiran, nasuprot promenljivom objektu koji se može modifikovati nakon što je kreiran. U nekim slučajevima, objekat se smatra nepromenljivim čak i ako su neki interni atributi promenljivi pod uslovom da je stanje objekta „spolja“ gledano nepromenjeno; na primer, objekat koji koristi memoizaciju za skupe proračune bi se mogao smatrati nepromenljivim objektom, iako dodaje novi rezultat u internu tabelu keširanih rezultata svaki put kada u njoj nema rezultata za zadatu konfiguraciju ulaza u proračun.

U čistim funkcionalnim programskim jezicima nije moguće kreirati mutabilne objekte bez proširenja jezika (npr. preko promenljive biblioteke referenci ili interfejsa stranih funkcija), tako da su u tim jezicima svi objekti imutabilni.

Nasuprot tome, u jezicima koji nisu čisti funkcionalni jezici objekti su podrazumevano mutabilni.

U programskom jeziku JavaScript koji mi ovde koristimo svi primitivni tipovi su podrazumevano imutabilni, dok su objekti podrazumevano imutabilni.

Naravno, i imutabilnost ima svoje prednosti i nedostatke. Osnovna prednost je što imutabilnost obezbeđuje istoriju promena za programski entitet i time značajno olakšava testiranje. Osnovni nedostatak je što zahteva veće resurse i komplikuje strukture podataka.

2.1.4.3.7. Funkcionalne strukture podataka

Čiste funkcionalne strukture podataka često se predstavljaju na drugačiji način od odgovarajućih imperativnih struktura. Dominantni razlog za to je imutabilnost.

Postoje dva osnovna uzroka koji dizajn i implementaciju struktura podataka čine težom u ambijentu funkcionalnog programiranja nego u slučaju imperativnog programiranja.

Prvi uzrok je problem naredbi dodele (destruktivnog ažuriranja) što se dominantno koristi u imperativnoj paradigmi a nije prihvatljivo za funkcionalnu paradigmu koja insistira na perzistenciji.

Drugi uzrok je što se od funkcionalnih struktura podataka očekuje veća fleksibilnost nego što je to slučaj sa imperativnim strukturama.

Kada se ažurira imperativna struktura podataka, po pravilu se prihvata da stara struktura neće biti više raspoloživa. U slučaju funkcionalnih struktura situacija je obrnuta, očekuje se da budu dostupne i stara i nova verzija strukture. Ovo svojstvo naziva se *perzistencija strukture*. Sve to, naravno, dovodi do veće složenosti i manje efikasnosti funkcionalnih struktura podataka. Na svu sreću, pokazuje se da je često moguće osmisliti funkcionalne strukture podataka koje su asimptotski podjednako efikasne kao i najbolja imperativna rešenja. U jeziku Closure, perzistentne strukture podataka se koriste kao funkcionalne alternative svojim imperativnim analogonima. Na primer, perzistentni vektori koriste stabla za parcijalno ažuriranje tako da pozivanje metode rezultuje kreiranjem samo nekih čvorova, a ne cele nove strukture.

Više detalja o čistim funkcionalnim strukturama podataka čitalac može naći na adresi https://en.wikipedia.org/wiki/Purely_functional_data_structure, a knjiga [2] autora Krisa Okasakija (Chris Okasaki) i njegova doktorska disertacija[50] su autoritativni izvori za ovu temu.

2.2. Sažetak

Paradigma programiranja je termin koji se koristi da označi klasifikaciju programskih jezika baziranu na svojstvima jezika. Razlikuju se dve osnovne paradigme:

- **Imperativna** koja opisuje računanje putem naredbi koje **menjaju stanje programa** i usredsređuje se na opisivanje **KAKO program radi**. Danas je najzastupljenija imperativna paradigma zvana objektno-orijentisano programiranje zasnovana na konceptu ***objekt*** i ***prosleđivanju poruka među objektima***. Objektno-orijentisani programski jezici su raznovrsni, ali je većina najpopularnijih ***klasno-bazirana*** što znači da su objekti instance klasa koje određuju tipove objekata.

- **Deklarativna** koja izgrađuje strukturu i elemente računarskog programa tako da oni **izražavaju logiku sračunavanja**, **bez opisivanja toka sračunavanja**. Deklarativni stil opisuje **ŠTA program treba da uradi**, dok se deo koji opisuje KAKO se to radi prepušta implementaciji jezika. Izraziti predstavnici ove paradigme su jezici funkcionalnog programiranja.

Funkcionalno programiranje je stil programiranja koji za rešavanje zadataka koristi *funkcije* i *kompoziciju funkcija*. Ima jaku teorijsku osnovu koja obuhvata teorijski model računanja zvani λ-račun i teorijski osnov koji se zove teorija kategorija.

Osnovni koncepti funkcionalnog programiranja su:

- Funkcije i čiste funkcije;
- Referencijalna transparentnost;
- Sistem tipiziranja i striktna i ne-striktna evaluacija;
- Rekurzija;
- Funkcionalne strukture podataka.

Jezici funkcionalnog programiranja spadaju u grupu deklarativnih jezika. Međutim, vrlo je prisutan trend uvođenja podrške funkcionalnom stilu u imperativne jezike. U ovoj knjizi kao tehnologija

[50] https://www.cs.cmu.edu/~rwh/students/okasaki.pdf

koristi se jezik JavaScript koji se klasifikuje u imperativne, objektno-orijentisane jezike, a ipak pruža vrlo upotrebljive mogućnosti funkcionalnog programiranja.

Istorijski gledano, ideja funkcionalnog programiranja pojavila se daleko pre ideje objektne paradigme, još 50-tih godina prošlog veka da bi od 70-tih godna do kraja prve decenije ovoga veka interesovanje prešlo na imperativne, posebno objektne jezik. Ideja funkcionalnog programiranja ponovo je aktuelizovana u prethodnih petnaestak godina.

Literatura uz poglavlje 2

[1] Eric Elliot, Composing Software: AnExploration of Functional Programming and Object Composition in JavaScript, Leanpub, 2019

[2] Chris Okasaki, Purely Functional Data Structures, Cambridge University Press, 1998.