FUNKCIJSKA PARADIGMA I BAZE PODATAKA

Zlatko Sirotić, univ.spec.inf.

ISTRA TECH d.o.o.

Pula

Neki izabrani (stručni) radovi

- HrOUG 2018: Testiranje konkurentnih transakcija
- **HrOUG 2015a: Povratak u Prolog (verzija 1)**
- HrOUG 2015b: Kada Oracle naredba nije serijabilna?
- HrOUG 2014: Nasljeđivanje je dobro, naročito višestruko - Eiffel, C++, Scala, Java 8
- **HrOUG 2013: Transakcije i Oracle - baza, Forms, ADF**
- HrOUG 2012: Ima neka loša veza (priča o in-doubt distribuiranim transakcijama)

http://www.istratech.hr/category/blog/

- **Mrav i med na prizmi**
- Povratak u Prolog (verzija 2)
- Strukturna složenost algoritama

Uvod

- Zadnjih nekoliko godina, **funkcijsko programiranje** stječe veliku popularnost u odnosu na imperativno programiranje (napomena: objektno-orijentirano programiranje je također imperativno). Npr. 2014. godine je i Java jezik dobio neke značajne funkcijske mogućnosti (verzija Java 8).
- Funkcijsko programiranje nije novo. **Prvi funkcijski jezik Lisp nastao je davne 1958. godine**, godinu dana nakon jezika Fortran i godinu dana prije jezika COBOL.
- Funkcijsko programiranje, kao i logičko programiranje, po nečemu je vrlo slično programiranju u SQL-u - **visoko je deklarativno**.
- U prezentaciji ćemo se osvrnuti na funkcijsko i logičko programiranje, dati par ideja za proširenje jezika PL/SQL nekim funkcijskim mogućnostima, te upitati se da li bi se rad s imutabilnim kolekcijama mogao "imitirati" kod baza podataka.

Teme

- Chomskyjeva hijerarhija jezika
- Logičko programiranje i logički jezik Prolog
- Lambda račun i funkcijsko programiranje
- Funkcijski jezik Haskell
- Objektno-funkcijski jezik Scala
- Funkcijska proširenja u jezicima Java i Kotlin
- Ideje za uvođenje nekih funkcijskih osobina u PL/SQL
- Razmišljanje o "funkcijskom radu" s podacima u bazi podataka (te ideje nisu nove, zagovarao ih je još 80-tih Jim Gray, znanstvenik na području baza podataka i transakcijskih sustava, dobitnik Turingove nagrade 1998. godine, ali su u zadnje vrijeme ponovno "moderne")

Chomskyjeva hijerarhija jezika

- Kako kaže biolog i matematičar M.Nowak, (ljudski) jezik je najvažnija invencija prirode nakon što su se prije oko 600 milijuna godina pojavili prvi razvijeni višestanični organizmi. (Prije toga važna je bila pojava prokariota, prije oko 3500 milijuna godina, i eukariota, prije oko 1500 milijuna godina.)
- Slavni američki lingvist i filozof (ali i neumorni politički aktivist) **Noam Chomsky** još je 50-ih godina prošlog stoljeća napravio poznatu klasifikaciju jezika (ljudskih i formalnih).
- Chomskyjeva je hijerarhija jezika postala važna 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.**

Hijerarhija (formalnih) jezika / gramatika / automata

- Chomsky je izvorno dao tipove 0, 1, 2, 3 (kasnije su nađena tri međutipa, koja nemaju brojeve).
- Tip 0 predstavlja najjači jezik, gramatiku i automat. Dakle, **Turingov stroj** je najjači automat. Nijedno računalo ne može riješiti problem koji ne može riješiti Turingov stroj.

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.			

Logičko programiranje i AI

- Primjena "umjetne inteligencije" (engl. skraćenica AI) u informatici ima dugu povijest, još od 50-ih godina 20. stoljeća.
- **Lisp** (LISt Processor) je **funkcijski programski jezik**, koji se intenzivno koristi(o) u AI (uglavnom u SAD), a specificiran je 1958. Od jezika koji se i danas intenzivno koriste, samo Fortran je stariji (1954.-57.).
- Izvan SAD-a se u AI području uglavnom koristio **logički jezik Prolog** (PROgrammming in LOGic), specificiran 1970.
- Temelj za Prolog je (matematička) **logika prvog reda** (first-order logic; **logika sudova** je njen podskup), iako podržava i neke predikate drugog reda (npr. setof i bugof). Visoko je deklarativan (u tom smislu usporediv sa SQL-om).
- IBM-ov AI računalni sustav (hardver i softver) **Watson** pobijedio je 2011. godine ljudske prvake u kvizu "Jeopardy!". Softver je pisan većinom u jezicima **C++, Java i Prolog**.

Prolog - program

- Program je skup klauzula oblika **glava :- tijelo.**
  Koristi se **:-** umjesto ← i obavezna je točka na kraju.
- **Klauzula-pravilo** (rule clause) ima oba dijela, **činjenica** (fact) ima samo glavu, a **cilj ili upit** (goal) samo tijelo.
- Varijable se u Prologu pišu velikim početnim slovima.
  Proceduru čine klauzule s istom glavom (ovdje su dvije):

```
ancestor(X, Y) :- parent(X, Y). %X je predak od Y
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y). %rekurzija
parent(bob, allen). %bob je allenov roditelj
parent(catherine, allen).
parent(dave, bob).
parent(ellen, bob).
parent(fred, bob).
parent(harry, george).
parent(ida, harry).
parent(joe, harry).
```

Prolog – postavljanje upita

- Primjeri postavljanja upita (klauzula-ciljeva), nad prethodnim programom:

```
?- parent(bob, allen). %Da li je bob alenov roditelj?
true.
?- ancestor(allen, bob). %Da li je allen bobov predak?
false.

?- parent(X, allen). %Tko je allenov roditelj?
X = bob ; %znak ; mi kucamo - tražimo sljedeći odgovor
X = catherine. %točka – sustav zna da više nema odgovora

?- ancestor(ellen, X). % Kome je ellen predak?
X = bob ;
X = allen ; %sustav još ne zna da nema više odgovora
false.
```

Prolog - dio (našeg) programa za (algebarsko) deriviranje

```
d(F1 * F2, X, R) :-
    d(F1, X, R1),
    d(F2, X, R2),
    R = R1 * F2 + F1 * R2, !.

d(F1 / F2, X, R) :-
    d(F1, X, R1),
    d(F2, X, R2),
    R = (R1 * F2 - F1 * R2) / F2 ^ 2, !.
```

Lambda račun

- Lambda izrazi (ili lambda funkcije), imaju teoretsko porijeklo u **lambda računu** (lambda calculus), kojega je sredinom 30-ih godina prošlog stoljeća kreirao **Alonzo Church**.
- Poznata je **Church-Turingova hipoteza** (inače, **Alen Turing** je kod Alonza Churcha radio doktorat), koja se ne može matematički dokazati, već se smatra intuitivno prihvatljivom. Ona (pojednostavljeno, te u današnjoj terminologiji) kaže da sve što se efektivno može izračunati, može se izračunati pomoću lambda računa ili **Turingovog stroja**.
- 50-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.

Funkcijsko programiranje

- Programske jezike moglo bi se grubo podijeliti na **imperativne** (objektne i ne-objektne) i **deklarativne**.
- Kod imperativnih jezika naglasak je: KAKO nešto napraviti, a kod deklarativnih: ŠTO treba napraviti.
- Deklarativni jezici su najčešće funkcijski jezici ili logički jezici (npr. Prolog), ali i SQL (koji nije niti funkcijski, niti logički).
- Kako je već rečeno, prvi funkcijski jezik je Lisp (i još se koristi). Danas je najpoznatiji **Haskell**.
- U zadnjih nekoliko godina se **povećao interes za funkcijske jezike,** naročito zbog toga što se drži da su funkcijski jezici bolji za konkurentno i paralelno programiranje.
- **Većina nas nije vična funkcijskom programiranju**. Bez obzira na količinu iskustva u imperativnom programiranju, suočavanje sa funkcijskim programiranjem predstavlja izazov, jer traži promjenu načina razmišljanja.

Osobine funkcijskih jezika

- Funkcije višeg reda (higher-order functions)
- Leksičko zatvaranje (lexical closure)
- Podudaranje (sparivanje) uzorka (pattern matching)
- Jednokratno pridruživanje (single assignment)
- Lijena evaluacija (lazy evaluation)
- Zaključivanje o tipovima (type inference)
- Eliminacija repnog poziva (tail call elimination, TCE)
- Razumijevanje listi (list comprehension): kompaktan i ekspresivan način definiranja listi i rada s listama, kao osnovnih podatkovnih struktura funkcijskog programa
- Monadički efekti (monadic effects)

Čiste funkcije

- Kod funkcijskog programiranja, najviše se govori o tome da bismo trebali (što više) koristiti **čiste funkcije** (pure functions), a s time je povezan i izraz **referencijalna transparentnost** (referential transparency).
- Čiste funkcije:
  - **za iste argumente uvijek daju istu povratnu vrijednost**
  - **nemaju vanjske efekte**; vanjski efekti su npr. mijenjanje neke globalne varijable (vidljive izvan funkcije), rad s bazom podataka, čitanje s tipkovnice, pisanje na ekran …
- 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 funkcije, ali one moraju zadovoljavati neke uvjete, minimalno da ne koriste DML i ne mijenjaju pakirane varijable.

Haskell – kratka povijest

- 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 70-ih je izradio niz lijenih (lazy) funkcijskih jezika, završno sa sustavom **Miranda**.
- 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 bit će Haskell 2020.

Haskell - definiranje funkcija

- Konvencija je da se prvo navede **tip funkcije** (u drugim jezicima bismo rekli deklaracija ili signatura), a onda **definicija funkcije**:

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

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

Haskell - curried functions

- Drugi način prikazivanja funkcija koje imaju dva ili više ulaznih argumenata (n-torku argumenata) su **curryzirane funkcije** (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 pretvoriti u funkciju 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
```

Haskell - rekurzivne funkcije

- Funkcijski jezici obilato koriste rekurziju, jer imaju **optimizaciju** kojom (često) mogu izbjeći dešavanje greške prelijevanja stoga (**stack overflow**):

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

- I neki ne-funkcijski jezici imaju **eliminaciju repnog poziva kod rekurzije** (tail call elimination ili tail call optimisation), 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, PL/SQL.

Haskell - funkcije višeg reda

- **Funkcije višeg reda** (higher order functions - HOF) 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.

```
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]
```

Scala – kratka povijest

- Programski jezik Scala kreirao je **Martin Odersky**, profesor na Ecole Polytechnique Fédérale de Lausanne (EPFL).
- Krajem 80-ih doktorirao je na ETH Zürich kod profesora Niklausa Wirtha (kreatora Pascala i Module-2).
- Nakon toga naročito se **bavio istraživanjima u području funkcijskih jezika**, zajedno sa kolegom Philom Wadlerom (jednim od dva glavna kreatora funkcijskog jezika Haskell).
- Kada je izašla Java, Odersky i Wadler su 1996. napravili jezik **Pizza** nad JVM-om. Na temelju projekta Pizza, napravili su 1997./98. **Generic Java (GJ)**, koji je uveden u Javu 5 (malo ga je nadopunio Gilad Bracha, sa wildcardsima).

Scala - osnovne osobine

- **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 da se napusti Java, jer se **Java i Scala programski kod mogu jako dobro upotpunjavati**.

Scala - funkcije višeg reda

- Funkcije koje kao argument ili povratnu vrijednost imaju neku drugu funkciju, zovu se funkcije višeg reda (ovdje je to **suma**):

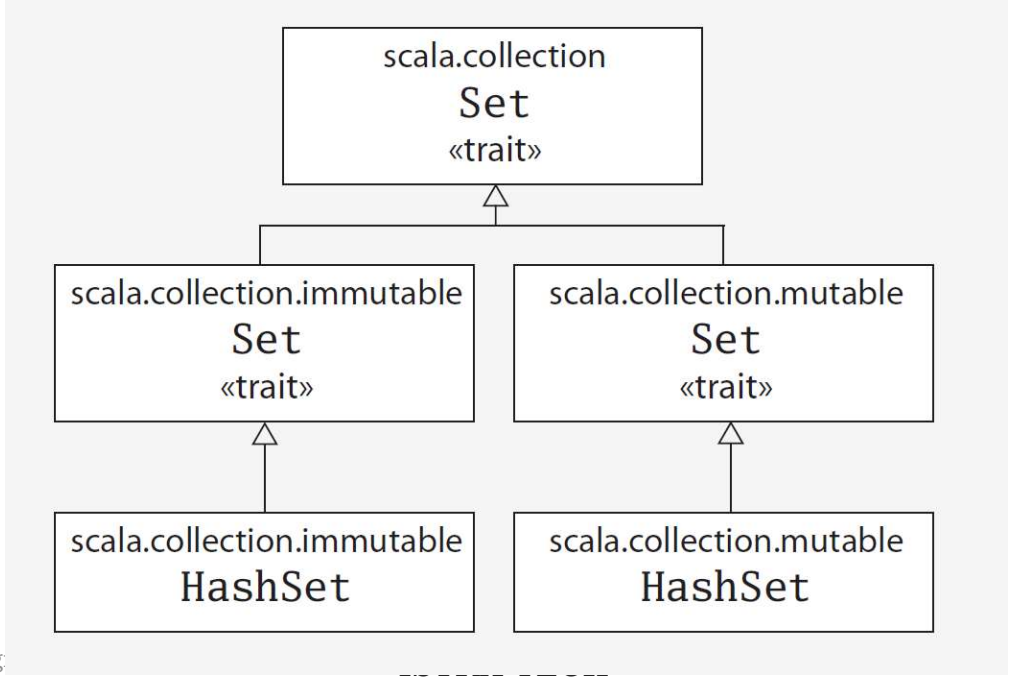
```
def suma(f: Int => Int, a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + suma(f, a + 1, b)
```

- Sada definiramo dvije funkcije i koristimo ih (za punjenje vrijednosti val varijabli) kao 1. parametar funkcije suma:

```
def kvadrat(x: Int): Int = x * x
def dvaNaNtu(x: Int): Int =
  if (x == 0) 1 else 2 * dvaNaNtu(x - 1)
val sumaKvadrata = suma(kvadrat, 1, 5)  // = 55
val sumaDvaNaNtu = suma(dvaNaNtu, 1, 5) // = 62
```

Scala – kolekcije, imutabilne i mutabilne

- Funkcijski jezici poznati su po tome da imaju izvrstan način rada sa kolekcijama, naročito sa listama (list).
- Glavne Scala kolekcije su **List, Set i Map**. List je uređena kolekcija objekata, Set je neuređena kolekcija objekata, a Map je skup parova (ključ, vrijednost).



Scala - lijena evaluacija (lazy evaluation)

- Inicijalizacija vrijednosti odgađa se do trenutka kada se (lijena) vrijednost prvi put koristi:

```
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) { // sa lijenom evaluacijom
  lazy val manager: Radnik = Db.get(managerId)
  lazy val tim: List[Radnik] = Db.tim(id)
}
```

Java - kratka povijest

- Java 1.0 se pojavila 1996. i (u pravo vrijeme!) reklamirana je kao jezik za Internet, čime je odmah stekla ogromnu slavu.
- Počeci Jave sežu u 1992., kada se zvala Oak i bila namijenjena za upravljanje uređajima za kabelsku televiziju i slične uređaje.
- **Sami autori su rekli da je Java = C++--**, tj. da je to pojednostavljeni (u pozitivnom smislu) C++. Nije stoga čudno da Java i C++ imaju sličnu sintaksu. Eiffel sintaksa je inspirirana jezikom ADA 83. Scala je negdje između.
- Međutim, Java nije podskup C++ jezika. Također, iako jednostavniji nego C++, Java nije baš jednostavan jezik.
- Eiffel i Scala su "čisti" OOPL jezici. C++ nije, jer je morao zadržati (potpunu) kompatibilnost sa jezikom C.
  Java je negdje između.

Java 8 - najvažnije nove mogućnosti: lambda izrazi, default metode, Streams

- 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).
- U Javi 8 pojavile su se i tzv. **default metode** u Java sučeljima (interfaces). One, zapravo, predstavljaju uvođenje **višestrukog nasljeđivanja implementacije** u Javu.
- Međutim, lambda izrazi i default metode su, na neki način, posljedica uvođenja treće važne mogućnosti u Javi 8, a to su **Streamsi**, koji nadograđuju dosadašnje Java kolekcije.

Java 8 - lambda izrazi

- Java 8 lambda (izrazi) temelje se na tzv. **funkcijskim sučeljima** (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.

Java 8 - default metode

```
// 1. 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);
}

// 2. 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);
    }
  }
}
```

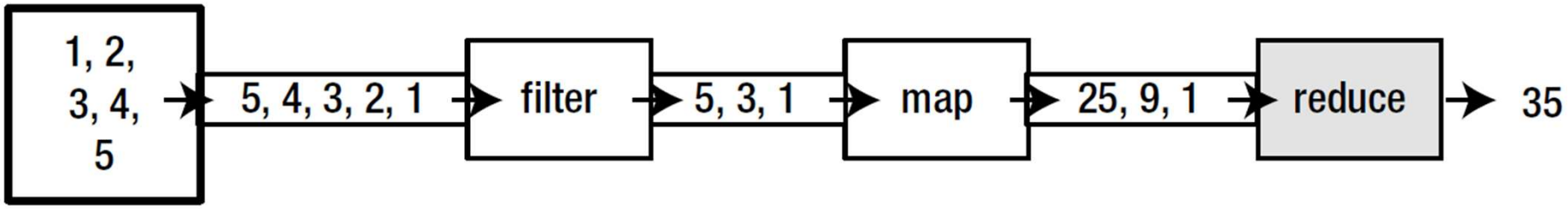
Java 8 - Stream (java.util.stream); Zašto su potrebni?

- 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.
- **Kako bi se olakšalo korištenje Streamsa, bilo je potrebno uvesti i lambda izraze i default metode**. Međutim, lambda izrazi i default metode mogu biti korisni i za ostale svrhe, ne samo tvorcima API-a, već i "običnim" programerima.
- 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.
- **Paralelne verzije Streamsa temelje se na ForkJoin.**

Java 8 - Stream programiranje je deklarativno (što, ne kako), kao SQL

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()
                 .filter(n -> n % 2 == 1)
                 .map(n -> n * n)
                 .reduce(0, Integer::sum);
```



numbers.stream( ).filter(n -> n % 2 == 1).map(n -> n * n).reduce(0, Integer::sum)

Kotlin

- Programski jezik Kotlin predstavila je 2011. poznata firma **JetBrains** (glavni autor je Andrej Breslav). Prva stabilna verzija 1.0 pojavila se 2016. JetBrains je "prepisao" izvorni kod svojih alata IntelliJ IDEA i Android Studio iz Jave u Kotlin.
- Kotlin je, kao i Scala, objektni jezik s funkcijskim proširenjima, koji radi (i) na JVM-u. Jezik je vrlo pragmatičan, tj. nije toliko napredan kao Scala, ali je lakši za učenje za Java programere. Moglo bi se reći da je Kotlin "bolja Java".
- Google je 2017. prihvatio Kotlin kao jedan od glavnih jezika za Android (uz Javu i C++). **U maju 2019., Google je izjavio da je Kotlin njihov preferirani jezik za Android programiranje.**
- Kao i Scala, Kotlin ima: eliminaciju repnog poziva (TCE); ime varijable piše se prije tipa varijable; varijable mogu biti imutabilne (**val**) i mutabilne (**var**); sintaksa kod funkcijskih proširenja je puno čitljivija od one u Javi.
- Za razliku od Scale, **Kotlin nema lijenu evaluaciju**, kao i dosta drugih naprednijih funkcijskih mogućnosti.

PL/SQL – prijedlog za uvođenje nekih funkcijskih mogućnosti

- Mislimo da bi bilo dobro (a vjerojatno nije nemoguće), da Oracle ugradi u PL/SQL neke funkcijske mogućnosti koje danas imaju svi funkcijski jezici i neki objektni jezici (npr. Scala, C++, Java, Kotlin):

- **funkcije višeg reda**
  (kod kojih argument može biti druga funkcija)

- **lambda izrazi** (anonimne funkcije)

- **Eliminacija (ili optimizacija) repnog poziva** (TCE ili TCO); niti Java još nema

- **mogućnost razlikovanja varijabli** koje se mogu mijenjati samo jednom (val) ili više puta (var)

SQL hijerarhijski upit – ima izbjegavanje greške kod ciklusa, ali to nije TCE!

```
-- varijanta koja puca kod petlje
SELECT empno, ename, mgr, LEVEL
  FROM emp
 START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr;

-- varijanta koja NE puca kod petlje
SELECT empno, ename, mgr, LEVEL,
       CONNECT_BY_ISCYCLE iscycle
  FROM emp
 START WITH mgr IS NULL
CONNECT BY NOCYCLE PRIOR empno = mgr;
```

In-place mutation
- direktno mijenjanje (zajedničkih) podataka

- Jim Gray, jedan od najvećih stručnjaka na području baza podataka (posebno transakcija), napisao je sljedeće

  http://www.hpl.hp.com/techreports/tandem/TR-81.3.pdf.
- **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 fail-fast.
- 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…

In-place mutation
- direktno mijenjanje (zajedničkih) podataka

- **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**.

- **Kako možemo primijeniti metode, koje su poznate u knjigovodstvu preko 550 godina, na kolekcije podataka**?
- Naime, kada dvije softverske dretve (ili više njih) mijenja kolekciju podataka na način da se radi in-place mutation, tada:

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

Izbjegavanje in-place mutation

- Rješenje (ne uvijek) je u tome da se izbjegne in-place mutation, 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.**

- Puno programera koji su navikli na imperativno programiranje isprva ostane skoro šokirani kada čuju za takav način rada.

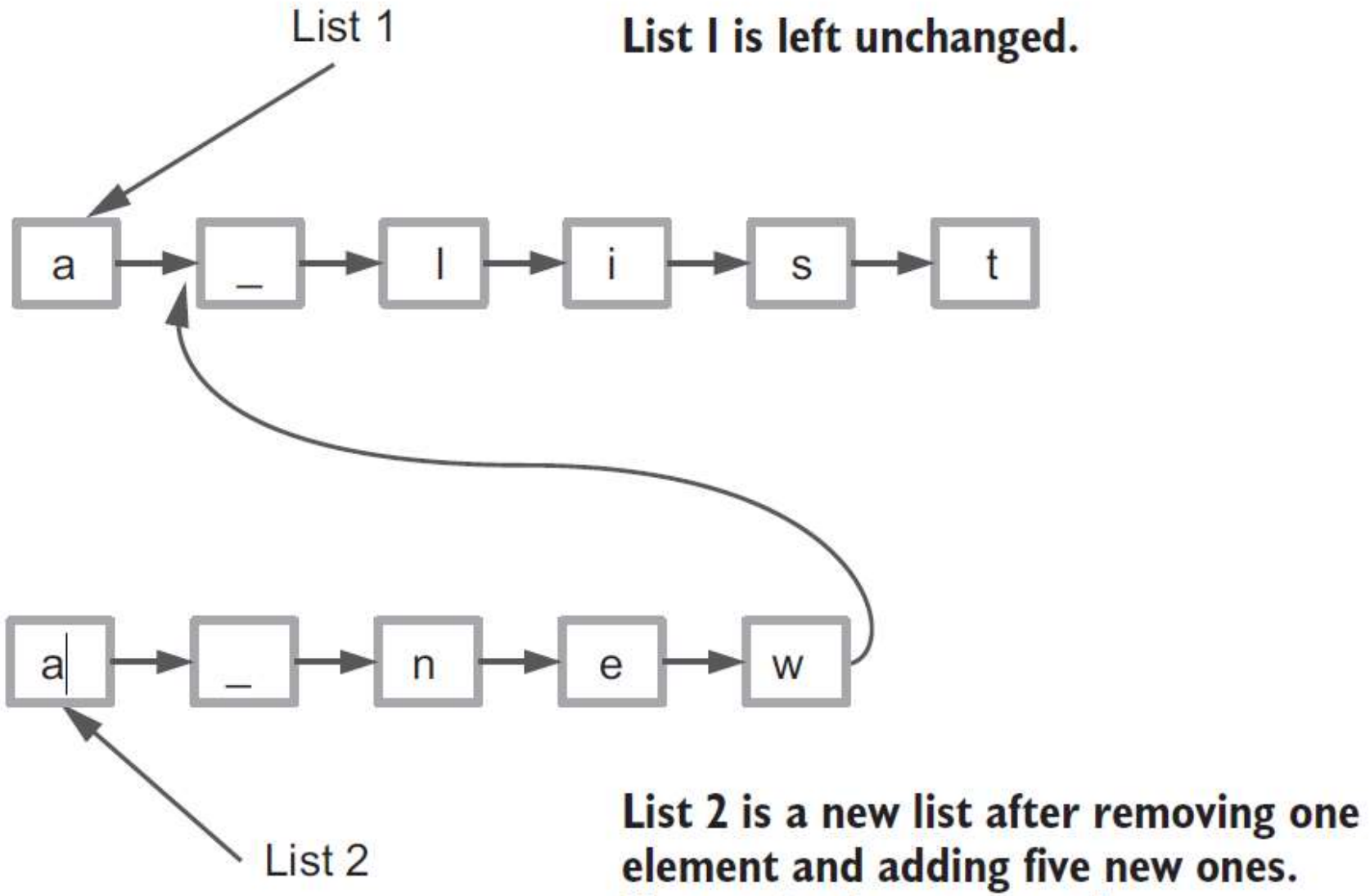
Uobičajene (dvije) primjedbe na izbjegavanje in-place mutation

- Uobičajene (dvije) primjedbe su:
  - ako jedna dretva stvara svoju kopiju podataka, i ne vidi tuđe izmjene, zar to nije loše?
  - zar ovakav način rada ne traži jako velike memorijske resurse, jer svaka dretva radi kopiju svojih podataka?

- Odgovor na prvo pitanje je – **pa 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 drugo pitanje – **najčešće ne treba raditi kopiju svih podataka, nego samo dijela.**

Primjer brisanja 1. elementa i dodavanja 5 novih elemenata u imutabilnu listu



Da li se s podacima u BP može raditi nešto slično kao sa imutabilnim listama kod programskih jezika?

- Iako to nije u potpunosti isto, postoji nešto što ima puno sličnosti, a to je – **ne koristiti (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.

Koje su prednosti i mane takvog pristupa

- Prednost bi bila (i) u tome što bi **podaci bili rjeđe zaključani** (INSERT isto radi zaključavanje, zbog provjere PK / UK).
- Naravno, može se izreći i puno primjedbi, npr.:

- tako bi nam trebalo **puno više prostora** za bazu podataka, jer bi se sve pamtilo; no današnji su diskovi sve veći, a danas neke regulative traže da se pamti sve
- danas skoro sve baze (Oracle od početka) rade tako da **DML naredbe ne zaključavaju podatke za čitanje**, tj. SELECT (Oracle multiversion consistency model rada s transakcijama)
- ako želimo pamtiti sve podatke, **u Oracle bazi možemo koristiti flashback tehnologije** (Flashback Data Archive, Flashback Query ...)
- **tako bi nam bilo jako teško programirati**; no možda se samo treba naviknuti – i funkcijsko programiranje je vrlo neobično onima koji su radili samo imperativno programiranje.

Zaključak

- Zadnjih nekoliko godina, funkcijsko programiranje stječe veliku popularnost u odnosu na imperativno programiranje.
- Funkcijsko programiranje, kao i logičko programiranje, po nečemu je vrlo slično programiranju u SQL-u - visoko je deklarativno.
- U prezentaciji smo vrlo kratko prikazali neke osobine funkcijskog logičkog jezika Prolog, čistog funkcijskog jezika Haskell i nekih funkcijskih proširenja u objektnim jezicima Scala, Java i Kotlin.
- Prikazali smo neke funkcijske mogućnosti koje bi Oracle (možda) mogao ugraditi u programski jezik PL/SQL.
- Ipak, najvažnije (ali i najteže) je pitanje: da li bismo u praksi mogli raditi s bazama podataka bez korištenja naredbi UPDATE i DELETE (samo INSERT)!?

Literatura (dio)

- Ben-Ari, M. (2012): Mathematical Logic for Computer Science (3. izdanje), Springer
- Gopalakrishnan, G. (2006): Computation Engineering - Applied Automata Theory and Logic, Springer
- Hutton, G. (2016): Programming in Haskell (2. izdanje), Cambridge University Press
- Meyer, B. (2009): Touch of Class - Learning to Program Well with Objects and Contracts, Springer
- Nilsson, U., Maluszynski J. (2000): Logic, programming and Prolog (2. izdanje), John Wiley & Sons
- Odersky, M., Spoon, L., Venners, B. (2016): Programming in Scala (3. izdanje), Artima Press
- Saumont, P-Y. (2017): Functional Programming in Java, Manning
- Saumont, P-Y. (2019): The Joy of Kotlin, Manning