

**Prolećni semestar, 2016/17**

##### Domaći zadatak: 01

­

**Ime:** Nemanja Andrić

**BrojIndeksa:** 2001

**Predmet:** Skripting jezici

**Šifra predmeta:** CS324

**Datum:** 09.07.2017.

# Cilj domaćeg zadatka:

Korišćenjem primera, analizirati i napisati izveštaj o funkcionalnom

programskom jeziku Haskel.

Ukratko objasniti sledeće modele programiranja:

* Objektno-orijentisano programiranje
* Imperativno programiranje
* Proceduralno programiranje
* Funkcionalno programiranje
* Deklarativno programiranje

# Rešenje:

**Programski jezik, Haskell**

Kratak opis funkcionalnog programskog jezika, Haskell

**Uvod u Haskell**

Haskell je funkcionalni programski jezik sa ne-striktnom (eng. lazy) semantikom. Dobio je ime po Haskell B. Curry-u, koji je bio jedan od pionira λ (lambda) računa— matematicka teorija funkcija i bila je inspiracija dizajnerima brojnih funckionalnih jezika. Haskell je prvi put bio specifikovan krajem 1980. godina, i od tada je prosao kroz dosta revizija. Prva stabilna verzija je izasla 2003. godine. Vazi za jedan je od najpopularnijih funkcijskih programskih jezika.

Haskell je staticko tipiziran jezik ali podrzava i polimorfizam i preopterecenje (eng. overloading). Kada se kompajlira program, kompajler tacno zna koji deo koda je broj, koji deo je string itd. Posto je deklarativan jezik, to znaci da nema implicitno stanje. Izvodjenje program se svodi na evaluaciju izraza i to bez stanja (eng. state-less).

Takodje predstavlja lazy jezik, sto znaci da izbegava sve popratne efekte. Popratni efekat (eng. side-effect) je svaka promena implicitnog stanja koja narusava referencijalnu prozirnost programa. Programi pisani u Haskell-u su koncizni—mnogo kraci od istih programa pisanih u drugim jezicima. Haskell ima vise implementacija za razlicite platforme— interpretiranu verziju, Hugs i nekoliko Haskell kompajlera (npr. ghc). Haskell varijable i tipovi Varijable u Haskellu se koriste za imenovanje ili deklaraciju izraza, a ne za čuvanje vrednosti u određenu memorijsku lokaciju. U Haskell jeziku, varijable kao da ne postoje. Ime moze da se veze za vrednost, ali jednom kad se veze, ime sluzi za tu vrednost i ne menja se tokom programa. Haskell promenljive su kao promenljive u matematickim izrazima. Ne sluze kao “kontejner” za skadistenje podataka kao u imperativnom programiranju.

1• Bool = False | True

• Char = 'a' | 'b' | ... | 'A' | 'B' | ...

• Int = -231 | ... | -1 | 0 | 1 | ... | 231-1

• Integer

• Double

Svi tipovi pocinju velikim slovima. Sve u Haskell-u ima tip, sto omogucava kompajleru da zna dosta o napisanom programu pre nego sto ga kompajlira. Iako je staticki tipiziran jezik, Haskell ima “type inference”. Ako napisemo broj, ne moramo da kazemo Haskellu da je to broj. Haskell moze to sam da zakljuci.

Primer:

λ pi

3.141592653589793

:: Floating a => a

λ

λ 5\*2

10

:: Num a => a

λ

Haskell medjutim ima i muttable promenljive. Iako nisu default, mogu da se koriste tako sto se definisu eksplicitno.

**Haskel nizovi (engl. array) i liste**

Haskell podrzava samo jedan niz konstruktor tip- Array koji daje immutable, boxed nizove. “Immutable” znaci da su ovi nizovi fiksni tj. ne mogu da se menjaju. “Boxed” znaci da su elementi nizova obicne Haskell (lazy) vrednosti.

Primer definisanja niza sa kvadratnim brojevima od 1 do 100:

squares = array (1,100) [(i, i\*i) | i <- [1..100]]

Primer - pristup elementu niza: squares!7 => 49

Liste u Haskellu su homogene strukture podataka. Sadrze vise elemenata istog tipa.

Liste se mogu kreirati uz pomoc ( : ) ili [ ]. U GHCI, liste se kreiraju na sledeci nacin:

ghci> let lostNumbers = [4,8,15,16,23,42]

ghci> lostNumbers

[4,8,15,16,23,42]

Liste se mogu spojiti uz pomoc ++ operatora.

**Haskel funkcije**

Posto je Haskell funkcionalni jezik, fukcije u Haskellu igraju veliku ulogu.

Primer curried funcije:

add :: Integer -> Integer -> Integer

add x y = x + y

Aplikacija dodaj (add) ima formu add e 1 e 2 , i ekvivalentna je (add e 1 ) e 2, posto fukcijska aplikacija asocira na levo (left). Drugim recima koriscenjem add na jedan argument, ima za posledicu da nova funkcija bude primenjena na drugi argument. Ovo je konzistentno sa tipom add, Integer -> Integer -> Integer, sto je ekvivaletno Integer -> (Integer -> Integer);

primer -> asocira na desno. Stoga koriscenjem add mozemo da definisemo inc na drugi nacin

One zapravo predstavljaju izraze koji trebaju izračunati neku vrednost.

Funkcija u programskom jeziku Haskell je postupak koji argumente jednog tipa preslikava u argumente nekog drugog tipa.

Primer faktorijal funkcije:

factorial 0 = 1

factorial n = n \* factorial (n - 1)

main = print (factorial 4)

Rezultat: 24

**Haskel I/O**

I/O sistem u Haskell-u, je cisto funkcionalan ali opet ima svu izrazajnu moc koja se moze naci u konvencijalnim programskim jezicima. U imperativnim jezicima, programi se izvrsavaju preko akcija koje ispituju i modifikuju trenutno stanje. To podrazumeva citanje i setovanje promenljivih, pisanje fajlova, citanje ulaza i otvaranje prozora. Takve akcije su deo Haskell-a ali su odvojene od ciste funkcionalne srzi jezika.

I/O sistem Haskell-a je izgradjen oko matematicke fondacije; Monad.

Monad predstavljaju konceptualne strukture na kojima se nalazi I/O.

Monad operatori na kojima je I/O sistem sagradjen se takodje koriste za druge svrhe. Na I/O monad mozemo gledati kao na abstraktan tip podataka.

Svaka I/O akcija vraca neku vrednost. U type sistemu, povratna vrednost je tipa IO . Na primer return vrednost funkcije getChar je:

getChar :: IO Char

IO Char oznacava da je fukciija getChar pozvana, odradila neke akcije i vratila karakter.

Akcije koje ne vracaju interesantu vrednost koriste vrednosti unit tip ( ). Primer putChar:

putChar

:: Char -> IO ( )

uzima karakter kao argument i ne vraca nista znacajno. Tip unit je slican tipu void u drugim jezicima.

**Rad sa datotekama**

Sto se tice mehanizma rada sa datotekama, tj. upis i ispis sadrzaja u i iz datoteke, on je veoma slican kao gore pomenutom upisu i ispisu na konzolu. Jedino što je potrebno je da se definiše ulazna i (ili) izlazna datoteka što se može videti na sledećim primerima.

Primer citanja i pisanja fajla:

doit file = do

contents <- readFile file

putStrLn contents

writeFile file "new content”

**Haskel klase i preklapanje (engl. overloading)**

Klasa u Haskell-u se najjednostavnije može definisati kao kolekcija tipova. Definiše se class deklaraciom, a instance deklaracija se koristi da bi se naznačilo (specificiralo) da određeni dati tip pripada toj klasi. Pojam tipa koji se pridružuje klasi se preciznije definiše preko konteksta. Kontekst zapravo zadaje neka ograničenja za promenljive tog tipa u okviru odgovarajuće klase.

Klase u Haskell-u ne predstavljaju neki tip podatka koje programer može da definise prema

potrebama programa vec one zapravo inicijalizuju neke operacije i pored toga je potrebno

definisati pomocu instance deklaracije nad kojim tipovima podataka se te operacije mogu

izvrsavati. Dakle, u Haskell-u klasama se ne mogu dodeliti vrednosti, već se vrednosti

dodeljuju tipovima podataka , a tipovi podataka mogu da pripadaju klasi tj. kao što je već

rečeno klasa se sastoji od kolekcije (familije) tipova podataka.

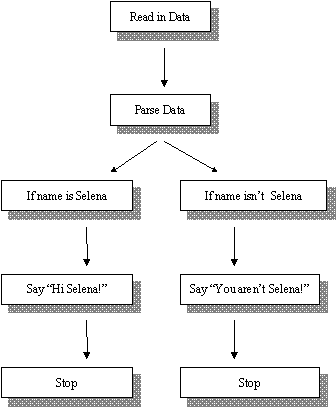
**Programske paradigme**

**Kratak opis programskih paradigm**

Programske paradigme obezbeđuju i definišu način viđenja ili daju ideju programeru kako se izvršava neki program. Neki programski jezici podržavaju samo jednu programsku paradigmu, za razliku od drugih koji obezbeđuju više njijh. Neke programske paradigme su poznatije po tome šta zabranjuju nego li po tome šta omogućuju. Postoji mnogo ovakvih modela programiranja, a u nastavku sledi opis onih najzastupljenijih.

## Proceduralno programiranje

S obzirom da su mašinski jezici veoma teški za korišćenje, i da njihova upotreba oduzima mnogo vremena pri pisanju koda, nastala je potreba da se kreira jezik približniji čoveku, a koji i dalje može da komunicira logikom računara. Jedana od paradigmi jezika višeg nivoa jeste i proceduralno programiranje. Ono se zasniva na nizu poziva određenih procedura, gde procedura može biti rutina, podrutina, funkcija...Praktično ima za cilj da izvrši niz koraka kako bi program izvršio neki zadatak. Koristeći sintaksu određenog programskog jezika programer piše instrukcije. Ove instrukcije su obično sačinjene od engleskih reči ili neke forme tih reči u sklopu sa aritmetičkim simbolima. Kasnije se te iste instrukcije konvertuju u mašinski jezik pomoću prevodioca ili kompajlera. Postoji mnogo proceduralnih jezika, međutim, samo mali broj njih je često korišćen, poput C/C++-a, COBOL-a, Fortran-a itd.



*Slika 1: Primer proceduralnog programiranja*

## Objektno-orijentisano programiranje

Kao što svi već znamo, računarski programi sastoje se od dva elementa: naredbi i podataka. Tako da program konceepcijski može biti oko svojih naredbi ili oko podataka s kojima radi. Drugim rečima, neki programi su napisani na osnovu onoga “šta se radi”, a drugi na osnovu onoga “sa čime se radi”. Prvi model nazivamo procesno-orijentisanim (*engl. Process-oriented*). Prema tom moedlu program se definiše kao niz uzastopnih koraka, što smo opisali u prethodnoj paradigm. Nasuprot tome, da bi se prevazišle poteškoće do kojih dovodi usložnjavanje programa, smišljen je drugi pristup – objektno-orijentisano programiranje (*engl. object-oriented programming*). Paradigma objektno-orijentisanog programiranja, ili skraćeno OOP, zasniva se na ideji da je program sastavljen od posebnih jedinica koje se nazivaju *objekti* (softverski skupovi koji se sastoje od podataka i programskih kodova kojima su opisane metode kojima objekti raspolažu) . Svaki objekat je sposoban da prima poruke, obrađuje podatke, i šalje poruke ka drugim objektima.

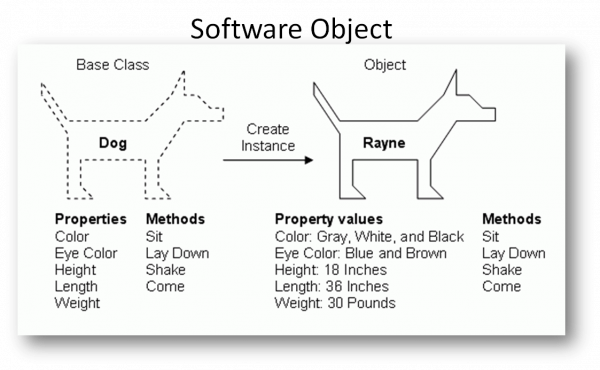
Bitan element objektno-orijentisanih jezika jeste *apstrakcija*. A to predstavlja sposobnost objekta da manipuliše samo podacima koji su potrebni njegovim metodama. Prilikom kreiranja nove klase definišu se atributi i metode koje klasa treba da ima. Izbor atributa zavisi od namene klase, tj. od metoda kojima će se opisati ponašanje klase. Svi objektno-orijentisani programi temelje se na tri osnovna mehanizma. A to su: kapsuliranje, nasleđivanje i polimorfizam.

**Kapsuliranje** *(engl. encaptulation*) predstavlja mehanizam koji povezuje programski kod i podatke sa kojima on radi, i oba elementa štiti od spoljnog uplitanja i zloupotrebe. Kapsulirani kod možete zamisliti kao “zaštitni omotač” koji štiti naredbe i podatke od proizvoljnog pristupa iz spoljnog sveta. Pristupanje ovim naredbama i podacima koji se nalaze unutar omotača strogo je kontrolisano pomoću dobro definisanih interfejsa.

**Nasleđivanje** *(engl. inheritance)* jeste postupak kojim jedan objekat preuzima svojstva drugog objekta. To je jako važno jer se poštuje koncept hijerarhije. Ovom prilikom potklase nasleđuju osobine svojih “roditelja”, ali mogu imati i svoje jedinstvene osobine, odnosno podatke i metode. Nasleđivanje i kapsuliranje su povezani. Ako određena klasa kapsulira izvesne osobine, onda će svaka njena podklasa imati te iste osobine, ali i osobine po kojima se ona izdvaja.

**Polimorfizam** *(engl. polymorphism)* je reč grčkog porekla, znači “mnogo oblika” i predstavlja osobinu koja omogućava da se jedan interfejs koristi za opštu klasu akcija. Koja će tačno akcija biti izvršena zavisi od prirode situacije. Naime, na poruku “izračunaj zapreminu” objekat klase cilindar bi to uradio na jedan, a objekat klase kocka na drugi način.

Prvi jezik koji je uveo pojam objekta bio je Simula 67, dok je prvi OO jezik Smalltalk. Danas postoje mnogi jezici koji podržavaju ovu paradigmu, a među najpoznatijima su: Ada 95, C#, C++, Fortran 2003, Perl, PHP, Python, Visual Basic, i naravno Java.

*Slika 2: Primer objektno-orijentisanog pristupa u programiranju*

## Imperativno programiranje

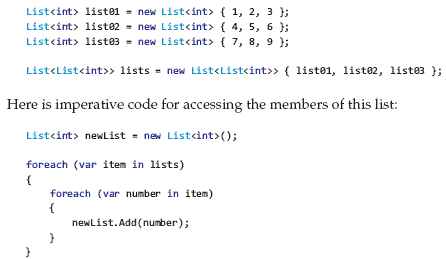
Imperativno programiranje predstavlja podskup proceduralnog programiranja, a oponent je funkcionalnom programiranju, i za njega se kaže da je “holy-grail" dobro napisanog koda. Ovom metodom se način izračunavanja može opisati nizom naredbi koje menjaju stanje programa, što je gore već navedeno. Identično kao u govornim jezicima imperativ se koristi da bi se izrazile komande koje pokreću neku akciju, i računar redom izvršava te komande. Ovakav način programiranje se naziva još i strukturno programiranje, a pojam procedure od suštinskog je značja da bismo shvatili kako se imperativni programi grade i kako oni izgledaju u smislu strukture koda. Ova tehnika omogućava lakše održavanje i daje sveukupno kvalitetnije programe.

Kako bi se još bolje približila imperativna paradigma dobro je krenuti od najnižeg nivoa – hardvera, obzirom na to da su osnovne ideje imperativnog programiranja konceptualno bliske hardveru, a takođe i ugrađene u sam hardver. Hardverska implementacija skoro svakog računara je imperativnog tipa, jer je on dizajniran da izvršava mašinski kod koji odgovara konkretnom računaru i takav kod je uvek imperativan.

## Deklarativno programiranje

Ovaj vid kodiranja uključuje kreiranje skupa relacija koje opisuju prostor rešenja, dok se analiza pojedinih koraka na putu pronalaženja tog rešenja ostavlja interpreteru. Kod ove vrste programiranja akcenat je strogo na tome ŠTA treba uraditi, a KAKO će se to postići nama je nebitno i ostavlja se na zadatak interpreteru. Na osnovu ovoga možemo zaključiti da je koncept deklarativnog programiranja sušta suprotnost gore navedenih paradigmi.

Deklarativna rešenja imaju dve nezavisne faze: deklarisanje i interpretiranje. A prednost ove paradigme ogleda se u tome da se rešenje nekog problema može prikazati na apstraktnom nivou, bez zalaženja u irelevantne detalje. Na taj način rešenje postaje preglednije i razumljivije. Programskmi jezici ovog tipa intenzivno se upotrebljavaju u sledećim sferama: upravljanje konfiguracijama, baze podataka, komunikacija između procesa. A među poznatije predstavnike ovog metoda kodiranja spadaju: Prolog I SQL.



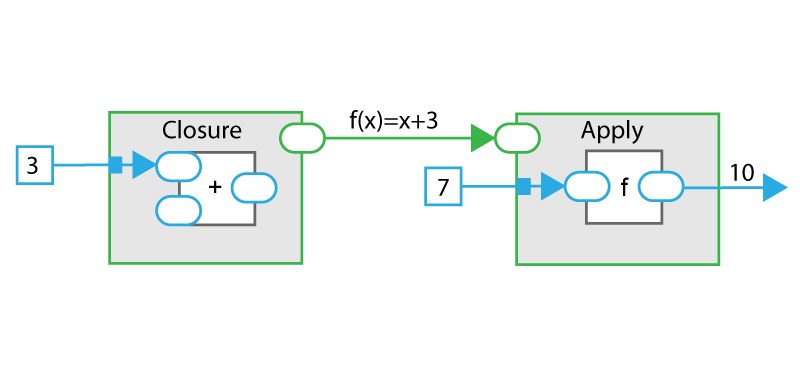
*Slika 3: Poređenje imperativnog i deklarativnog pristupa*

Deklarativno programiranje može biti: logičko i funkcionalno (koje će biti opisano u nastavku).

## Funkcionalno programiranje

Kao što smo gore već pomenuli, funkcionalna paradigma spada u grupu deklarativnih jezika i uz logičku paradigmi predstavlja direktni oponent proceduralnoj i imperativnoj paradigmi. U računarstvu, funkcionalno programiranje izvršavanje programa tretira kao izračunavanje matematičkih funkcija, čime se izbegava promena stanja. Ovaj vid programiranja je teorijski zasnovan na lambda računu. U praksi, razlika između funkcije u kontekstu funkcionalnog programiranja i matematičke funkcije je u tome što ona u matematičkom smislu ne može imati spoljašnje efekte koji mogu promeniti stanje efekta, što je već navedeno na početku. Npr. ukoliko je funkcija definisana da prima ceo i vraća ceo broj, koliko god puta da bude pozvana za za određen argument x, uvek će vratiti njemu odgovarajući izlaz. Zapravo, jedne od najkorišćenijih tehnika u funkcionalnom programiranju su tzv. funkcije višeg reda. Ove funkcije u suštini kao svoje argumente mogu imati proizvoljno složene funkcije.

Funkcionalni programski jezici se više koriste na univerzitetima nego li u komercijalnom razvoju softvera. Ipak, neki od poznatijih funkcionalnih jezika jesu: Haskell, Erlang, Scheme, XSLT(XML)…



*Slika 4: Primer funkcionalne tehnike kodiranja*