FUNKCIJSKO PROGRAMIRANJE, IMUTABILNE KOLEKCIJE I BAZE PODATAKA

SAŽETAK

U ovom radu govorimo o primjeni "funkcijske paradigme" kod ažuriranja podataka u bazi podataka, tj. o tome da u bazi podataka radimo na sličan način kao što funkcijski jezici rade sa imutabilnim kolekcijama - bez UPDATE i DELETE naredbi (samo sa INSERT naredbom), kako je Jim Gray zagovarao još prije 40-ak godina.

Prikazujemo neke najvažnije osobine funkcijskog programiranja, neke osobine funkcijskog jezika Haskell i neke funkcijske osobine objektno-orijentiranih jezika Scala i Java, rad s imutabilnim kolekcijama, te realizaciju (2021. godine) blockchain i imutabilnih tablica u Oracle 19c DBMS.

Ključne riječi: funkcijsko programiranje, imutabilne kolekcije, imutabilne tablice (u bazi podataka)

ABSTRACT

In this paper, we are talking about the application of the "functional paradigm" when updating data in the database, i.e. about working in the database in a similar way as functional languages work with immutable collections - without UPDATE and DELETE commands (only with the INSERT command), as Jim Gray advocated some 40 years ago.

We show some of the most important features of functional programming, some features of the functional language Haskell and some features of the object-oriented languages Scala and Java, working with immutable collections, and the realization (in 2021) of blockchain and immutable tables in Oracle 19c DBMS.

Keywords: functional programming, immutable collections, immutable tables (in the database)

1. UVOD

U posljednjih nekoliko godina, funkcijsko programiranje steklo je veliku popularnost u odnosu na imperativno programiranje (napomena: objektno-orijentirano programiranje je također imperativno). Funkcijsko programiranje, kao i logičko programiranje i programiranje u SQL-u, visoko je deklarativno, tj. programom se iskazuje što se želi napraviti, a ne kako to napraviti. Funkcijsko programiranje nije novo. Prvi funkcijski jezik Lisp nastao je još 1958., godinu dana nakon jezika Fortran (prvi viši programski jezik) i godinu dana prije jezika COBOL.

U drugom poglavlju ukratko se podsjećamo radova Churcha i Turinga, koji su nezavisno dokazali Church-Turingov teorem, inspirirani jednim od 23 Hilbertova problema. Pritom su stvorili lambda račun (Church) i Turingov stroj (Turing), koji su teoretski ekvivalentni, ali Turingovi strojevi su po svom ponašanju bliži imperativnoj programskoj paradigmi, dok je lambda račun teoretska osnova za funkcijske programske jezike. Podsjetit ćemo se i Chomskyjeve hijerarhije jezika, gramatika i automata. Korištena je literatura [1], [3], [4], [6], [7], [9], [13], [14], [17], [18], [20], [21], [22].

Treće poglavlje daje kratak prikaz osnovnih osobina funkcijskog programiranja. Korištena je literatura [2], [8], [10], [11], [14], [15], [19], [23].

Četvrto poglavlje daje prikaz nekih funkcijskih osobina čistog funkcijskog jezika Haskell. Korištena je literatura [8], [19].

Peto poglavlje prikazuje neke funkcijske dodatke u objektno-orijentiranim jezicima Scala i Java. Korištena je literatura [11], [15], [23].

Glavna tema prikazana je u šestom poglavlju, razmišljanje o "funkcijskom radu" s podacima u bazi podataka, tj. o tome možemo li u bazi podataka raditi na sličan način kao što funkcijski jezici rade sa imutabilnim kolekcijama - bez UPDATE i DELETE naredbi (samo sa INSERT). Korištena je literatura [5], [15], [16].

U sedmom poglavlju prikazujemo blockchain tablice i imutabilne tablice, uvedene u Oracle 19c DBMS (2021. godine, u Oracle DBMS podverzijama 19.10 i 19.11). Korištena je literatura [12].

2. TURINGOV STROJ, CHURCHOV LAMBDA RAČUN, CHOMSKYJEVA HIJERARHIJA

Njemački matematičar David Hilbert održao je 1900. u Parizu govor na međunarodnom kongresu matematičara, u kojem je naveo 23 tada neriješena matematička problema. Npr. drugi problem (od 23) bio je: dokazati konzistentnost aritmetike realnih brojeva. Hilbert je 1928. definirao još dva velika problema: prvi je dokazati potpunost aksiomatskog sustava aritmetike prirodnih brojeva (Peanova aritmetika), a drugi je problem odlučivosti (njem. Entscheidungsproblem, engl. decision problem). Problem odlučivosti se može izraziti (i) ovako [3]: Postoji li općeniti algoritam za odlučivanje je li data izjava (u nekom formalnom sustavu) dokaziva iz aksioma korištenjem pravila logike prvog reda (Fregeova pravila).

Američki matematičar Alonzo Church i britanski matematičar Alan Turing 1936. su nezavisno dokazali da takav algoritam ne postoji. Budući da je Church publicirao svoj dokaz nešto prije Turinga, taj se teorem uobičajeno naziva Churchov teorem (dokaz teorema je npr. u [1] i [7]), ali se ponekad koristi i naziv Church-Turingov teorem. Postoje različite, međusobno ekvivalentne, formulacije tog teorema, a jedna od njih je i [21]: "Logika prvog reda je neodlučiva, tj. ne postoji test kojim bi se za svaku formulu u konačno mnogo koraka mogli ispitati je li valjana." Napomena: za razliku od logike prvog reda, logika sudova je odlučiva [21].

Church i Turing su prvo dokazali da ne postoji rješenje problema zaustavljanja (engl. halting problem) [3]. On se može izraziti na ovaj način: "Ako je dat bilo koji računalni program i bilo koji ulaz, nađimo (opći) algoritam koji će odrediti da li će program nad ulazom završiti, ili će raditi beskonačno." Onda su pokazali da negativan odgovor na to pitanje povlači i negativan odgovor na pitanje oko problema odlučivosti.

Ključni dio Turingovog dokaza za halting problem bila je definicija (apstraktnog) računala i programa, što je postalo poznato kao Turingov stroj. Napomena: Turingova nagrada (postoji od 1966.) je najprestižnije priznanje u računarstvu, ekvivalent Nobelove nagrade.

Alonzo Church je za dokaz istog teorema izumio lambda račun (engl. lambda calculus), prikazan npr. u [14] i [19]. Turing i Church su dokazali i to da su lambda račun i Turingov stroj ekvivalentni [3], tj. (pojednostavljeno, u današnjoj terminologiji): sve što se efektivno može izračunati, može se izračunati pomoću lambda računa ili Turingovog stroja. 1950-ih godina se formalno dokazalo da postoje i neki drugi sustavi koji su njima ekvivalentni: Gödelove rekurzivne funkcije, Postov sustav, Markovljevi algoritmi i dr.

Bez obzira na teoretsku ekvivalentnost, Turingovi strojevi su po svom ponašanju bliži imperativnoj programskoj paradigmi, dok je lambda račun teoretska osnova za funkcijske programske jezike (npr. Lisp, Haskell). Lambda izrazi (ili lambda funkcije) su u zadnjih dvadesetak godina uvedeni i u skoro sve objektno-orijentirane jezike, npr. u C++, Eiffel, Java, C#, Scala, Kotlin, Swift (napomena: redoslijed navođenja je od najstarijeg prema najmlađem jeziku, a ne redoslijed uvođenja funkcijskih osobina).

Poznata je Church-Turingova teza (naziva se i Churchova teza ili Turingova teza), koja se ne može matematički dokazati, već se smatra intuitivno prihvatljivom [18]. Pojednostavljeno, te u današnjoj terminologiji, ona kaže da sve što se efektivno može izračunati, može se izračunati pomoću lambda računa ili Turingovog stroja. Preciznija definicija data je u [22] (pogledati npr. i [4], [6], [9], [17]): "Svaka izračunljiva funkcija je parcijalno rekurzivna", a autor dalje navodi: "Pošto je pojam izračunljive funkcije intuitivan pojam, tj. nije strogo definiran, nemoguće je dati dokaz Churchove teze. Oboriti pak Churchovu tezu značilo bi odrediti funkciju za koju bi se svi složili da je izračunljiva, a istovremeno bi dokazali da nije parcijalno rekurzivna. No, to do sada nije učinjeno."

Američki lingvist i filozof Noam Chomsky 1950-ih godina prošlog stoljeća napravio je poznatu klasifikaciju jezika (ljudskih i formalnih). Chomskyjeva hijerarhija jezika postala je važna i u računarstvu, naročito u konstrukciji jezičnih procesora i teoriji automata. Ta je klasifikacija dala vezu između određenih jezika, gramatika koje ih opisuju i generiraju nizove znakova u tim jezicima, te (apstraktnih) automata koji prihvaćaju rečenice u tim jezicima.

Chomsky je izvorno dao tipove 0, 1, 2, 3 (kasnije su nađena još tri međutipa, koja nemaju brojeve). Tip 0 predstavlja jezik, gramatiku i automat s najvećim mogućnostima: rekurzivno prebrojiv jezik, gramatiku neograničenih produkcija i Turingov stroj. Nijedno računalo ne može riješiti problem koji ne može riješiti Turingov stroj (ili lambda račun). Slika 1. prikazuje tu klasifikaciju:

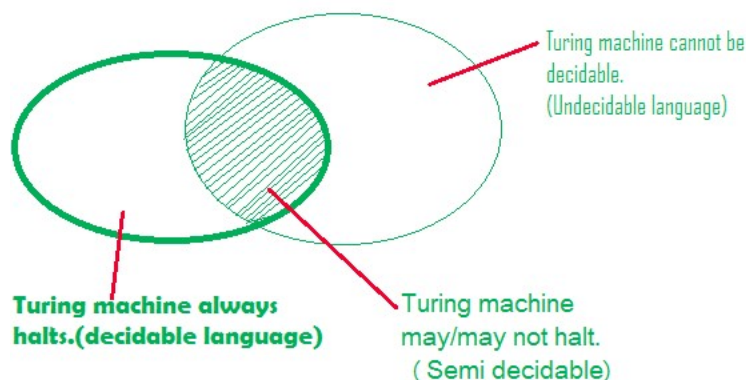
Teorija automata: formalni jezici i formalne gramatike			
Chomskyjeva hijerarhija	Gramatike	Jezici	Minimalni automat
Tip 0	Neograničenih produkcija	Rekurzivno prebrojiv	Turingov stroj
n/a	(nema uobičajenog imena)	Rekurzivni	Odlučitelj
Tip 1	Kontekstno ovisna	Kontekstno ovisni	Linearno ograničen
n/a	Indeksirana	Indeksirani	Ugniježđenog stoga
Tip 2	Kontekstno neovisna	Kontekstno neovisni	Nedeterministički potisni
n/a	Deterministička kontekstno neovisna	Deterministički kontekstno neovisni	Deterministički potisni
Tip 3	Regularna	Regularni	Konačni
Svaka kategorija jezika ili gramatika je pravi podskup nadređene kategorije.			

Slika 1. Teorija automata: formalni jezici i formalne gramatike; Izvor: [20]

Važno je napomenuti da postoji i širi skup od tipa 0, a to je skup svih mogućih jezika nad određenom abecedom (teoretski nije važno koja je abeceda, jer se svaka može svesti na binarnu abecedu, koja ima samo znakove 0 i 1). Taj širi skup jezika (koji je neprebrojivo beskonačan, kao i skup realnih brojeva, dok je skup rekurzivno prebrojivih jezika beskonačan, ali prebrojiv) nema odgovarajuće gramatike i automata.

Budući da se prepoznavanje i generiranje jezika može interpretirati kao rješavanje problema, postoje i problemi (i to neprebrojivo beskonačno njih) za koje ne postoje (opći) algoritmi (tj. za jezike ne postoje automati i gramatike). To su tzv. neodlučivi (undecidable) jezici i problemi.

Odlučiv (decidable) jezik je onaj kod kojeg Turingov stroj uvijek stane i odluči o prihvaćanju ili neprihvaćanju niza znakova (tog jezika). Rekurzivni jezici (tip jezika koji se nalazi između jezika tipa 0 i 1, tj. između rekurzivno prebrojivih i kontekstno ovisnih jezika) su uvijek odlučivi. No, rekurzivno prebrojivi jezici (tip 0), a koji nisu rekurzivni, nemaju Turingov stroj koji uvijek stane. Kod njih, ako niz pripada jeziku, Turingov stroj uvijek stane, ali ako niz ne pripada jeziku, moguće je da Turingov stroj ne stane. Dakle, rekurzivno prebrojivi jezici, a koji nisu rekurzivni, jesu poluodlučivi (semi-decidable).



Slika 2. Odlučivi, poluodlučivi i neodlučivi jezici / problemi; Izvor: [13]

3. FUNKCIJSKO PROGRAMIRANJE

Programske jezike moguće je klasificirati po različitim kriterijima. Npr. autor u [10] navodi desetak kriterija za klasifikaciju. Nama se čini najpogodnijom ova jednostavna klasifikacija iz [19]:

"Programski jezici su imperativni i deklarativni. Imperativni program izvodi se (slijednim) izvođenjem naredbi, dok se deklarativni program izvodi vrednovanjem izraza. Deklarativni jezici mogu biti logički (izrazi su relacije) ili funkcijski (izrazi su funkcije)."

Dodali bismo da je deklarativan jezik i SQL (koji se nekada nije smatrao Turing kompletnim programskim jezikom), koji ne spada u logičke ili funkcijske jezike, već je temeljen na "mješavini" relacijske algebre i relacijskog računa.

Kako je već rečeno, prvi funkcijski jezik Lisp nastao je još 1958. i još se koristi. Danas je vjerojatno najpoznatiji funkcijski jezik Haskell. U zadnjih dvadesetak godina povećao se interes za funkcijske jezike i funkcijsko programiranje općenito. Autor u [23] navodi tri razloga:

"Svako desetljeće ili dva, velika računalna ideja postane mainstream ...

Objektno-orijentirano programiranje (OOP), koje je izumljeno 1960-ih, postalo je glavni tok 1980-ih, vjerojatno kao odgovor na pojavu grafičkih korisničkih sučelja, u koje se OOP paradigma prirodno uklapa. Funkcijsko programiranje (FP) prolazi kroz sličan proboj. Dugo samo kao istraživačka tema, i zapravo mnogo starije od OOP-a, FP nudi učinkovite tehnike za tri glavna izazova našeg vremena:

1. ... paralelno programiranje sada je bitna vještina koju svaki programer mora savladati.

2. Potreba za pisanjem aplikacija usmjerenih na podatke (npr. 'Big Data') ...

3. Potreba za pisanjem aplikacija bez grešaka. Naravno, ovo je izazov star koliko i samo programiranje, ali FP nam daje nove alate, iz matematike, koji nas pokreću dalje u smjeru programa koji su dokazano bez bugova."

Definicije funkcijskog programiranja su uglavnom jednake ili vrlo slične. Npr. autor u [19] navodi:

"Funkcijsko programiranje je stil programiranja u kojemu se sve svodi na izračunavanje funkcija. Funkcija je građanin prvoga reda: ravnopravna je ostalim tipovima podataka te može biti povratna vrijednost ili argument druge funkcije."

Veća je razlika u popisu najvažnijih osobina koje bi trebao imati funkcijski jezik.

Navodimo prvo popis iz [2], u kojem se ne govori o nekom konkretnom programskom jeziku, nego o lambda računu, gdje se navode tri osnovna svojstva:

"Kada pričamo o teoriji funkcijskih jezika, podrazumijevaju se jezici sa sljedećim svojstvima:

- nepromjenjivost (engl. immutability) memorijskih lokacija, što znači da se ne može mijenjati vrijednost pojedinih varijabli jednom kada su inicijalizirane. Točnije, smatra se da su sve vrijednosti konstante, a jedini način za generiranje novih vrijednosti jest izračunavanje povratnih vrijednosti funkcija.

- čistoća (engl. purity) funkcija, koja garantira da funkcije neće utjecati na vrijednosti koje joj nisu direktno proslijeđene. ...

- korištenje funkcija višeg reda (engl. higher-order functions), tj. funkcija koje primaju druge funkcije kao argumente te vraćaju funkcije kao povratne vrijednosti."

U radu [2], navode se sljedeće osnovne osobine jezika Haskell, koji je čisti funkcijski jezik:

- komprehenzija listi (engl. list comprehension): "Jedan od najčešćih načina strukturiranja i manipuliranja podacima u računalstvu je korištenje listi vrijednosti. U tu svrhu, Haskell nudi liste kao osnovni koncept u jeziku, zajedno s jednostavnom, ali moćnom notacijom za konstruiranje nove liste odabirom i filtriranjem elemenata iz jednog ili više postojećih listi."

- rekurzivne funkcije: "Većina programa uključuje neki oblik petlje. U Haskellu, osnovni mehanizam koji zamjenjuje petlje su rekurzivne funkcije, koje su definirane u terminima sebe samih."

- funkcije višeg reda: "Haskell je funkcijski jezik višeg reda, što znači da funkcije mogu slobodno uzimati (druge) funkcije kao argumente i proizvoditi funkcije kao rezultate. Korištenje funkcija višeg reda omogućuje da se uobičajeni programski obrasci, kao što je kompozicija dviju funkcija, definiraju kao funkcije unutar samog jezika."

- generičke funkcije: "Većina jezika dopušta definiranje funkcija koje su generičke preko niza jednostavnih tipova, kao što su različiti tipovi brojeva. Međutim, Haskell također podržava funkcije koje su generičke nad mnogo bogatijim tipovima podataka."

- lijena evaluacija (engl. lazy evaluation): "Haskell programi se izvode koristeći tehniku koja se zove lijena evaluacija, a koja se temelji na ideji da se ne bi trebalo izvoditi nikakvo računanje dok njegov rezultat nije stvarno potreban."

- funkcije koje imaju (popratne) efekte (engl. effectful functions): "Funkcije u Haskell-u su čiste funkcije koje sve svoje ulaze uzimaju kao argumente i proizvode sve svoje izlaze kao rezultate. Međutim, mnogi programi zahtijevaju neki oblik nuspojava koje izgledaju u suprotnosti s čistoćom, kao što je čitanje unosa s tipkovnice ili pisanje izlaza na zaslon, dok je program pokrenut. Haskell pruža jedinstven okvir za programiranje s efektima, bez ugrožavanja čistoće funkcija, temeljen na korištenju monada i aplikativa."

Kako je već rečeno, danas su lambda izrazi (ili lambda funkcije) uvedeni u skoro sve objektno-orijentirane jezike (npr. u C++, Eiffel, Java, C#, Scala, Kotlin, Swift), ali su ti jezici dobili i druge funkcijske osobine. Vezano za Scalu, autori u [11] (od kojih je jedan autor kreator jezika Scala), uz prije navedene funkcijske osobine, spominju i druge, a neke ćemo prikazati u petom poglavlju. U nastavku slijede mišljenja autora koji pišu o funkcijskom programiranju u Javi, C++ i Eiffelu.

Autor knjige o funkcijskom programiranju u Javi [15] naglašava korist od korištenja imutabilnih kolekcija (ili čistih funkcijskih struktura podataka, engl. purely functional data structures), ali naglašava i to da je Java sa stanovišta funkcijskih osobina inferiornija ne samo od Scale (koju se smatra OOPL jezikom s najboljom podrškom za funkcijsko programiranje), nego i od jezika Kotlin (isti autor napisao je i knjigu o funkcijskom programiranju u Kotlinu, [16]).

Zanimljivo je što kaže autor knjige [2], koja govori o funkcijskom programiranju u jeziku C++:

"Uvijek me zabavljalo što većina programera kaže da je C++ objektno-orijentirani jezik. Razlog zašto je ovo zabavno je taj što se na vrlo malo mjesta u standardnoj C++ biblioteci (obično se naziva Standard Template Library ili STL) koristiti polimorfizam temeljen na nasljeđivanju, koji je u srcu OOP paradigme. STL je stvorio Alexander Stepanov, glasni kritičar OOP-a. Htio je stvoriti generičku programsku biblioteku, a to je učinio koristeći kombinirani sustav C++predložaka s nekoliko FP tehnika. Ovo je jedan od razloga zašto se dosta oslanjam na STL u ovoj knjizi - čak i ako to nije odgovarajuća FP biblioteka, modelira puno FP koncepata, što ga čini odličnom polaznom točkom za ulazak u svijet funkcijskog programiranja.“

Kreator programskog jezika Eiffel (OOPL koji isto podržava brojne funkcijske osobine) u [10] postavlja zanimljivo pitanje: “Zašto onda svi nisu prešli na funkcijsko programiranje?“, pa odgovara:

"Možemo uočiti tri značajna problema:

- brzina izvođenja: ...

- skalabilnost: ... Za strukturiranje velikih sustava, pojam klase prisutan u objektno-orijentiranim jezicima je učinkovitiji od pojma funkcije.

- bez promjene stanja (engl. statelessnes): ... neki aspekti računarstva temeljno zahtijevaju pojam stanja ... Funkcijski jezici, posebice Haskell, dodali su posebne mehanizme za rukovanje takvim imperativnim aspektima programiranja, ali oni nisu tako jednostavni kao osnovni funkcijski model."

Zatim nastavlja:

„U očima mnogih praktičara razvoja softvera, imperativna objektna tehnologija - korištena u ovoj knjizi i implementirana od strane mnogih najpopularnijih programskih jezika današnjice - daje bolji odgovor na kritične izazove razvoja softvera.

Ali nisu svi ovoga mišljenje, a svakom slučaju važno je razumjeti pojmove i primjene funkcijskog programiranja.“

Na kraju bismo mogli ponoviti najvažnije od navedenog u ovom poglavlju.

Kod deklarativnih jezika naglasak je na to ŠTO treba napraviti, a ne KAKO to napraviti.

Kod funkcijskog programiranja trebali bismo koristiti čiste funkcije. Jasno je da je nemoguće napraviti koristan softver koji bi se sastojao isključivo od čistih funkcija. Cilj je da većina funkcija bude čista, a da "nečiste radnje" stavimo u posebne funkcije (napomena: poznato je da u Oracle SQL upitu možemo koristiti PL/SQL ili Java funkcije, ali one moraju zadovoljavati neke uvjete, minimalno da ne koriste DML i ne mijenjaju pakirane varijable – to je onda jedna specifična vrsta čistih funkcija).

4. FUNKCIJSKI JEZIK HASKELL

Peter Landin je 1960-ih napravio ISWIM, prvi čisti funkcijski jezik strogo temeljen na lambda računu, bez naredbi pridruživanja. John Backus je 1970-ih napravio FP, funkcijski jezik koji je imao funkcije višeg reda, a Robin Milner je napravio ML, prvi moderni funkcijski jezik, koji je uveo zaključivanje o tipovima (type inference) i polimorfične tipove. 1970-ih i 1980-ih David Turner je izradio niz lijenih (lazy) funkcijskih jezika.

1987. je internacionalni komitet započeo na izradi jezika Haskell. 1990-ih je Phil Wadler kreirao klase tipova (type classes) i monade (monads), što je glavna inovacija koju je donio Haskell. 2003. je publiciran Haskell Report, koji je definirao prvu stabilnu verziju jezika Haskell. Ažurirana verzija publicirana je 2010. (Haskell 2010.). Sljedeća verzija trebala je biti Haskell 2020, ali se rad na novoj verziji odužio.

Pogledajmo primjer (iz [8], kao i ostali primjeri iz ovog dijela) definiranja dvije funkcije u Haskellu. Konvencija je da se prvo navede tip funkcije (u drugim jezicima bismo rekli deklaracija ili signatura), a onda definicija funkcije.

Prva funkcija služi za zbrajanje dva argumenta cjelobrojnog tipa i kao rezultat daje cjelobrojni tip:

```
add :: (Int, Int) -> Int
add (x, y) = x + y
```

Druga funkcija ima jedan argument cjelobrojnog tipa i kao rezultat daje niz cijelih brojeva od nula do zadanog broja:

```
zeroto :: Int -> [Int]
zeroto n = [0..n]
```

Zanimljiv je drugi način prikazivanja funkcija koje imaju dva ili više (ulaznih) argumenata (n-torku argumenata) - to su curryzirane funkcije (engl. curried functions), koje se tako zovu po prezimenu matematičara koji ih je izmislio (Haskell Curry).

Npr. funkcija add, koja je imala dva argumenta, može se napisati kao funkcija add' s jednim argumentom, koja vraća drugu funkciju:

```
add’ :: Int -> (Int -> Int)
add’ x y = x + y
```

Curryzirane funkcije su fleksibilnije, jer se parcijalnom primjenom argumenata često mogu napraviti druge korisne funkcije. Npr. iz add’ možemo ovako dobiti funkciju koja povećava (neki) broj za 1:

```
add’ 1 :: Int -> Int
```

Funkcijski jezici (kao i logički jezici) obilato koriste rekurziju, jer imaju optimizaciju kojom (često) mogu izbjeći dešavanje greške prelijevanja stoga (engl. stack overflow). Slijedi primjer rekurzivne funkcije koja ima jedan argument, a to je niz elemenata tipa a (koji je proizvoljan) i vraća niz u kojem je redoslijed elemenata obrnut:

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ []
```

I neki ne-funkcijski jezici imaju eliminaciju repnog poziva kod rekurzije (engl. tail call elimination ili tail call optimisation – TCE ili TCO), tj. onda kada je zadnja operacija u kodu ("na repu koda") poziv funkcije same. Tu mogućnost imaju npr. Scala, Kotlin, C++ (svi kompajleri), a nemaju npr. Java i Oracle PL/SQL ([2], [10], [11], [15], [16]).

Funkcijski jezici imaju i funkcije višeg reda (engl. higher order functions - HOF). Funkcije višeg reda su takve funkcije koje kao argument ili/i kao povratnu vrijednost imaju (neku drugu) funkciju. Budući da sve curryzirane funkcije imaju kao povratnu vrijednost (drugu) funkciju, najčešće se funkcijama višeg reda zovu samo one koje imaju (drugu) funkciju kao argument.

Slijedi Haskell primjer funkcije višeg reda map, koja prima dva argumenta (u curryziranom obliku). Prvi argument (funkcije map) je neka funkciju koja ima argument tipa a i vraća rezultat tipa b, a drugi argument (funkcije map) je niz čiji su elementi tipa a. Funkcija map vraća niz elemenata tipa b:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

> map (+1) [1, 3, 5, 7]
[2,4,6,8]
```

5. OBJEKTNI JEZICI SCALA I JAVA – NEKE NJIHOVE FUNKCIJSKE MOGUĆNOSTI

Programski jezik Scala kreirao je Martin Odersky, profesor na EPFL Lausanne. Nakon doktorata na ETH Zürich (kod profesora Niklausa Wirtha), bavio se istraživanjima u području funkcijskih jezika, zajedno sa kolegom Philom Wadlerom (jednim od dva glavna kreatora funkcijskog jezika Haskell).

Scala je čisti objektno-orijentirani jezik (sa statičkom provjerom tipova). Osim toga, na temelju objektno-orijentiranih mogućnosti izgrađene su i brojne funkcijske mogućnosti, tako da je Scala i funkcijski jezik (ali nije čisti). Sa funkcijskim osobinama došle su i neke osobine koje su vrlo pogodne za konkurentno programiranje. Scala je izvrstan jezik i za pisanje DSL-ova (Domain-Specific Language), jezika za specifičnu problemsku domenu. No, važno je da se može programirati u Scali bez napuštanja Java okoline, jer se Java i Scala programski kod mogu jako dobro upotpunjavati.

Scala primjer funkcije višeg reda - funkcija suma (iz [11], kao i ostali Scala primjeri iz ovog dijela):

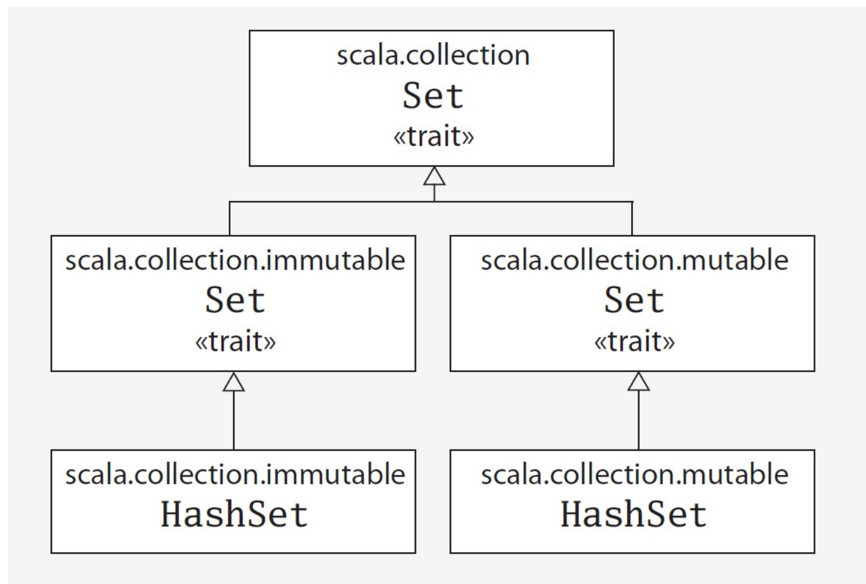
```
def suma(f: Int => Int, a: Int, b: Int): Int =
   if (a > b) 0 else f(a) + suma(f, a + 1, b) // rekurzija
```

Sada definiramo dvije funkcije i koristimo ih (za punjenje vrijednosti val varijabli) kao prvi parametar kod poziva funkcije suma:

```
def kvadrat(x: Int): Int = x * x

def dvaNaNtu(x: Int): Int =
   if (x == 0) 1 else 2 * dvaNaNtu(x - 1) // rekurzija

val sumaKvadrata = suma(kvadrat, 1, 5) // = 55

val sumaDvaNaNtu = suma(dvaNaNtu, 1, 5) // = 62
```

Funkcijski jezici poznati su po tome da imaju izvrstan način rada sa memorijskim kolekcijama podataka, naročito sa listama (list). U Scali, temeljne kolekcije su List, Set i Map. List je uređena kolekcija objekata, Set je neuređena kolekcija objekata, a Map je skup parova (ključ, vrijednost).

Kako prikazuje slika 3. (na primjeru kolekcije Set), Scala ima dvije vrste kolekcija – standardne (mutabilne) i funkcijske (imutabilne):



Slika 3. Kolekcije u jeziku Scala javljaju se kao mutabilne i imutabilne; Izvor: [11]

Funkcijski jezici imaju lijenu evaluaciju (engl. lazy evaluation), kod koje se inicijalizacija vrijednosti varijable odgađa do trenutka kada se (lijena) vrijednost prvi put koristi. Scala je jedan od rijetkih objektno-orijentiranih jezika koji ima lijenu evaluaciju. Za razliku od funkcijskog jezika Haskell, u Scali to treba eksplicitno navesti pomoću ključne riječi lazy:

```
class Radnik (id: Int, ime: String, managerId: Int) {
  // bez lijene evaluacije
  val manager: Radnik = Db.get(managerId)
  val tim: List[Radnik] = Db.tim(id)
}

class Radnik (id: Int, ime: String, managerId: Int) {
  // s lijenom evaluacijom
  lazy val manager: Radnik = Db.get(managerId)
  lazy val tim: List[Radnik] = Db.tim(id)
}
```

Java je dobila neke funkcijske osobine u verziji 8 [15]. Na temelju onoga što čitamo i čujemo, mogli bismo zaključiti da je najvažnija nova mogućnost u Javi 8 lambda izraz (ili kraće, lambda). Inače, lambda izraz je (u Javi) naziv za metodu bez imena. U pravilu je ta metoda funkcija, a ne procedura. Zato možemo reći i da lambda izraz je anonimna funkcija, koja se može javiti kao parametar (ili povratna vrijednost) druge funkcije (koja je, onda, funkcija višeg reda).

Java 8 lambda (izrazi) temelje se na tzv. funkcijskim sučeljima (engl. functional interface), koji su postojali od početka. Funkcijska sučelja su ona sučelja koja imaju točno jednu (jednu i samo jednu) apstraktnu funkciju. No, od Jave 8 funkcijska sučelja mogu imati i statičke metode i default metode (koje nisu postojale prije Jave 8).

Npr. kad Java kompajler naiđe na ovakvu naredbu (lambda izraz je desno od znaka jednakosti; ovo je samo jedna od brojnih varijanti pisanja lambda izraza):

```
StringToIntMapper mapper = (String str) -> str.length();
```

kompajler provjerava da li postoji odgovarajuće sučelje StringToIntMapper, koje ima samo jednu apstraktnu funkciju.

No u Javi 8 pojavile su se i tzv. default metode u Java sučeljima. One, zapravo, predstavljaju uvođenje višestrukog nasljeđivanja implementacije u Javu.

Npr. ako bismo postojeće sučelje Iterable htjeli proširiti sa novom metodom forEach, morali bismo mijenjati svaku klasu koja (direktno ili indirektno) nasljeđuje to sučelje:

```
public interface Iterable<T> {
  public Iterator<T> iterator();

  public void forEach(Consumer<? super T> consumer);
}
```

Default metode rješavaju taj problem:

```
public interface Iterable<T> {
  public r<T> iterator();

  public default void forEach(Consumer<? super T> consumer) {
    for (T t : this) {
      consumer.accept(t);
    }
  }
}
```

Međutim, lambda izrazi i default metode su, na neki način, posljedica uvođenja treće važne mogućnosti u Javi 8, a to su Streamsi, koji nadograđuju dotadašnje Java kolekcije. Pojava masivno višejezgrenih procesora traži bolji i lakši način izrade paralelnih programa od dosadašnjih načina. Jedan (dobar) način je da se koristi funkcijsko programiranje, a naročito paralelne kolekcije. Java nema paralelne kolekcije (collections), ali su zato uvedeni Streamsi, kako bi se postojeće kolekcije "zaogrnule" u (paralelne) Streamse i mogle (indirektno) paralelizirati. Za razliku od dosadašnjih kolekcija, koje sadrže sve svoje elemente u memoriji, Streamse možemo shvatiti kao "vremenske kolekcije", čiji se elementi (kojih teoretski može biti i beskonačan broj) stvaraju po potrebi.

Navodimo neka naša razmišljanja o tome kako bi bilo dobro da Oracle uvede neke funkcijske mogućnosti u svoj programski jezik PL/SQL, koje danas imaju svi funkcijski jezici i skoro svi objektno-orijentirani jezici (npr. C++, Eiffel, Java, C#, Scala, Kotlin, Swift).
- funkcije višeg reda (kod kojih argument može biti druga funkcija)
- lambda izrazi (anonimne funkcije)
- mogućnost razlikovanja varijabli koje se mogu mijenjati samo jednom (val) ili više puta (var).
- eliminacija (ili optimizacija) repnog poziva (TCE ili TCO); no niti Java to još nema.

6. IMUTABILNE KOLEKCIJE – MOŽEMO LI TAKO RADITI I S BAZAMA PODATAKA?

Jim Gray, jedan od najvećih stručnjaka na području baza podataka (posebno transakcija), napisao je sljedeće [5]:

"UPDATE IN PLACE: A poison apple?

When bookkeeping was done with clay tablets or paper and ink, accountants developed some clear rules about good accounting practices.

One of the cardinal rules is double-entry bookkeeping so that calculations are self checking, thereby making them failfast.

A second rule is that one never alters the books; if an error is made, it is annotated and a new compensating entry is made in the books. The books are thus a complete history of the transactions of the business…

Update-in-place strikes many systems designers as a cardinal sin: it violates traditional accounting practices which have been observed for hundreds of years."

Napomena: Benedikt Kotruljević (Dubrovnik, oko 1400.-1468., talijanski Benedetto Cotrugli, latinski Benedictus de Cotrullis) smatra se izumiteljem dvojnog knjigovodstva.

Može se postaviti pitanje možemo li zaista primijeniti metode, koje su poznate u knjigovodstvu skoro 600 godina, na kolekcije podataka. Naime, kada dvije softverske dretve (ili više njih) mijenja kolekciju podataka na način da se radi direktno mijenjanje podataka (engl. in-place mutation) tj., tada [15]:

- nećemo koristiti zaključavanje, pa će doći do nekonzistentnosti
- ili ćemo koristiti zaključavanje, i smanjiti paralelnost rada.

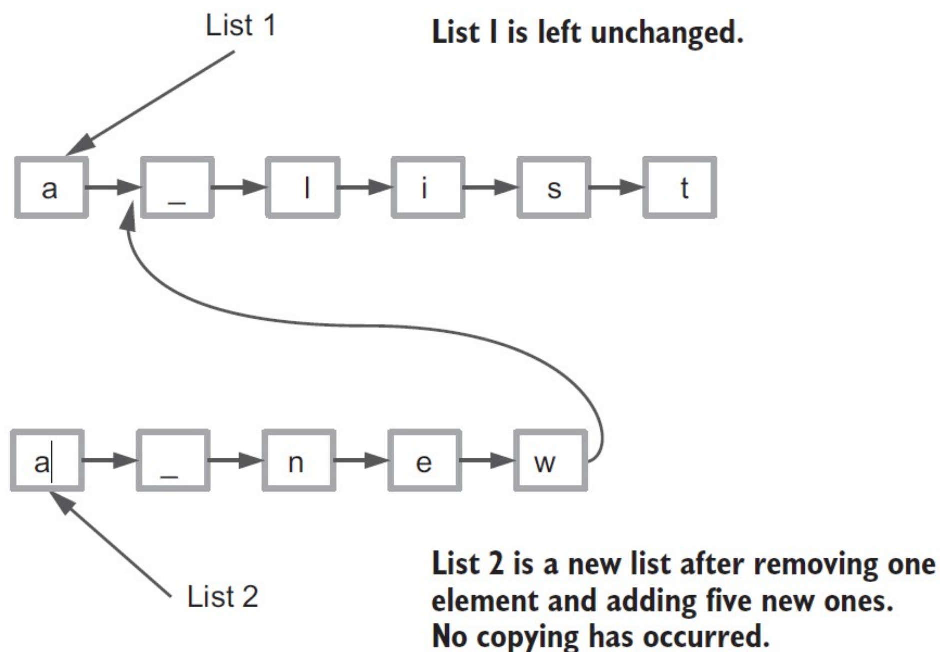
Rješenje (ne uvijek) je u tome da se izbjegne direktno mijenjanje podataka, tako da se, kao kod dvostrukog knjigovodstva, ne mijenjaju postojeći podaci, nego se uvijek stvaraju novi. Umjesto da mijenjamo postojeći element, stvaramo novi. Umjesto da dodajemo element u postojeću kolekciju, stvaramo novu kolekciju s dodanim elementom.

Programeri koji su navikli na imperativno programiranje isprva ostanu začuđeni kada čuju za takav način rada. Uobičajene (dvije) primjedbe na to su [15]:

- ako jedna dretva stvara svoju kopiju podataka, i ne vidi tuđe izmjene, to je loše
- ovakav način rada ne traži jako velike memorijske resurse, jer svaka dretva radi kopiju svojih podataka.

Odgovor na prvu primjedbu je: dretve i ne bi trebale vidjeti privremene podatke, koji se ne bi vidjeli u serijskom načinu rada (napomena: kod baza podataka uobičajeno je da transakcija ne vidi rezultate druge transakcije, dok ta druga ne napravi COMMIT).

Odgovor na drugu primjedbu pitanje: najčešće ne treba raditi kopiju svih podataka, nego samo dijela, kako pokazuje slika 4.



Slika 4. Primjer brisanja 1. elementa i dodavanja 5 novih elemenata u imutabilnu listu; Izvor: [15]

Možemo postaviti pitanje da li se s podacima u bazi podataka može raditi nešto slično kao sa imutabilnim listama kod programskih jezika. Iako to nije u potpunosti isto, u bazama podataka postoji sličan način rada, koji se svodi na to da ne koristimo (SQL) UPDATE i DELETE naredbe, nego samo naredbu INSERT.

Umjesto da mijenjamo postojeći redak (UPDATE), unesemo novi (INSERT), koji će "zamijeniti" prethodni redak. Naravno, mora postojati stupac, ili više njih, koji će označavati da je to najnoviji redak.

Umjesto da brišemo postojeći redak (DELETE), unesemo novi, koji će sadržavati informaciju da je taj redak (i njegove prethodne verzije) izbrisan. Na fizičkoj razini ovo nije ništa novo, jer DELETE naredba na fizičkoj razini ne briše redak, već ga označava slobodnim za upis novog retka.

Prednost takvog načina rada je (i) u tome što bi podaci bili rjeđe zaključani (napomena: INSERT isto radi zaključavanje, zbog provjere primarnog ključa ili jedinstvenog ključa).

Naravno, može se izreći i puno primjedbi, npr.:

- tako nam treba puno više prostora za bazu podataka, jer se sve pamti (protuprimjedba: današnji su diskovi sve veći, a danas neke regulative i traže da se pamti sve)
- danas skoro sve baze (Oracle od početka) rade tako da DML naredbe ne zaključavaju podatke koje se čitaju (npr. Oracle od početka ima multiversion consistency model rada s transakcijama)
- ako želimo pamtiti sve podatke, u Oracle bazi možemo koristiti flashback tehnologije (Flashback Data Archive, Flashback Query ...), koje pamte sve (ili neke) izmjene
- rad samo sa INSERT naredbom bi bio jako težak (protuprimjedba: možda se samo treba naviknuti; i funkcijsko programiranje je vrlo neobično onima koji su radili samo imperativno programiranje).

7. BLOCKCHAIN TABLICE I IMUTABILNE TABLICE U ORACLE DBMS 19c

Oracle je od baze 18c (izašla je 2018.) uveo označavanje prema godinama i uveo je pojmove Long Term Release (LTR) i Innovation Release ili non-LTR (napomena: Java 19 ne predstavlja godinu i LTS u Javi znači Long-Term Support). Sve baze do 12c, baza 19c i buduća baza 23c su LTR. Baze 18c i 21c su non-LTR.

Oracle je početkom 2021. uveo blockchain tablice u bazi 21c, a onda ih je uključio i u verziju 19c, u 19.10 (napomena: 19 je godina, a 10 nije mjesec, nego redni broj). Nekoliko mjeseci kasnije, Oracle je u bazu 21c uveo imutabilne tablice (engl. immutable tables), a onda ih uključio i u verziju 19c, u 19.11.

Blockchain tablice štite podatke koji bilježe važne radnje, imovinu, entitete i dokumente od neovlaštene izmjene ili brisanja [12]. Blockchain tablice sprečavaju neovlaštene promjene napravljene pomoću baze podataka i otkrivaju neovlaštene promjene koje zaobilaze bazu podataka.

Blockchain tablice su samo za čitanje (insert-only), i retke organiziraju u više lanaca. Svaki redak u lancu, osim prvog retka, lančano je povezan s prethodnim retkom u lancu pomoću kriptografskog hash-a.

Primjer kreiranja blockchain tablice koja se može brisati (DROP) nakon 1000 dana neaktivnosti (NO DROP znači da se ne može brisati), a njeni redci se mogu brisati 1000 dana nakon što su uneseni (NO DELETE znači da se ne mogu brisati):

```
CREATE BLOCKCHAIN TABLE bank_ledger
  (bank VARCHAR2(128), deposit_date DATE, deposit_amount NUMBER)
  NO DROP UNTIL 1000 DAYS IDLE – ili NO DROP za potpunu zabranu
  NO DELETE UNTIL 366 DAYS AFTER INSERT – ili NO DELETE
  HASHING USING "SHA2_512" VERSION "v1";
```

Imutabilne tablice pružaju zaštitu od neovlaštene izmjene podataka. Imutabilne tablice su tablice samo za čitanje (insert-only) i sprečavaju neovlaštene izmjene podataka od strane insajdera i slučajne izmjene podataka koje su rezultat ljudskih pogrešaka [12].

Novi redci mogu se dodati u imutabilnu tablicu, ali se postojeći redci ne mogu mijenjati. Redci postaju zastarjeli nakon zadanog razdoblja zadržavanja i tada se mogu brisati. Korištenje imutabilnih tablica ne zahtijeva izmjene postojećih aplikacija.

Primjer kreiranja imutabilne tablice koja se može brisati (DROP) nakon 1000 dana neaktivnosti (NO DROP znači da se ne može brisati), a njeni redci se mogu brisati 1000 dana nakon što su uneseni (NO DELETE znači da se ne mogu brisati):

```
CREATE IMMUTABLE TABLE trade_ledger
  (id NUMBER, ledger_user VARCHAR2(40), value NUMBER)
  NO DROP UNTIL 1000 DAYS IDLE
  NO DELETE UNTIL 366 DAYS AFTER INSERT;
```

Kako pokazuje tablica 1, iako su nastale malo kasnije, Oracle imutabilne tablice su manjih mogućnosti nego blockchain tablice. Između ostalog, za razliku od imutabilnih tablica, blockchain tablice otkrivaju neovlaštene promjene koje zaobilaze bazu podataka. Naravno, blockchain tablice i imutabilne tablice koriste se u različite svrhe.

Tablica 1. Razlike između imutabilnih i blockchain tablica; Izvor: [12]

Imutabilne tablice	Blockchain tablice
Imutabilne tablice sprječavaju neovlaštene promjene od strane lažnih ili kompromitiranih insajdera koji imaju pristup korisničkim vjerodajnicama.	Osim što sprečavaju neovlaštene promjene od lažnih ili kompromitiranih insajdera, blockchain tablice pružaju sljedeće funkcije: - otkrivaju neovlaštene promjene koje su napravljene zaobilazeći Oracle Database softver - otkrivaju lažno predstavljanje kao legalni krajnji korisnik i ažuriranje podataka u ime legalnog korisnika, ali bez njegovog ovlaštenja - sprečavaju neovlašteno mijenjanje podataka i osiguravaju da su podaci zaista legalno uneseni u tablicu.
Redci nisu lančano povezani.	Svaki redak, osim prvog, vezan je za prethodni redak pomoću kriptografske hash funkcije. Hash vrijednost tekućeg retka izračunava se na temelju podataka tekućeg retka i hash vrijednost prethodnog retka u lancu. Svaka izmjena retka prekida lanac, što ukazuje na to da se redak htjelo (vjerojatno) neovlašteno mijenjati.
Ažuriranje redaka ne zahtijeva dodatne radnje kod commit procesiranja.	Potrebno je dodatno vrijeme kod commit procesiranja, kako bi se ulančali redci.

Međutim, imutabilne tablice imaju i prednosti. Jednostavnije su, imaju manje restrikcija za korištenje i ne traže dodatno procesorsko vrijeme za insert podataka, dok je kod blockchain tablica potrebno dodatno vrijeme za ulančavanje blokova (za vrijeme commit procesiranja).

Imutabilne tablice slične su imutabilnim kolekcijama, i u duhu su razmišljanja o kojima je govorio (i) Jim Gray.

8. ZAKLJUČAK

Zadnjih nekoliko godina, funkcijsko programiranje stječe veliku popularnost u odnosu na imperativno programiranje. Ne samo da postoje vrlo efikasni čisti funkcijski jezici (najpoznatiji je Haskell), nego su i skoro svi značajni objektno-orijentirani jezici (npr. C++, Eiffel, Java, C#, Scala, Kotlin, Swift) dobili barem neke funkcijske osobine.

Jedna od značajnih osobina funkcijskog programiranja je rad s imutabilnim kolekcijama, kod kojih postojeće kolekcije ne mijenjamo direktno (engl. in-place mutation), nego (teoretski) uvijek stvaramo novu kolekciju, bez izmjene stare kolekcije. Zapravo, ne moramo uvijek kopirati cjelokupnu staru kolekciju u novu, jer često možemo iskoristiti dijelove stare kolekcije i time poboljšati performanse.

Jim Gray, jedan od najvećih stručnjaka na području baza podataka (posebno transakcija), naglasio je u [5] još prije 40-ak godina da bi baze podataka trebale raditi na način koji je poznat u (dvojnom) knjigovodstvu od sredine 15. stoljeća, gdje se knjigovodstveni podaci ne mijenjaju direktno, već se ispravak knjiženja radi tako da se kreiraju novi zapisi.

Kod našeg rada s bazama podataka (uglavnom Oracle DBMS), zadnjih desetak godina smo se često pitali bismo li u praksi zaista mogli efikasno raditi bez korištenja naredbi UPDATE i DELETE (samo sa INSERT). Probali smo neka testiranja, ali nismo u praksi otišli u tom smjeru.

Međutim, Oracle je početkom 2021. uveo (u kratkotrajnu bazu 21c, pa onda i u dugotrajnu bazu 19c) blockchain tablice, a nekoliko mjeseci kasnije i imutabilne tablice (koje su, zapravo, jednostavnije od blockchain tablica). Čini nam se da će to potaknuti (i kod nas i kod drugih) rad s bazama podataka na način koji je zagovarao Jim Gray.

Literatura:

[1] M. Ben-Ari (2012): Mathematical Logic for Computer Science (3. izdanje), Springer
[2] I. Čukić (2019): Functional programming in C++, Manning
[3] M. Davis (2001): Engines of Logic: Mathematicians and the Origin of the Computer, W. W. Norton & Company
[4] G. Gopalakrishnan (2006): Computation Engineering - Applied Automata Theory and Logic, Springer
[5] J. Gray (1981): The Transaction Concept: Virtues and Limitations, TANDEM COMPUTERS Technical Report 81.3, dostupno na https://www.hpl.hp.com/techreports/tandem/TR-81.3.pdf (10.03.2023.)
[6] S. Hedman (2006): A First Course in Logic, Oxford University Press
[7] M. Huth, M. Ryan (2004): Logic in Computer Science, Cambridge University Press
[8] G. Hutton (2016): Programming in Haskell (2. izdanje), Cambridge University Press
[9] J.C. Martin (2011): Introduction to Languages and the Theory of Computation (4. izdanje), McGraw-Hill
[10] B. Meyer (2009): Touch of Class - Learning to Program Well with Objects and Contracts, Springer
[11] M. Odersky, L. Spoon, B. Venners (2016): Programming in Scala (3. izdanje), Artima Press
[12] Oracle priručnik (2022): Oracle Database Administrator's Guide 19c (priručnik E96348-16)
[13] Relation between decidability, semidecidability and undecidability, dostupno na https://www.geeksforgeeks.org/undecidability-and-reducibility-in-toc/ (10.03.2023.)
[14] L. Rožić, J. Šnajder, M. Vuković (2016): Lambda račun kao osnova funkcijskog programiranja, Hrvatski matematički elektronički časopis, Vol. 29 No. 1, 2016.
[15] P-Y. Saumont (2017): Functional Programming in Java, Manning
[16] P-Y. Saumont (2019): The Joy of Kotlin, Manning
[17] S. Skansi (2020): Logika i dokazi, Element, Zagreb
[18] S. Srbljić (2007): Uvod u teoriju računarstva, Element, Zagreb
[19] J. Šnajder (2012): Funkcijsko programiranje i programski jezik Haskell, materijal kolegija Programske paradigme i jezici, FER, Zagreb
[20] Teorija automata, dostupno na https://hr.wikipedia.org/wiki/Teorija_automata (10.03.2023.)
[21] M. Vuković (2007): Matematička logika 1 (4. izdanje), skripta Sveučilišta u Zagrebu, PMF-Matematički odjel
[22] M. Vuković (2009): Izračunljivost, skripta Sveučilišta u Zagrebu, PMF-Matematički odjel
[23] D. Wampler, A. Payne (2014): Programming Scala (2. izdanje), O'Reilly Media

Podaci o autoru:

Zlatko Sirotić, univ.spec.inf.
ISTRA TECH d.o.o., Pula
e-mail: zlatko.sirotic@istratech.hr

Autor radi na informatičkim poslovima od 1984. godine, uglavnom u poduzeću ISTRA TECH d.o.o., Pula (ISTRA TECH je novo ime poduzeća Istra informatički inženjering, osnovanog 1990. godine). Oracle softverske alate (baza, Designer CASE, Forms 4GL, Reports, JDeveloper IDE, Java) koristi oko 25 godina. Objavljivao je stručne radove na kongresima / konferencijama CASE, KOM, HrOUG, JavaCro, "Hotelska kuća", u časopisima "Mreža", "InfoTrend" i "Ugostiteljstvo i turizam", a neka njegova programska rješenja objavljivana su na web stranicama firmi Oracle i Quest. Na Fakultetu informatike u Puli sudjeluje (od početka osnivanja, 2011.) kao vanjski suradnik - predavač, uglavnom na kolegijima Baze podataka 2 i Praktikum.