UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Ivana Trajčevski

ELEKTRONSKE LEKCIJE O PROGRAMSKIM PARADIGMAMA ZA UČENIKE SREDNJIH ŠKOLA

master rad

Beograd, 2022.

Mentor:

prof. dr Miroslav Marić

Članovi komisije:

dr Ivana Tomašević

Vladimir Kuzmanović

Datum odbrane: 14.07.2022.

Sadržaj

1 Uvod **1**

2 Elektronske lekcije **3**

3 Programske paradigme **7**

3.1 Vrste programskih paradigmi 7
3.2 Deklarativno programiranje 9

4 Iskazna logika **12**

4.1 Iskazi i iskazne formule 12
4.2 KNF i DNF 16
4.3 DPLL algoritam 18
4.4 Metod rezolucije u iskaznoj logici 20

5 Predikatska logika **22**

5.1 Sintaksa predikatske logike 22
5.2 Preneks normalna forma 28
5.3 Skolemizacija 29
5.4 Supstitucija 31
5.5 Unifikacija 32
5.6 Metod rezolucije u predikatskoj logici 35

6 Prolog **38**

6.1 Uvod u Prolog 38
6.2 Osnove Prolog-a 39
6.3 Sintaksa 42
6.4 Sistemski operatori i predikat odsecanja 44
6.5 Izračunavanje odgovora u Prolog-u 47
6.6 Rekurzija u Prolog-u 51
6.7 Liste 53
6.8 Rešavanje kombinatornih problema 57
6.9 Rešavanje logičkih problema 63

7 Haskell **70**

7.1 Uvod u Haskell 70
7.2 Funkcije u Haskell-u 74
7.3 Sistem tipova 75
7.4 Definisanje funkcija 81
7.5 Rekurzivne funkcije 86
7.6 Funkcije višeg reda 86
7.7 Korisnički tipovi i klase tipova 91

8 Zaključak **98**

Literatura **99**

Glava 1

Uvod

Internet danas predstavlja osnovnu vezu između onih koji uče i informacija koje su im potrebne. Razvoj interneta u poslednjih nekoliko decenija, kao i količina sadržaja koja je dostupna i koja se svakodnevno dopunjava, doveli su do razvoja i ekspanzije veb tehnologija, a sledstveno je povećana i potražnja za stručnjacima iz polja informacionih tehnologija. Elektronske platforme za učenje su pogodne zbog lakog pristupa, besplatnog korišćenja, kao i mogućnosti unapređivanja. Na internetu je dostupan širok spektar sadržaja o različitim oblastima koje pomažu u učenju programskih jezika. Međutim, većina je na engleskom jeziku. Kako bi se olakšao i omogućio proces učenja na srpskom jeziku, formirana je platforma *eŠkola veba* [1], potekla iz projekta radne grupe za obrazovni softver Matematičkog fakulteta Univerziteta u Beogradu. Javno je dostupna na adresi `http://www.edusoft.matf.bg.ac.rs/eskola_veba/#/home`. Kursevi se sastoje iz elektronskih lekcija, a one koje će biti opisane u ovom radu su elektronske lekcije o programskim paradigmama.

Programske paradigme su nastale sa ciljem da olakšaju proces programiranja. Novi programski jezici se brzo razvijaju i nije moguće savladati ih sve. Zbog toga, ukoliko poznajemo osnovne programske paradigme i koncepte na kojima se zasnivaju, možemo jednostavnije naučiti i primeniti novi programski jezik. Poznavanje različitih paradigmi pruža mogućnost sagledavanja problema iz više uglova, na osnovu čega se donosi odluka o načinu rešavanja problema, tj. bira se odgovarajuća paradigma i programski jezik koji će rešiti problem na što efikaniji način [17].

U ovom radu će biti opisane logička i funkcionalna paradigma, kao predstavnici deklarativne paradigme programiranja, kao i matematička teorija koja se nalazi u osnovi logičke paradigme. U uvodnom delu rada će biti reči o samom pojmu programske paradigme, kao i njihovim vrstama, naročito o logičkoj i funkcionalnoj paradigmi. U četvrtoj glavi biće opisani pojmovi iskazne logike, a zatim i predikatske logike kao njenog proširenja u okviru pete glave. Iskazna i predikatska logika će biti neophodne za razumevanje logičke

paradigme programiranja. U okviru šeste glave biće uvedeno logičko programiranje kroz programski jezik Prolog, dok će u sedmoj glavi osnovni koncepti funkcionalnog programiranja biti predstavljeni kroz programski jezik Haskell.

Glava 2

Elektronske lekcije

Elektronske lekcije o programskim paradigmama za učenike srednjih škola su dostupne na platformi eŠkola veba, na linku `http://edusoft.matf.bg.ac.rs/eskola_veba/#/course-details/pp`. Namenjene su svima koji su zainteresovani, a posebno učenicima IT odeljenja koji prvi put dolaze u dodir sa funkcionalnom i logičkom paradigmom, kao i sa matematičkom teorijom koja se nalazi u osnovi logičke paradigme. Sve lekcije su besplatne i javno dostupne na srpskom jeziku. Početni izgled platforme prikazan je na slici 2.1.

Slika 2.1: Početna strana platforme eŠkola veba

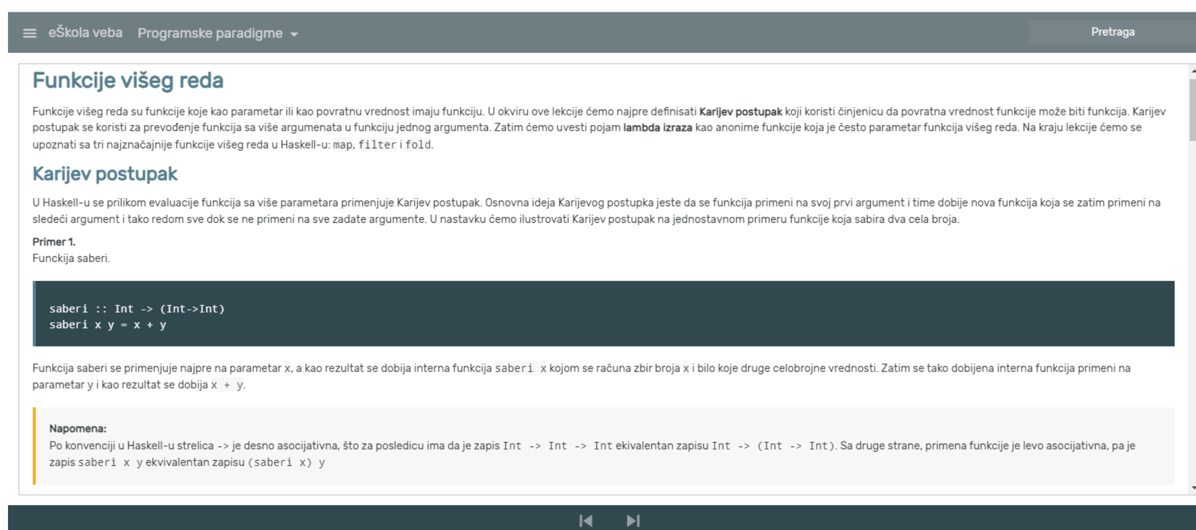
Na platformi se nalaze kursevi o različitim tehnologijama, a u okviru kurseva za srednjoškolce se nalazi kurs o programskim paradigmama kreiran za potrebe ovog rada. Izborom ovog kursa prikazuje se početna strana kao na slici 2.2.

Elektronske lekcije o programskim paradigmama su kreirane sa ciljem da učenicima specijalizovanog IT odeljenja bude dostupna literatura koja bi doprinela razumevanju

Slika 2.2: Početna strana kursa Programske paradigme

programskih paradigmi.

Izgled jedne od dostupnih lekcija prikazan je na slici 2.3.



Slika 2.3: Lekcija u okviru kursa

Za potrebe ovog rada kreirane su elektronske lekcije grupisane u pet celina, kao što je navedeno u nastavku.

1. Uvod (Uvod u programske paradigme, Deklarativna paradigma);
2. Iskazna logika (Iskazna logika, KNF i DNF, DPLL algoritam, Metod rezolucije u iskaznoj logici);
3. Predikatska logika (Predikatska logika, Preneks normalna forma, Skolemizacija, Supstitucija, Unifikacija, Metod rezolucije u predikatskoj logici);

4. Prolog (Uvod u Prolog, Osnove Prolog-a, Sintaksa, Sistemski predikati i operatori, Izračunavanje u Prolog-u, Rekurzija u Prolog-u, Liste, Osnovna pravila za rad sa listama, Rešavanje kombinatornih problema, Rešavanje logičkih problema, Rezime o Prolog-u);

5. Haskell (Uvod u Haskell, Funkcije u Haskell-u, Sistem tipova, Definisanje funkcija, Rekurzivne funkcije, Funkcije višeg reda, Korisnički tipovi i klase tipova, Rezime o Haskell-u).

Sve lekcije su pažljivo osmišljene, detaljno su objašnjeni svi kocepti i pojmovi, a zatim i ilustrovani odgovarajućim primerima. Na slici 2.4 prikazana je detaljana analiza jednog od kompleksnijih zadataka u okviru lekcija.

Problem misionara i ljudoždera

Formulacija problema

Tri misionara i tri ljudoždera *(eng. missionaries and cannibals)* koji se nalaze na levoj obali reke je potrebno prevesti na desnu obalu čamcem koji prima dve osobe. Pri tome, ne sme u jednom trenutku na obali biti više ljudoždera od misionara, jer će u suprotnom ljudožderi pojesti misionare. Potrebno je naći raspored prevoženja, tako da svi bezbedno pređu reku.

Analiza rešenja

U nastavku su definisana pravila i činjenice koji da omogućavaju bezbedno prelaženje reke.

- Potrebno je opisati činjenice koje će govoriti o broju misionara i ljudoždera na obalama i položaju čamca. Dovoljno je zadati broj misionara i ljudoždera na levoj strani, jer je time jednoznačno određen broj misionara i ljudoždera i na desnoj strani:
  `stanje(br_misionara, br_ljudozdera,levo).`
  Početno stanje je `stanje(3,3,levo)`, a završno `stanje(0,0, desno)`.
- Potrebno je zadati legalne prelaze imajući u vidu da se čamcem mogu prevesti najviše dve osobe. Primer legalnog prelaza je `legalan_prelaz(2, 0)`, sa značenjem da se u čamcu prevoze dva misionara i nijedan ljudožder.
- Pored legalnih prelaza, potrebno je zadati i legalna stanja imajući u vidu da broj ljudoždera ni na jednoj obali, ni u jednom trenutku, ne sme biti veći od broja misionara. Prilikom zadavanja legalnih stanja položaj čamca je irelevantan i zbog toga taj podatak izostavljamo. Primer legalnog stanja je `legalno_stanje(X, X)`, sa značenjem da se na obali nalazi jednak broj misonara i ljudoždera.
- Predikat `novo_stanje` na osnovu trenutnog stanja formira legalno sledeće stanje:
  `novo_stanje(stanje(M1, Lj1, levo), stanje(M2, Lj2, desno)).`
- Na osnovu trenutnog i sledećeg stanja, predikat `izbor_poteza` nalazi odgovarjući potez, takav da ljudožderi ne mogu pojesti misionare:
  `izbor_poteza(stanje(M1, Lj1, levo), stanje(M2, Lj2, desno), potez(M, Lj, desno)).`
  Primer jednog poteza je `potez(1,1,desno)`, sa značenjem da se jedan misioar i jedan ljudožder prevoze čamcem na desnu stranu obale.
  Lista ovako izakranih poteza će upravo biti rešenje zadatka.
- Ključan predikat, `rasporedi`, generiše prethodno spomenutu listu poteza koji omogućavaju bezbedan raspored prevoženja. Ovaj predikat kao argumente prima trenutno stanje, listu do tada primenjenih stanja i kao treći argument, listu koja predstavlja rešenje:
  `rasporedi(TrenutnoStanje, Do_sada, [Potez | PreostaliPotezi]).`

Slika 2.4: Analiza problema misionara i ljudoždera iz lekcije

A zatim je na slici 2.5 detaljno prikazan kôd rešenja.

```
/*legalni prelazi*/
legalan_prelaz(2, 0).
legalan_prelaz(1, 0).
legalan_prelaz(1, 1).
legalan_prelaz(0, 1).
legalan_prelaz(0, 2).

/*legalna stanja*/
legalno_stanje(X, X).
legalno_stanje(3, X).
legalno_stanje(0, X).

rasporedi(stanje(0, 0, desno), _, []).
rasporedi(TrenutnoStanje, Do_sada, [Potez | PreostaliPotezi]) :-
        novo_stanje(TrenutnoStanje, SledeceStanje),
        not(clan(SledeceStanje, Do_sada)),
        izbor_poteza(TrenutnoStanje, SledeceStanje, Potez),
        rasporedi(SledeceStanje, [SledeceStanje | Do_sada], PreostaliPotezi).


izbor_poteza(stanje(M1, Lj1, levo), stanje(M2, Lj2, desno), potez(M, Lj, desno)) :-
        M is M1 - M2,
        Lj is Lj1 -Lj2.
izbor_poteza(stanje(M1, Lj1, desno), stanje(M2, Lj2, levo), potez(M, Lj, levo)) :-
        M is M2 - M1,
        Lj is Lj2 - Lj1.
```

Slika 2.5: Deo koda koji predstavlja rešenje problema misionara i ljudoždera

Nakon pokretanja prethodnog programa u okviru interpretera, prikazuje se sadržaj kao na slici 2.6.

```
| ?- misionari.
potez(1,1,desno)
potez(1,0,levo)
potez(0,2,desno)
potez(0,1,levo)
potez(2,0,desno)
potez(1,1,levo)
potez(2,0,desno)
potez(0,1,levo)
potez(0,2,desno)
potez(1,0,levo)
potez(1,1,desno)
```

Slika 2.6: Rezultat poziva prethodnog programa

Lekcije sadrže pitanja i zadatke pomoću kojih će korisnici moći da ispitaju nivo razumevanja i primene stečenog znaja. Na slici 2.7 prikazan je zadatak u okviru lekcije namenjen korisnicima.

Zadatak 1.
Koristeći rekurziju i poklapanje šablona napisati funkciju koja računa n-ti stepen broja 2.

Slika 2.7: Zadatak namenjen korisnicima

Pitanja koja se nalaze na kraju jedne od lekcija prikazana su na slici 2.8.

Pitanja:
1. Šta karakteriše deklarativnu paradigmu?
2. Koje su osnovne karakteristike logičke paradigme?
3. Koje su osnovne karakteristike funkcionalne paradigme?

Slika 2.8: Pitanja na kraju lekcije

Glava 3

Programske paradigme

Reč *paradigma* je starogrčkog porekla, a njeni sinonimi su uzor, obrazac, šablon. U svakodnevnom govoru pod paradigmom često podrazumevamo skup nekih objekata sa zajedničkim karakteristikama.

Programska paradigma (engl. programming paradigm) je stil ili način programiranja [5]. Stil pisanja koda ostaje gotovo nepromenjen među programskim jezicima koji pripadaju istoj programskoj paradigmi. To nam omogućava da odaberemo i savladamo nekoliko programskih jezika koji su predstavnici različitih paradigmi, a zatim to znanje možemo primeniti i brže savladati nove programske jezike.

3.1 Vrste programskih paradigmi

Programske paradigme se prema [2] dele na četiri osnovne:

- imperativna paradigma;
- objektno-orijentisana paradigma;
- funkcionalna paradigma;
- logička paradigma.

Imperativna paradigma je stil programiranja u kome se kôd programa izvršava liniju za linijom onim redom kojim je napisan. Kontrola toka programa vrši se preko uslovnog grananja i ciklusa. Tipični programski jezici za imperativno programiranje su *C*, *PASCAL*, *FORTRAN* i drugi. Ovaj stil blizak je načinu na koji mašina funkcioniše, pa je pogodan za sistemsko programiranje.

Objektno-orijentisana paradigma predstavlja stil programiranja u kome se kôd programa organizuje u okviru klasa i objekata. Klasa predstavlja šablon na osnovu kojeg se kreiraju objekti. Podaci i instrukcije su odvojeni, pa tako podatke čuvamo u poljima objekata, dok su instrukcije grupisane u metode. Na ovaj način možemo pisati apstraktiniji kôd koji je bliži onome što vidimo u stvarnom svetu i organizovan je na način da svaki objekat obavlja tačno određeni zadatak. Na taj način možemo jednostavno već iz samog imena objekta i metoda pretpostaviti funkcionalnost koju on obavlja. Podaci se mogu obrađivati isključivo primenom metoda što smanjuje zavisnosti između različitih komponenata programskog koda i čini ovu paradigmu pogodnom za razvoj velikih aplikacija uz mogućnost saradnje većeg broja programera [3]. Tipični predstavnici objektno-orijentisane paradigme su: *Java*, *C#*, *C++* i drugi. Ova paradigma trenutno predstavlja najzastupljeniji stil programiranja.

Funkcionalna paradigma je programska paradigma u kojoj se sva izračunavanja vrše kroz evaluaciju matematičkih funkcija. Iako funkcije postoje i u savremenim imperativnim programskim jezicima, to nije dovoljno da bi govorili o funkcionalnom programiranju. Kod funkcionalnog programiranja neophodno je da budu zadovoljeni još neki uslovi. Izlazna vrednost funkcije zavisi isključivo od njenih ulaznih vrednosti. Ovo znači da funkcija ne sme da pristupa i koristi globalne promenjljive. Pored toga, funkcija ne sme da obavlja dodatne zadatke koji bi blokirali izvršavanje funkcije. Osnovna ideja funkcionalnog programiranja je da oponaša osobine matematičkih funkcija. Trenutno, funkcionalno programiranje je najveću primenu pronašlo u razvoju veb (engl. Web) aplikacija. Često se koristi i u kombinaciji sa objektno-orijentisanim programiranjem. Tipični predstavnici funkcionalne paradigme su: *Lisp, Haskell* i *Scala.*

Logička paradigma predstavlja programsku paradigmu u kojoj se za programiranje koristi matematička logika. Nastala je kao direktna posledica rada na automatskom dokazivanju teorema. Najpoznatiji predstavnik logičke paradigme je *Prolog.*

Iako smo za svaku od osnovnih paradigmi naveli programske jezike koji su njeni tipični predstavnici, ne znači nužno da jedan programski jezik podržava samo jednu paradigmu. Takođe, ukoliko sintaksa programskog jezika odgovara jednoj paradigmi, moguće je veštački oponašari neke segmente druge paradigme. Tako na primer, u programskom jeziku *C*, koji je prvenstveno namenjen imperativnom programiranju, moguće je imitirati principe objektno-orijentisane paradigme uz pomoć struktura i funkcija. Takođe, postoje i programski jezici čija sintaksa podržava kombinovanje različitih paradigmi. Programski jezik *C++* možemo koristiti za objekto-orijentisano, imperativno i funkcionalno programiranje. Zbog toga *C++* nazivamo hibridnim jezikom.

Pored prethodno navedene podele programskih paradigmi na četiri osnovne, postoji i opštija podela prema načinu rešavanja problema na **proceduralnu** i **deklarativnu** paradigmu. Za razliku od proceduralnog programiranja gde je osnovni zadatak programera da opiše način (proceduru) kojim se dolazi do rešenja problema, kod deklarativnog programiranja osnovni zadatak programera je da opiše problem, dok se mehanizam programskog jezika bavi pronalaženjem rešenja problema.

3.2 Deklarativno programiranje

Odlika deklarativnog programiranja je da omogući pisanje koda koji predstavlja opis problema, a ne i koraka neophodnih za rešavanje tog problema. To olakšava proces programiranja, ali zbog nemogućnosti automatskog pronalaženja efikasnih algoritama koji bi rešili široku klasu problema, domen primene deklarativnih jezika je često ograničen.

Još jedna bitna karakteristika deklarativnog programiranja je ta da za zadate ulazne parametre program mora da vrati isti rezultat pri svakom izvršenju, nezavisno od bilo kog drugog stanja.

Izvršavanje programa u deklarativnom stilu programiranja se može svesti na evaluaciju izraza. U zavisnosti od izraza, razlikujemo *logičku* i *funkcionalnu* paradigmu. Kod logičke paradigme programiranja izrazi su *relacije*, a kod funkcionalne paradigme izrazi predstavljaju *funkcije*.

Logičko programiranje

U osnovi logičke paradigme nalazi se predikatska logika (logika prvog reda). Za razliku od ostalih osnovnih paradigmi programiranja, logička paradigma ima uzak domen primene. Nastala je sa idejom da omogući automatizaciju dokazivanja teorema. Pogodna je za rešavanje problema matematičke logike, baza podataka i nekih oblasti veštačke inteligencije.

U okviru logičkog programiranja, programer navodi niz pravila o rezultatu računa. Zbog toga se ovaj stil programiranja naziva još i programiranje zasnovano na pravilima (engl. Rule based). Činjenica da se logičko programiranje zasniva na pravilima čini ga pogodnim za rad sa podacima. Ovo je razlog zbog koga je logička paradigma popularna kod upitnih jezika među kojima su: *SQL, SPARQL, XQuery* i drugi. Najkorišćeniji jezik za rad sa bazama podataka je *SQL*. On omogućava da definišemo niz uslova, odnosno pravila, na osnovu kojih nam sistem za upravljanje bazama podataka vraća rezultat. Na taj način, mi ne definišemo na koji način se ti podaci čitaju iz memorije, već samo uslove

koje rezultat upita mora da zadovoljava.

Pored upitnih jezika, najpoznatiji i najznačaniji jezik za logičko programiranje je *Prolog*. Programski jezik Prolog koristi matematičku logiku za opisivanje problema, a dokazivač teorema kao mehanizam za njegovo rešavanje [4]. Slično kao kod upitnih jezika, u Prolog-u se definišu objekti i relacije među njima. Najveća primena Prolog-a ogleda se u dokazivanju teorema i veštačkoj inteligenciji. Zbog toga veću primenu ima u domenu istraživanja i nauke, nego u praktičnom računarstvu.

Funkcionalno programiranje

Osnovni cilj funkcionalnog programiranja je da oponaša matematičke funkcije [7]. Iako u imperativnom programiranju takođe postoje funkcije, pojam funkcije u imperativnom i funkcionalnom programiranju se razlikuje. U imperativnim jezicima funkciju koristimo kako bi izdvojili grupu naredbi, a zatim ih pozivali po potrebi. Sa druge strane, funkcionalno programiranje posmatra pojedinačne operacije kao izračunavanje matematičkih funkcija. Ovo znači da se kompletan kôd programa kod funkcionalnog programiranja sastoji isključivo iz niza definicija i poziva funkcija. Kako bi ovo bilo moguće, programski jezik koji podržava funkcionalnu paradigmu mora da poseduje ugrađene osnovne funkcije, kao i mehanizme za kreiranje novih, kompleksnijih funkcija [8]. Važno je napomenuti da svi programi napisani imperativnim stilom, mogu biti napisani i funkcionalnim stilom.

Kod funkcionalnih programskih jezika, funkcija je ravnopravna sa svim ostalim tipovima podataka. To znači da funkcija može biti ulazni parametar druge funkcije, ili njena povratna vrednost. Ukoliko funkcija prima funkciju kao argument, ili vraća funkciju kao rezultat, tada nju nazivamo *funkcijom višeg reda*.

Funkcionalno programiranje se zasniva na *lambda računu*. Lambda račun je koncept koji se bavi isključivo pravilima za transformaciju izraza, ali ne i arhitekturom mašine na kojoj se program izvršava. Zbog toga se kôd napisan funkcionalnim programskim jezicima mora prevesti u imperativni mašinski kôd. Lambda račun razdvaja imenovanje i definisanje funkcije. Ovo nam omogućava da definišemo funkciju, a da joj pritom ne dodelimo ime. Funkcije bez imena nazivamo još i *lambda funkcije* ili *anonimne funkcije*.

Danas većina programskih jezika podržava neke od osobina funkcionalnog programiranja. Funkcionalna paradigma postaje sve popularnija. Veliki doprinos popularizaciji funkcionalnom programiranju doneo je i *React* programski okvir za razvoj veb aplikacija. Takođe, mnoge svetski poznate kompanije poput Microsoft-a, Google-a, Intel-a i drugih, počinju da koriste funkcionalne programske jezike. Konkretno, kompanija Facebook

implementira anti-spam programe u programskom jeziku Haskell.

Glava 4

Iskazna logika

Ključni izazovi iskazne logike su ispitivanje da li je data iskazna formula valjana (tautologija) i da li je data iskazna formula zadovoljiva. Pre nego što pređemo na metode za ispitivanje valjanosti i zadovoljivosti iskaznih formula, podsetićemo se osnovnih pojmova iskazne logike.

4.1 Iskazi i iskazne formule

Iskaz je rečenica koja ima svojstvo da je ili tačna ili netačna [9]. Rečenice „Jupiter je planeta" i „Nebo je plavo", primeri su iskaza. Rečenica „x je jednako 3" nije iskaz, jer njena istinitosna vrednost nije poznata, tačnije zavisi od vrednosti promenljive x.

Elementarni iskazi su promenljive koje nazivamo **iskaznim slovima** ili **atomičkim iskazima**.

Prethodno navedeni primeri rečenica iz svakodnevnog govora, „Jupiter je planeta" i „Nebo je plavo", su atomički iskazi. U logici ćemo atomičke iskaze obeležavati malim slovima latinice.

Složenije iskaze gradimo pomoću **logičkih veznika** ili **logičkih operacija**:

- negacija $\neg$
- konjunkcija $\wedge$
- disjunkcija $\vee$
- implikacija $\Rightarrow$
- ekvivalencija $\Leftrightarrow$

Primeri složenih iskaza iz svakodnevnog govora su: „Jupiter je planeta i nebo je plavo", „Ako Petar položi prijemni, upisaće željeni fakultet".

Simbolima $\top$ i $\bot$ ćemo redom označavati istinitosne vrednosti *tačno* i *netačno*. $\top$ i $\bot$ nazivamo **iskaznim konstantama**.

Definicija 4.1 *Iskazne formule su iskazne konstante, iskazna slova i složeni iskazi nastali upotrebom logičkih veznika.*

Iskazna slova i logičke konstante nazivamo **atomičkim iskaznim formulama**.

Literal je iskazna formula koja je ili atomička iskazna formula ili negacija atomičke iskazne formule. Primeri literala su: $p, \neg q, s$. **Klauza** je disjunkcija literala. Primer jedne klauze je $p \vee \neg q \vee r$.

Konvencija o prioritetu logičkih veznika: $\neg$ je veznik najvišeg prioriteta, zatim slede veznici $\wedge$ i $\vee$ koje su podjednakog prioriteta i na kraju su $\Rightarrow$ i $\Leftrightarrow$ takođe jednakog prioriteta.

Valjanost i zadovoljivost iskaznih formula

Glavni zadatak iskazne logike je da odredi istinitosnu vrednost iskazne formule.

Istinitosnu vrednost atomičkog iskaza p, koju označavamo sa $v(p)$, definišemo na sledeći način:

$$v(p) = \begin{cases} 1 & \text{, ako je iskaz p tačan,} \\ 0 & \text{, ako je iskaz p netačan.} \end{cases}$$

Dodeljivanje istinitosnih vrednosti iskaznim slovima, u oznaci v, naziva se **valuacija**.

Dodeljivanje istinitosnih vrednosti iskaznim formulama za datu valuaciju v u oznaci I_v naziva se **interpretacija**.

Vrednost iskazne formule A u interpretaciji I_v označavamo sa $I_v(A)$. Ukoliko za valuaciju v važi $I_v(A) = 1$, kažemo da je formula tačna u interpretaciji I_v, a ukoliko je $I_v(A) = 0$, kažemo da je netačna.

Interpretaciju I_v definišemo na sledeći način:

- $I_v(p) = v(p)$, za svaki atomički iskaz p;
- $I_v(\top) = 1$,
  $I_v(\bot) = 0$;
- $I_v(\neg A) = \begin{cases} 1, & \text{ako } I_v(A) = 0, \\ 0, & \text{ako } I_v(A) = 1; \end{cases}$
- $I_v(A \wedge B) = \begin{cases} 1, & \text{ako je } I_v(A) = 1 \text{ i } I_v(B) = 1, \\ 0, & \text{inače;} \end{cases}$
- $I_v(A \vee B) = \begin{cases} 0, & \text{ako je } I_v(A) = 0 \text{ i } I_v(B) = 0, \\ 1, & \text{inače;} \end{cases}$
- $I_v(A \Rightarrow B) = \begin{cases} 0, & \text{ako je } I_v(A) = 1 \text{ i } I_v(B) = 0, \\ 1, & \text{inače;} \end{cases}$

- $I_v(A \Leftrightarrow B) = \begin{cases} 1, & \text{ako je } I_v(A) = I_v(B), \\ 0, & \text{inače.} \end{cases}$

Iskazna formula A je **zadovoljiva** ukoliko postoji valuacija iskaznih slova v u kojoj je ta formula tačna, tj. ukoliko važi da je $I_v(A) = 1$. Tada kažemo i da je v *model* za A i pišemo $v \models A$.

Iskazna formula A je **valjana** ili **tautologija** ako je, za sve moguće valuacije njenih iskaznih slova, uvek tačna. To kraće zapisujemo $\models A$.

Istinitosne tablice

Pravila za određivanje vrednosti iskazne formule u zadatoj valuaciji navedena u prethodnom poglavlju mogu biti reprezentovana osnovnim *istinitosnim tablicama*:

A	$\neg A$
$\top$	$\bot$
$\bot$	$\top$

A	B	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Na osnovu navedenih tablica, može se konstruisati istinitosna tablica za proizvoljnu iskaznu formulu [10].

Primer 4.1 *Ispitati istinitosne vrednosti formule* $(p \wedge q) \vee r \Rightarrow p \vee r$ *za sve moguće valuacije iskaznih slova* p, q *i* r.

Tabela 4.1: Istinitosna tablica iskazne formule $(p \wedge q) \vee r \Rightarrow p \vee r$

p	q	r	$p \wedge q$	$(p \wedge q) \vee r$	$p \vee r$	$(p \wedge q) \vee r \Rightarrow p \vee r$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\top$	$\bot$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$
$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\top$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$

Svakoj vrsti tablice odgovara jedna valuacija iskaznih slova p, q i r. U zavisnosti od vrednosti iskaznih slova, izračunate su vrednosti složenijih iskaznih formula, sve do tražene koja se nalazi u poslednjoj koloni. Primetimo da su u poslednjoj koloni istnitosne tablice sve vrednosti zadate iskazne formule jednake $\top$. Drugim rečima, za sve moguće valuacije iskaznih slova formule $(p \wedge q) \vee r \Rightarrow p \vee r$, ona je uvek tačna. Sledi da je data iskazna formula tautologija.

Vrednost iskazne formule zavisi isključivo od istinitosnih vrednosti iskaznih slova. Primetimo da za n iskaznih slova postoji ukupno 2^n različitih valuacija. Metod istinitosnih tablica se svodi na proveravanje svih mogućih vrednosti promenljivih koje se javljaju u formuli. U zavisnosti od broja promenljivih, ispitivanje valjanosti neke formule metodom istinitosne tablice može biti jako zahtevno. Upravo je to motivacija za uvođenje naprednije metode za ispitivanje valjanosti i zadovoljivosti iskaznih formula. U pitanju je DPLL algoritam.

Logička posledica

Na osnovu tautologije, kao iskaza koji je uvek tačan, možemo izvesti pravila logičkog zaključivanja.

Za neki iskaz kažemo da je **logička posledica** ili **zaključak** skupa **pretpostavki** ako kad god je tačna svaka od pretpostavki, tačan je i zaključak. Na primer, ukoliko iz uslova p, q i r koji su uvek tačni izvodimo zaključak s, to ćemo označiti sa:

$$\frac{p,\ q,\ r}{s}$$

Neka od pravila zaključivanja u iskaznoj logici prikazana su u nastavku.

- *Modus ponens*

$$\frac{p,\ p \Rightarrow q}{q}$$

  Čitamo: „Ako p i iz p sledi q, onda q".

  Ovo pravilo zaključivanja opravdava tautologija $p \wedge (p \Rightarrow q) \Rightarrow q$.

- *Tranzitivnost implikacije*

$$\frac{p \Rightarrow q,\ \ q \Rightarrow r}{p \Rightarrow r}$$

  Čitamo: „Ako p sledi q i iz q sledi r, onda iz p sledi r".

  Ovo pravilo zaključivanja opravdava tautologija $(p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$.

- *Pravilo kontrapozicije*

$$\frac{\neg p \Rightarrow \neg q}{q \Rightarrow p}$$

  Čitamo: „Ako $\neg p$ sledi $\neg q$, onda iz q sledi p".

  Ovo pravilo zaključivanja opravdava tautologija $(\neg p \Rightarrow \neg q) \Rightarrow (q \Rightarrow p)$.

- *Svođenje na protivrečnost*

$$\frac{\neg p \Rightarrow (q \wedge \neg q)}{p}$$

  Čitamo: „Ako iz $\neg p$ sledi kontradikcija, onda p".

  Ovo pravilo zaključivanja opravdava tautologija $(\neg p \Rightarrow (q \wedge \neg q)) \Rightarrow p$.

Logički ekvivalentne formule

Za iskazne formule A i B kažemo da su **logički ekvivalente**, u oznaci $A \equiv B$, ako i samo ako važi $A \Leftrightarrow B$.

Za potrebe DPLL algoritma, navešćemo nekoliko elementarnih logičkih ekvivalencija (tabela 4.2).

Tabela 4.2: Elementarne logičke ekvivalencije

$A \wedge A \equiv A$	zakon idempotencije za $\wedge$
$A \vee A \equiv A$	zakon idempotencije za $\vee$
$\neg\neg A \equiv A$	zakon dvojne negacije
$A \vee B \equiv B \vee A$	zakon komutativnosti za $\vee$
$A \wedge B \equiv B \wedge A$	zakon komutativnosti za $\wedge$
$A \vee (B \vee C) \equiv (A \vee B) \vee C$	zakon asocijativnosti za $\vee$
$A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$	zakon asocijativnosti za $\wedge$
$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$	zakon distributivnosti $\wedge$ u odnosu na $\vee$
$(B \vee C) \wedge A \equiv (B \wedge A) \vee (C \wedge A)$	zakon distributivnosti $\wedge$ u odnosu na $\vee$
$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$	zakon distributivnosti $\vee$ u odnosu na $\wedge$
$(B \wedge C) \vee A \equiv (B \vee A) \wedge (C \vee A)$	zakon distributivnosti $\vee$ u odnosu na $\wedge$
$\neg(A \wedge B) \equiv \neg A \vee \neg B$	De Morganov zakon
$\neg(A \vee B) \equiv \neg A \wedge \neg B$	De Morganov zakon

4.2 KNF i DNF

Definicija 4.2 *Iskazna formula je u **konjunktivnoj normalnoj formi** (KNF) ako je oblika*

$$D_1 \wedge D_2 \wedge \ldots \wedge D_n,$$

pri čemu je svaka od formula $D_i (1 \leq i \leq n)$ disjunkcija literala.

Primer formule u konjunktivnoj normalnoj formi je $(p \vee q) \wedge (\neg q \vee r) \wedge (s \vee p)$.

Definicija 4.3 *Iskazna formula je u* ***disjunktivnoj normalnoj formi*** *(DNF) ako je oblika*

$$C_1 \vee C_2 \vee \ldots \vee C_n,$$

pri čemu je svaka od formula C_i $(1 \leq i \leq n)$ *konjunkcija literala.*

Primer formule u disjunktivnoj normalnoj formi je $(p \wedge q) \vee (\neg q \wedge r) \vee (s \wedge p)$.

Svaka iskazna formula se može transformisati u svoju konjunktivnu, odnosno disjunktivnu normalnu formu korišćenjem odgovarajućih ekvivalencija.

Algoritam svođenja na KNF i DNF

Neka je data iskazna formula koja sadrži iskazna slova p i q. Tada se se algoritam svođenja na KNF i DNF izvodi na sledeći način:

- eliminacija veznika $\Leftrightarrow$ koristeći logičku ekvivalenciju

$$p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (q \Rightarrow p),$$

- eliminacija veznika $\Rightarrow$ koristeći logičku ekvivalenciju

$$p \Rightarrow q \equiv \neg p \vee q,$$

- ubacivanje $\neg$ u zagradu koristeći logičke ekvivalencije

$$\neg(p \wedge q) \equiv \neg p \vee \neg q, \neg(p \vee q) \equiv \neg p \wedge \neg q,$$

- eliminacija višestruke negacije koristeći

$$\neg\neg p \equiv p,$$

- podešavanje na KNF ili DNF, primenjivanjem odgovarajućih logičkih ekvivalencija
  - Za KNF

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

$$(q \wedge r) \vee p \equiv (q \vee p) \wedge (r \vee p)$$

  - Za DNF

$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$$

$$(q \vee r) \wedge p \equiv (q \wedge p) \vee (r \wedge p)$$

- primeniti naredne trivijalne logičke ekvivalencije

$$p \wedge p \equiv p, p \vee p \equiv p, q \wedge \top \equiv, q \vee \top \equiv \top, q \wedge \bot \equiv \bot, q \vee \bot \equiv q.$$

Formule zapisane u konjunktivnoj i disjunktivnoj normalnoj formi pogodne su za primenu raznih algoritama. Primer takvog algoritma je DPLL koji se primenjuje jedino na formule zapisane u konjunktivnoj normalnoj formi.

4.3 DPLL algoritam

Dejvis–Patnam–Logman–Lovelandova ili **DPLL** algoritam ispituje zadovoljivost iskaznih formula. Znajući da je formula valjana ako i samo ako njena negacija nije zadovoljiva, DPLL algoritam se koristi i za ispitivanje valjanosti date formule.

Primenjuje se na iskazne formule zapisane u konjunktivnoj normalnoj formi. Za potrebe DPLL algoritma, KNF formulu ćemo predstaviti skupom klauza, a klauzu ćemo predstaviti skupom literala. Na primer, formulu $(p \wedge \neg q) \vee (q \wedge \neg r \wedge s)$ možemo predstaviti kao skup klauza na sledeći način: $\{\{p, \neg q\}, \{q, \neg r, s\}\}$. Upravo će skup klauza date formule u KNF biti ulaz DPLL algoritma.

Ulaz: Skup kauza D, ($D = \{C_1, C_2, \ldots, C_n\}$)
Izlaz: DA, ako je skup D zadovoljiv, u suprotnom izlaz je NE

- ako je D prazan, vrati DA
- zameni sve literale $\neg\bot$ sa $\top$ i zameni sve literale $\neg\top$ sa $\bot$
- obriši sve literale jednake $\bot$
- ako D sadrži praznu klauzu, vrati NE
- ako neka klauza C_i sadrži $\top$ ili sadrži i neki literal i njegovu negaciju, onda vrati vrednost koju vraća $DPLL(D \backslash C_i)$
- ako je neka klauza jedinična i jednaka nekom iskaznom slovu p, onda vrati vrednost koju vraća $DPLL(D[p \to \top])$;
  ako je klauza jedinična i jednaka $\neg p$, gde je p neko iskazno slovo, onda vrati vrednost koju vraća $DPLL(D[p \to \bot])$ (**propagacija jediničnih klauza** (engl. unit propagation))
- ako D sadrži literal p, ali ne i $\neg p$, onda vrati vrednost koju vraća $DPLL(D[p \to \top])$;
  ako D sadrži literal $\neg p$, ali ne i p, onda vrati vrednost koju vraća $DPLL(D[\neg p \to \top])$ (**eliminacija čistih literala** (engl. pure literal))
- ako je p jedno od iskaznih slova koja se javljaju u D i $DPLL(D[p \to \top])$ vraća DA, onda vrati DA; inače vrati vrednost koju vraća $DPLL(D[p \to \bot])$ (**split** (engl. splitting rule))

Algoritam: DPLL

Odabir literala u okviru *split* pravila je veoma važan. Neke varijante ovog pravila su da se bira literal sa najviše pojavljivanja u tekućoj formuli, da se bira neki literal iz najkraće klauze itd. U okviru koraka *propagacija jediničnih klauza* zaključuje se da ukoliko formula sadrži klauzu sa tačno jednim literalom, taj literal ne može biti netačan u zadovoljavajućoj valuaciji. Na osnovu pravila *eliminacija čistih literala* zaključujemo da ukoliko se u formuli pojavljuje literal i ne pojavljuje se njemu suprotan literal, taj literal se može postaviti na tačno.

DPLL algoritam je znatno efikasniji od metode istinitosnih tablica. Efikasnost se ogleda u ranoj proveri zadovoljivosti formule za izabranu valuaciju iskaznog slova u okviru koraka split.

4.4 Metod rezolucije u iskaznoj logici

Metod rezolucije je metod za izvođenje zaključaka (logičkih posledica) u iskaznoj logici koji se primenjuje na formule zapisane u KNF. Metod rezolucije se zasniva na uzastopnoj primeni pravila rezolucije koje ćemo obeležavati sa res.

Ako su C' i C'' klauze, onda se pravilom rezolucije iz klauza $C' \vee l$ i $C'' \vee \neg l$ izvodi klauza $C' \vee C''$, što kraće zapisujemo nasledeći način:

$$\frac{C' \vee l \quad C'' \vee \neg l}{C' \vee C''} \quad (res)$$

Za literale l i $\neg l$ kažemo da su *komplementni*.
Klauzu $C' \vee C''$ zovemo *rezolventom roditeljskih* klauza $C' \vee l$ i $C'' \vee \neg l$.

Primetimo da na osnovu logičkih ekvivalencija

$$C' \vee l \equiv \neg C' \Rightarrow l, \;\; C'' \vee \neg l \equiv l \Rightarrow C''$$

i tranzitivnosti implikacije, pravilo rezolucije možemo zapisati i na sledeći način:

$$\frac{\neg C' \Rightarrow l \quad l \Rightarrow C''}{\neg C' \Rightarrow C''} \quad (res)$$

Napomena: Metod rezolucije predstavlja uopštenje pravila *modus ponens*.

Naredni primer ilustruje primenu pravila rezolucije nad iskaznim formulama zapisanim govornim jezikom.

Primer 4.2 *Metodom rezolucije dokazati da iz pretpostavki:*

„Vuk je izgubio u finalnom meču ili je osvojio turnir”
„Vuk nije osvojio turnir ili je dobio pobednički pehar”

sledi zaključak:

„Vuk je izgubio u finalnom meču ili je dobio pobednički pehar.”

Označimo sa p tvrđenje „Vuk je izgubio u finalnom meču”, sa q tvrđenje „Vuk je osvojio turnir” i sa r zaključak „Vuk je dobio pobednički pehar”.

Pretpostavke možemo prevesti u iskazne formule na sledeći način:

$$p \vee q$$

$$(\neg q) \vee r$$

Sada direktno iz klauza $p \vee q$ i $\neg q \vee r$ primenom metoda rezolucije dobijamo traženi zaključak:

$$p \vee r.$$

Primer 4.3 *Metodom rezolucije dokazati da iz pretpostavki:*

„Ako pada kiša, Vuk ponese kišobran."
„Ako Vuk ima kišobran, on nije mokar."
„Ako ne pada kiša, Vuk nije mokar."

sledi zaključak:

„Vuk nije mokar."

Označimo sa iskaznim slovom p pretpostavku „Pada kiša", sa q „Vuk ima kišobran" i sa r „Vuk je mokar".

Sada navedene pretpostavke možemo prevesti u iskazne formule na sledeći način:

1. $\neg p \vee q$

2. $\neg q \vee \neg r$

3. $p \vee \neg r$
   Koristeći metod rezolucije zaključujemo:

4. $\neg p \vee \neg r$ (rezolventa roditeljskih klauza 1. i 2. dobijena primenom pravila rezolucije)

5. $\neg r \vee \neg r$ (rezolventa roditeljskih klauza 3. i 4. dobijena primenom pravila rezolucije)

6. $\neg r$ (traženi zaključak dobijen primenom zakona indempotencije za $\vee$ na 5.).

Glava 5

Predikatska logika

Glavni nedostatak iskazne logike je nedovoljna izražajnost, drugim rečima to što iskazna logika ne razmatra smisao samih iskaza. Rečenica „x je jednako 3" nije iskaz jer njena istinitosna vrednost zavisi od promenljive x. Ako posmatramo rečenice „Postoji x tako da važi x je jednako 3" i „Za svako x važi x je jednako 3", zaključujemo da je njihova istinitosna vrednost jasna. Prva rečenica je uvek tačna, a druga uvek netačna.

Reči *postoji* i *za svaki* nazivamo **kvantifikatorima**. Kvantifikatori predstavljaju jednu od novina u predikatskoj logici i čine je znatno izražajnijom u odnosu na iskaznu logiku.

5.1 Sintaksa predikatske logike

Naredna formula:

$$\forall x(x \in \mathbf{R} \Rightarrow x^2 \geq 0),$$

prevedena na govorni jezik ima značenje da je kvadrat svakog realnog broja nenegativan. Primetimo da u terminima iskazne logike ne bismo mogli na matematički ispravan način zapisati ovo tvrđenje. Predikatska logika predstavlja proširenje iskazne logike. U navedenoj formuli pored prethodno spomenutih kvantifikatora, kao novinu uočavamo i promenljive, operaciju kvadriranja, kao i relaciju $\geq$. U nastavku ćemo se upoznati sa sintaksom predikatske logike.

U izgradnji predikatskih formula pored logički simbola učestvuju i nelogički simboli.

Logički simboli:

- beskonačan skup promenljivih: x, y, z, ...
- logički veznici: $\wedge, \vee, \neg, \Rightarrow$ i $\Leftrightarrow$
- logičke konstante: $\top$, $\bot$

- kvantifikatori: univerzalni $\forall$ i egzistencijalni $\exists$
- pomoćni simboli (zarez i zagrade)

Nelogički simboli:

- simboli konstanti: a, b, c, ..., 0, 1, ...
- funkcijski simboli: f, g, h, ..., $+$, $*$, ...
- relacijski (predikatski) simboli: p, q, r, ..., $\leq$, $\subseteq$, ...

Skup funkcijskih simbola ćemo obeležavati sa $\mathcal{F}$, a skup relacijskih (predikatskih) simbola sa $\mathcal{R}$.

Funkcija arnosti, $ar : (\mathcal{F} \cup \mathcal{R}) \rightarrow \mathbf{N}$, svakom funkcijskom ili relacijskom simbolu dodeljuje njegovu dužinu. Simboli konstanti su funkcijski simboli arnosti nula.

Rečnik ili *signatura* $\mathcal{L}$ sastoji se od funkcijskih i relacijskih simbola, kao i funkcije arnosti.

$$\mathcal{L} = \{\mathcal{F}, \mathcal{R}, ar\}$$

Još jedna novina predikatske logike je **term**.

Termovi

Skup termova definišemo nad signaturom $\mathcal{L}$ i skupom promenljivih, kao najmanji skup za koji važi:

- svaki simbol promenljive je term;
- svaki simbol konstante je term;
- ako su $t_1, t_2, \ldots, t_n$ termovi i f funkcijski simbol arnosti n, onda je i $f(t_1, t_2, \ldots, t_n)$ term.

Primer 5.1 *Data je signatura $\mathcal{L}$ sa konstantnim simbolima a i 1, i funkcijskim simbolom f arnosti 2. Navesti nekoliko primera termova nad $\mathcal{L}$.*

Primeri predikatskih formula su: $\rho(a,x), \neg\rho(a,x), \forall z \rho(a,z), \forall x \exists y \rho(x,y)$.

Atomička formula

Skup atomičkih formula definišemo nad signaturom $\mathcal{L}$ i skupom promenljivih, kao najmanji skup za koji važi:

- logičke konstante $\top$ i $\bot$ su atomičke formule;

- ako su $t_1, t_2, \ldots, t_n$ termovi i p predikatski simbol arnosti n, onda je i $p(t_1, t_2, \ldots, t_n)$ atomička formula.

Primer 5.2 *Data je signatura $\mathcal{L}$ sa konstantnim simbolima 3 i 1, i relacijskim simbolima $\{=, <, >\}$ arnosti 2. Navesti nekoliko primera atomičkih formula nad $\mathcal{L}$.*

Primeri atomičkih formula su: $x = 3, 3 > 1, x < 1$.

Predikatska formula

Skup predikatskih formula definišemo nad signaturom $\mathcal{L}$ i skupom promenljivih, kao najmanji skup za koji važi:

- atomičke formule su formule;
- ako je A formula, onda je i $\neg A$ formula;
- ako su A i B formule, onda su i $(A \vee B)$, $(A \wedge B)$, $(A \Rightarrow B)$ i $(A \Leftrightarrow B)$ formule;
- Ako je A formula i x promenljiva, onda su i $(\forall x A)$ i $(\exists x A)$ formule.

Napomena: Zapis $(\forall x A)$ čitamo: „Za svako x A". Zapis $(\exists x A)$ čitamo: „Postoji x takvo da je A."

Primer 5.3 *Data je signatura $\mathcal{L}$ sa konstantnim simbolom a i relacijskim simbolom ρ arnosti 2. Navesti nekoliko primera predikatskih formula nad $\mathcal{L}$.*

Primeri predikatskih formula su: $\rho(a, x), \neg\rho(a, x), \forall z \rho(a, z), \forall x \exists y \rho(x, y)$.

Kao i u iskaznoj logici, definišemo pojmove literal i klauza.

Literal je atomička formula ili negacija atomičke formule. **Klauza** je disjunkcija literala.

Napomena: Kada su prioriteti veznika u pitanju, predikatska logika nasleđuje konvenciju koja je uvedena u iskaznoj logici, uz dodatak da kvantifikatori imaju najveći prioritet.

Zapisivanje rečenica

Razmotrimo formulu:

$$\forall x(p(x) \wedge q(y)) \Rightarrow (\exists z r(z)),$$

i primetimo da se promenljive x i z nalaze pod dejstvom kvantifikatora. Za takve promenljive kažemo da su *vezane*. Ukoliko se promenljiva ne nalazi pod dejstvom kvantifikatora kažemo da je *slobodna*. U prethodno navedenoj formuli slobodna promenljiva je y. Za formulu koja nema slobodnih promenljivih kažemo da je *zatvorena formula* ili *rečenica*.

Uvođenjem kvantifikatora rečenice iz svakodnevnog života se mogu prevesti u predikatsku logiku. *Generalna tvrđenja*, koja važe za svaki slučaj, zapisivaćemo koristeći kvantifikator $\forall$. Na primer, neka je dato tvrđenje: „Ljudi idu u kupovinu". Dato tvrđenje se može prevesti u predikatsku logiku na sledeći način:

$$\forall x p(x),$$

gde je $p(x)$ predikat sa značenjem „x ide u kupovinu".

Primer 5.4 *Tvrđenja*

„Svako zadovoljstvo se plaća" i
„Postoji zadovoljstvo koje se plaća"

Prevesti u predikatsku logiku.

Označimo sa $z(x)$ predikat koji ima značenje „x je zadovoljstvo" i sa $p(x)$ predikat sa značenjem „x se plaća". Navedena tvrđenja možemo prevesti na sledeći način:

$$\forall x(z(x) \Rightarrow p(x))$$

$$\exists x(z(x) \wedge p(x)).$$

Interpretacija i vrednost predikatske formule

Razmotrimo predikatsku formulu

$$\exists x P(x).$$

Kako bismo odredili istinitosnu vrednost ove formule, potrebno je da za promenljivu x odredimo domen, a predikatu P pridružimo neko značenje.

Na primer, ukoliko sa $P(x)$ označimo jednačinu $3x+1=0$, a za domen izaberemo skup prirodnih brojeva, formula $\exists x P(x)$ će biti netačna. Ukoliko x pripada skupu racionalnih brojeva, formula će biti tačna.

Kako bi istinitosna vrednost formule bila nedvosmislena, jasno ćemo definisati način interpretiranja predikatskih formula.

Interpretacija

Za dati jezik $\mathcal{L}$ definišemo strukturu $\mathcal{D} = (D, I^{\mathcal{L}})$ takvu da važi:

- D je neprazan skup i predstavlja *domen*;
- svakom simbolu konstante iz $\mathcal{L}$ funkcija $I^{\mathcal{L}}$ pridružuje jedan element $c_I \in D$;

- svakom funkcijskom simbolu f, $ar(f) = n$ iz $\mathcal{L}$ funkcija $I^{\mathcal{L}}$ pridružuje funkciju $f_I : D^n \rightarrow D$
- svakom predikatskom simbolu p, $ar(p) = n$ iz $\mathcal{L}$ funkcija $I^{\mathcal{L}}$ pridružuje funkciju $p_I : D^n \rightarrow \{0, 1\}$

Valuacija v za skup promenljivih V i domen D je preslikavanje koje svakom elementu iz V dodeljuje vrednost iz D. To zapisujemo na sledeći način:

$$v(x_i) = d_i,$$

i kažemo da je d vrednost promenljive x u valuaciji v. Ako su v i w valuacije za isti skup promenljivih i isti domen, onda sa $v \sim_x w$ označavamo da je $v(y) = w(y)$ za svaku promenljivu y različitu od x, pri čemu vrednosti $v(x)$ i $w(x)$ mogu, a ne moraju biti iste.

Interpretacija I_v je predstavljena parom $(\mathcal{D}, v)$.

Definicija 5.1 *Vrednost terma t u interpretaciji I_v definišemo na sledeći način:*

- *ako je t jednako promenljivoj x, onda je $I_v(t) = v(x)$;*
- *ako je t jednako konstanti c, onda je $I_v(t) = c_I$;*
- *ako je t jednako funkcijskom simbolu $f(t_1, \ldots, t_n), ar(f) = n$ i ako važi $I_v(t_i) = d_i$ onda $I_v(f(t_1, \ldots, t_n)) = f_I(d_1, \ldots, d_n)$.*

Definicija 5.2 *Istinitosna vrednost formule u interpretaciji I_v određenoj $\mathcal{L}$-strukturom $\mathcal{D}$ i valuacijom v, definišemo na sledeći način:*

- $I_v(\top) = 1,$
  $I_v(\bot) = 0;$
- $I_v(p(t_1, t_2, \ldots, t_n)) = p_I(I_v(t_1), I_v(t_2), \ldots, I_v(t_n));$
- $I_v(\neg A) = \begin{cases} 1, & \textit{ako } I_v(A) = 0, \\ 0, & \textit{ako } I_v(A) = 1; \end{cases}$
- $I_v(A \wedge B) = \begin{cases} 1, & \textit{ako je } I_v(A) = 1 \textit{ i } I_v(B) = 1, \\ 0, & \textit{inače}; \end{cases}$
- $I_v(A \vee B) = \begin{cases} 0, & \textit{ako je } I_v(A) = 0 \textit{ i } I_v(B) = 0, \\ 1, & \textit{inače}; \end{cases}$
- $I_v(A \Rightarrow B) = \begin{cases} 0, & \textit{ako je } I_v(A) = 1 \textit{ i } I_v(B) = 0, \\ 1, & \textit{inače}; \end{cases}$

- $I_v(A \Leftrightarrow B) = \begin{cases} 1, & \textit{ako je } I_v(A) = I_v(B), \\ 0, & \textit{inače.} \end{cases}$

- $I_v \forall x A) = \begin{cases} 1, & \textit{ako za svaku valuaciju } w, \textit{ takvu da je } w \sim_x v \textit{ važi } I_w(A) = 1, \\ 0, & \textit{inače.} \end{cases}$

- $(I_v(\exists x A) = \begin{cases} 1, & \textit{ako postoji valuacija } w, \textit{ takva da je } w \sim_x v \textit{ i } I_w(A) = 1, \\ 0, & \textit{inače.} \end{cases}$

Primer 5.5 *Nad signaturom $\mathcal{L} = \{U, F, ar(U) = 2, ar(F) = 1\}$ i skupom promenljivih $V = \{x, y, z, \ldots\}$ data je formula*

$$\forall y(F(y) \Rightarrow U(c, y)).$$

Neka je domen skup $D = \{$ Platon, Aristotel, Ksenofon, Tales $\}$. Funkcija $I^{\mathcal{L}}$ konstantu i relacijske simbole interpretira na sledeći način:

- *konstanti c pridružuje „Sokrat”*
- *relacijskom simbolu U pridružuje relaciju ucitelj*
- *relacijskom simbolu F pridružuje relaciju filozof*

Nakon interpretacije formula postaje tvrđenje:

$$\forall y(F(y) \Rightarrow U(Sokrat, y))$$

sa značenjem: „Za svako y, ukoliko važi da je y filozof, tada je Sokrat učitelj od y”. Ovo tvrđenje nije tačno jer postoji y iz domena za koje tvrđenje ne važi.

Ključni izazovi predikatske logike, kao i iskazne logike, su ispitivanje da li je data formula valjana (tautologija) i da li je data formula zadovoljiva. Za razliku od iskazne logike, u predikatskoj logici postoje metodi koji za svaku valjanu formulu mogu da utvrde da je valjana (ali ne mogu za svaku formulu koja nije valjana da utvrde da ona nije valjana). Jedan od ključnih metoda je metod rezolucije koji predstavlja osnovni mehanizam za izvođenje zaključaka u programskom jeziku Prolog. Da bi se metod rezolucije primenio, predikatska formula mora proći kroz četiri faze.

1. Prevođenje formule u preneks normalnu formu (PNF).
2. Skolemizacija.
3. Supstitucija.
4. Unifikacija.

U nastavku će detaljno biti opisan svaki od navedenih postupaka.

5.2 Preneks normalna forma

Formula je u **preneks normalnoj formi** ukoliko je oblika

$$Q_1x_1Q_2x_2\dots Q_nx_nF,$$

gde $Q_i \in \{\forall, \exists\}$ i F je formula bez kvantifikatora.

$Q_1x_1Q_2x_2\dots Q_nx_n$ se naziva *prefiks* formule F. Pošto prefiks može biti pazan, svaka formula bez kvantifikatora se smatra formulom u preneks normalnoj formi.

Postupak svođenja na preneks normalnu formu:

- dok god je moguće, primenjivati sledeće logičke ekvivalencije

$$A \Leftrightarrow B \equiv (A \Rightarrow B) \wedge (B \Rightarrow A) \ \ i$$

$$A \Rightarrow B \equiv \neg A \vee B$$

- dok god je moguće, primenjivati sledeće logičke ekvivalencije

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

$$\neg\forall x A \equiv \exists x \neg A$$

$$\neg\exists x A \equiv \forall x \neg A$$

- eliminisati višestruke veznike koristeći logičku ekvivalenciju

$$\neg\neg A \equiv A$$

- dok god je moguće, primenjivati sledeće logičke ekvivalencije

$$(\forall x A) \wedge B \equiv \forall x(A \wedge B)$$

$$(\forall x A) \vee B \equiv \forall x(A \vee B)$$

$$B \wedge (\forall x A) \equiv \forall x(B \wedge A)$$

$$B \vee (\forall x A) \equiv \forall x(B \vee A)$$

$$(\exists x A) \wedge B \equiv \exists x(A \wedge B)$$

$$(\exists x A) \vee B \equiv \exists x(A \vee B)$$

$$B \wedge \exists x A \equiv \exists x(B \wedge A)$$

$$B \vee \exists x A \equiv \exists x(B \vee A)$$

  Ukoliko je u formuli B x slobodna promenljiva, potrebno je izvršiti preimenovanje na sledeći način:

$$(\forall x A) \wedge B \equiv \forall u(A[x \to u]) \wedge B)$$

Primer 5.6 *Transformisati formulu*

$$\forall x \exists y P(x,y) \wedge (\forall x \exists y Q(x,y) \Rightarrow \forall x \exists y S(x,y))$$

u preneks normalnu formu.

$$\forall x \exists y P(x,y) \wedge (\forall x \exists y Q(x,y) \Rightarrow \forall x \exists y S(x,y)) \equiv$$

$$\forall x \exists y P(x,y) \wedge (\neg \forall x \exists y Q(x,y) \vee \forall x \exists y S(x,y)) \equiv$$

$$\forall x \exists y P(x,y) \wedge (\exists x \forall y \neg Q(x,y) \vee \forall x \exists y S(x,y)) \equiv$$

ukoliko je moguće, uvek prvo izvući egzistencijalni, pa onda univerzalni kvantifikator

$$\exists x \forall x \exists y P(x,y) \wedge (\forall y \neg Q(x,y) \vee \forall x \exists y S(x,y))) \equiv$$

$$\exists x \forall z \exists y P(z,y) \wedge (\forall y \neg Q(x,y) \vee \forall x \exists y S(x,y))) \equiv$$

$$\exists x \forall z (\exists y P(z,y) \wedge (\forall y \neg Q(x,y) \vee \forall x \exists y S(x,y))) \equiv$$

$$\exists x \forall z \exists y (P(z,y) \wedge (\forall y \neg Q(x,y) \vee \forall x \exists y S(x,y))) \equiv$$

$$\exists x \forall z \exists y (P(z,y) \wedge (\forall y \neg Q(x,y) \vee \forall u \exists y S(u,y))) \equiv$$

$$\exists x \forall z \exists y \forall u (P(z,y) \wedge (\forall y \neg Q(x,y) \vee \exists y S(u,y))) \equiv$$

$$\exists x \forall z \exists y \forall u (P(z,y) \wedge (\forall y \neg Q(x,y) \vee \exists v S(u,v))) \equiv$$

$$\exists x \forall z \exists y \forall u \exists v (P(z,y) \wedge (\forall y \neg Q(x,y) \vee S(u,v))) \equiv$$

$$\exists x \forall z \exists y \forall u \exists v (P(z,y) \wedge (\forall w \neg Q(x,w) \vee S(u,v))) \equiv$$

$$\exists x \forall z \exists y \forall u \exists v \forall w (P(z,y) \wedge (\neg Q(x,w) \vee S(u,v)))$$

U nastavku ćemo opisati postupak skolemizacije, koji se primenjuje upravo na formule zapisane u preneks normalnoj formi.

5.3 Skolemizacija

Formula bez kvantifikatora je u **konjunktivnoj normalnoj formi** ako je oblika

$$A_1 \wedge A_2 \wedge \dots A_n$$

pri čemu je svaka od formula $A_i (1 \leq i \leq n)$ disjunkcija literala.

Formula je u **klauzalnoj formi** ukoliko je oblika

$$\forall x_1 \forall x_2 \dots \forall x_n F$$

gde je F formula bez kvantifikatora koja je u konjunktivnoj normalnoj formi.

Jasno je da za svaku rečenicu ne postoji njoj ekvivalenta rečenica koja je u klauzalnoj normalnoj formi, primer takve rečenice je $\exists x p(x)$.

Međutim, za svaku rečenicu A postoji formula B u klauzalnoj formi takva da je A zadovoljiva ako i samo ako je B.

Transformacija rečenice A u formulu B podrazumeva eliminaciju egzistencijalnih kvantifikatora.

Skolemizacija je postupak eliminisanja egzistencijalnih kvantifikatora, koji se zasniva na izmeni početne signature dodavanjem novih funkcijskih simbola i simbola konstanti.

- Ukoliko je formula oblika $\exists x A$ signaturi dodajemo novi simbol konstante c. Promenljivu x zamenjujemo simbolom c, brišemo egzistencijalni kvantifikator i posmatramo formulu $A[x \to c]$
- Ukoliko je formula oblika $\forall x_1 \forall x_2 \ldots \forall x_n \exists y A$, signaturi dodajemo novi funkcijski simbol f arnosti n, promenljivu y zamenjujemo funkcijskim simbolom f i dobijamo formulu $\forall x_1 \forall x_2 \ldots \forall x_n A[y \to f(x_1, x_2, \ldots, x_n)]$

Primer 5.7 *Formulu*

$$F \equiv \forall x \exists y P(x,y) \wedge (\forall x \exists y Q(x,y) \Rightarrow \forall x \exists y S(x,y))$$

zapiši u Skolemovoj normalnoj formi.

Najpre tražimo preneks normalnu formu formule F:

$$\forall x \exists y P(x,y) \wedge (\forall x \exists y Q(x,y) \Rightarrow \forall x \exists y S(x,y)) \equiv$$

$$\forall x \exists y P(x,y) \wedge (\neg \forall x \exists y Q(x,y) \vee \forall x \exists y S(x,y)) \equiv$$

$$\forall x \exists y P(x,y) \wedge (\exists x \forall y \neg Q(x,y) \vee \forall x \exists y S(x,y)) \equiv$$

ukoliko je moguće, uvek prvo izvući egzistencijalni, pa onda univerzalni kvantifikator

$$\exists x \forall x \exists y P(x,y) \wedge (\forall y \neg Q(x,y) \vee \forall x \exists y S(x,y))) \equiv$$

$$\exists x \forall z \exists y P(z,y) \wedge (\forall y \neg Q(x,y) \vee \forall x \exists y S(x,y))) \equiv$$

$$\exists x \forall z (\exists y P(z,y) \wedge (\forall y \neg Q(x,y) \vee \forall x \exists y S(x,y))) \equiv$$

$$\exists x \forall z \exists y (P(z,y) \wedge (\forall y \neg Q(x,y) \vee \forall x \exists y S(x,y))) \equiv$$

$$\exists x \forall z \exists y (P(z,y) \wedge (\forall y \neg Q(x,y) \vee \forall u \exists y S(u,y))) \equiv$$

$$\exists x \forall z \exists y \forall u (P(z,y) \land (\forall y \neg Q(x,y) \lor \exists y S(u,y))) \equiv$$

$$\exists x \forall z \exists y \forall u (P(z,y) \land (\forall y \neg Q(x,y) \lor \exists v S(u,v))) \equiv$$

$$\exists x \forall z \exists y \forall u \exists v (P(z,y) \land (\forall y \neg Q(x,y) \lor S(u,v))) \equiv$$

$$\exists x \forall z \exists y \forall u \exists v (P(z,y) \land (\forall w \neg Q(x,w) \lor S(u,v))) \equiv$$

$$\exists x \forall z \exists y \forall u \exists v \forall w (P(z,y) \land (\neg Q(x,w) \lor S(u,v)))$$

Zatim primenjujemo postupak skolemizacije:
Nakon zamene promenljivih x, y i v, dobijamo F u Skolemovoj normalnoj formi

$$\forall z \forall u \forall w (P(z, f(z)) \land (\neg Q(c,w) \lor S(u, g(z,u))))$$

5.4 Supstitucija

U predikatskoj logici razmatramo supstituciju (zamenu) kao preslikavanje koje promenljivama dodeljuje termove.

Supstitucija za term

Term dobijen supstitucijom promenljive x termom t_x u termu t označavamo sa $t[x \to t_x]$ i definišemo na sledeći način:

- ako je t simbol konstante $t[x \to t_x] = t$;
- ako je t promenljiva:
  - ako je $t = x$, onda je $t[x \to t_x] = t_x$;
  - ako je $t = y$, gde je $y \neq x$, onda je $t[x \to t_x] = t$;
- ako je $t = f(t_1, t_2, \ldots, t_n)$, onda je $t[x \to t_x] = f(t_1[x \to t_x], t_2[x \to t_x], \ldots, t_n[x \to t_x]])$.

Supstitucija za formulu

Formula dobijena supstitucijom promenljive x termom t_x u formuli A označavamo sa $A[x \to t_x]$ i definišemo na sledeći način:

- $\top[x \to t_x] = \top$;
- $\bot[x \to t_x] = \bot$;
- ako je $A = p(t_1, t_2, \ldots, t_n)$, onda je $A[x \to t_x] = p(t_1[x \to t_x], t_2[x \to t_x], \ldots, t_n[x \to t_x]);$

- $(\neg A)[x \rightarrow t_x] = \neg(A[x \rightarrow t_x])$
- $(A \wedge B)[x \rightarrow t_x] = (A[x \rightarrow t_x] \wedge B[x \rightarrow t_x])$
- $(A \vee B)[x \rightarrow t_x] = (A[x \rightarrow t_x] \vee B[x \rightarrow t_x])$;
- $(A \Rightarrow B)[x \rightarrow t_x] = (A[x \rightarrow t_x] \Rightarrow B[x \rightarrow t_x])$;
- $(A \Leftrightarrow B)[x \rightarrow t_x] = (A[x \rightarrow t_x] \Leftrightarrow B[x \rightarrow t_x])$;
- $(\forall xA)[x \rightarrow t_x] = \forall xA)$;
- $(\exists xA)[x \rightarrow t_x] = (\exists xA)$;
- ako je $x \neq y$, a z promenljiva koja se ne pojavljuje ni u $\forall yA$, ni u t_x, tada je

$$(\forall xA)[x \rightarrow t_x] = (\forall zA)[y \rightarrow z][x \rightarrow t_x];$$

- ako je $x \neq y$, a z promenljiva koja se ne pojavljuje ni u $(\exists yA)$, ni u t_x, tada je

$$(\exists xA)[x \rightarrow t_x] = (\exists zA)[y \rightarrow z][x \rightarrow t_x];$$

Napomena: Poslednja dva pravila obezbeđuju da $\forall yp(x,y)[x \rightarrow y]$ ne bude $\forall yp(y,y)$ već $\forall zp(y,z)$.

Sa $[x_1 \rightarrow t_1, x_2 \rightarrow t_2, \ldots, x_n \rightarrow t_n]$ označavamo *uopštenu supstituciju* σ, gde su x_i promenljive, a t_i proizvoljni termovi.

Primer 5.8 *Za* $\sigma = [x \rightarrow f(y)]$ *i* $s = g(a,x)$, *važi* $s\sigma = g(a, f(y))$. *Za* $\sigma = [x \rightarrow f(z), y \rightarrow a]$ *i* $r = g(y, g(x,y))$, *važi* $r\sigma = g(a, g(f(z), a))$. *Za* $\sigma = [x \rightarrow$ *sokrat,* $y \rightarrow$ *platon* $]$ *i* $s =$ *učitelj*(x,y) *važi* $s\sigma =$ *učitelj(sokrat, platon).*

Napomena: Kako funkcijski simboli i konstante nisu promenljive, ostaju nepromenjeni prilikom primene supstitucije.

5.5 Unifikacija

Zadatak unifikacije je da, ukoliko postoji, pronađe suspstituciju koja će dva izraza (dva terma ili dve formule) činiti jednakim. Na primer, za izraze $f(x)$ i $f(a)$ postoji supstitucija $[x \rightarrow a]$, takva da nakon njene primene izrazi postaju jednaki (u smislu identični simbol po simbol).

Ako za izraze e_1 i e_2 postoji supstitucija σ takva da važi $e_1\sigma = e_2\sigma$, tada kažemo da su e_1 i e_2 *unifikabilni* i da je supstitucija σ *unifikator* za ta dva izraza.

Primer 5.9 *Ispitati unifikabilnost izraza* $e_1 = g(x, z)$ *i* $e_2 = g(a, f(y))$ *gde su* x, y, z *promenljive, dok je* a *simbol konstante.*

Primenimo supstituciju $\sigma = [x \to a, z \to f(y)]$. *Tada je* $\sigma e_1 = \sigma e_2 = g(a, f(y))$, *pa zaključujemo da su izrazi* e_1 *i* e_2 *unifikabilni, a* σ *je jedan njihov unifikator.*

Napomena: Dva unifikabilna izraza mogu imati više unifikatora.

Ukoliko su dva izraza unifikabilna, onda postoji **najopštiji unifikator**, iz koga se mogu dobiti svi ostali kompozicijom najopštijeg unifikatora i neke sustitucije. Na primer, za izraze $f(x)$ i $g(x)$ unifikatori su: $\sigma_1 = [x \to a, y \to a]$,$\sigma_2 = [x \to b, y \to b]$ i $\sigma_3 = [x \to y]$. Međutim, σ_3 je najopštiji.

Postoji algoritam za nalaženje najopštijeg unifikatora. Primenjuje se nad skupom parova izraza

$$E = \{(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)\}$$

i ukoliko unifikator postoji, nalazi najopštiji σ za skup E takav da važi:

$$u_1\sigma = v_1\sigma, u_2\sigma = v_2\sigma, \ldots, u_n\sigma = v_n\sigma.$$

Skup E najčešće zadajemo na sledeći način:

$$E = \{u_1 \sim v_1, u_2 \sim v_2, \ldots, u_n \sim v_n\}$$

Algoritam za nalaženje najopštijeg unifikatora

Za dati skup parova izraza $E = \{s_1 \sim t_1, s_2 \sim t_2, \ldots, s_n \sim t_n\}$ predstavićemo algoritam za nalaženje najopštijeg unifikatora, koji obeležavamo sa *mgu* (engl. most general unifier).

Ulaz: skup E

Izlaz: najopštiji unifikator (ako postoji) σ takav da važi $s_1\sigma = t_1\sigma, s_2\sigma = t_2\sigma, \dots, s_n\sigma = t_n\sigma$.

1. ako postoje jednakosti koje imaju više od jednog pojavljivanja, obrisati za svaku od njih sva pojavljivanja, osim jednog;
2. obriši sve jednakosti oblika $t \sim t$;
3. ako se $t \sim x$ pojavljuje u nizu parova, gde je x promenljiva, a t nije promenljiva, onda zameni $t \sim x$ sa $x \sim t$;
4. ako postoji par takav da važi $s \sim t$, gde ni s ni t nisu promenljive, onda razmatramo sledeće slučajeve:
   - ako je $s = \varphi(u_1, u_2, \dots, u_k)$ i $t = \varphi(v_1, v_2, \dots, v_k)$, gde je φ funkcijski ili relacijski simbol, onda dodaj jednakosti $u_1 \sim v_1, u_2 \sim v_2, \dots, u_n \sim v_n$ i obriši $s \sim t$;
   - ako su s i t bilo kog drugog oblika, zaustavi rad i kao rezultat vrati *neuspeh*;
5. ako je x promenljiva i t izraz koji sadrži x, zaustavi rad i kao rezultat vrati *neuspeh*;
6. ako postoji $x \sim t$, gde je x promenljiva i t term koji ne sadrži x i x se pojavljuje i u nekim drugim jednakostima, onda primeni supstituciju $[x \to t]$ na sve druge jednakosti.

Algoritam za nalaženje najopštijeg unifikatora

Primer 5.10 *Ilustrovati rad algoritma za nalaženje najopštijeg unifikatora za sledeće parove formula:*

$$g(x, h(y, z)) \sim g(u, x),\ f(x) \sim f(h(c, v)),\ g(z, u) \sim g(y, u)$$

gde je c simbol konstante.

$$x \sim u, h(y, z) \sim x,\ f(x) \sim f(h(c, v)),\ g(z, u) \sim g(y, u)\ \ (korak\ 1)$$

$$x \sim u, x \sim h(y, z),\ f(x) \sim f(h(c, v),\ g(z, u) \sim g(y, u)\ \ (korak\ 3)$$

$$x \sim u, u \sim h(y, z),\ f(u) \sim f(h(c, v)),\ g(z, u) \sim g(y, u)\ \ (korak\ 6)$$

$$x \sim u, u \sim h(y, z),\ u \sim h(c, v),\ g(z, u) \sim g(y, u)\ \ (korak\ 4a)$$

$$x \sim u, u \sim h(y,z),\ u \sim h(c,v),\ z \sim y, u \sim u\ \ (korak\ 4a)$$

$$x \sim u, u \sim h(y,z),\ u \sim h(c,v),\ z \sim y\ \ (korak\ 2)$$

$$x \sim h(y,z), u \sim h(y,z),\ h(y,z) \sim h(c,v),\ z \sim y\ \ (korak\ 6)$$

$$x \sim h(y,z), u \sim h(y,z),\ y \sim c, z \sim v,\ z \sim y\ \ (korak\ 4a)$$

$$x \sim h(c,z), u \sim h(c,z),\ y \sim c, z \sim v,\ z \sim c\ \ (korak\ 6)$$

$$x \sim h(c,c), u \sim h(c,c),\ y \sim c, c \sim v,\ z \sim c\ \ (korak\ 6)$$

$$x \sim h(c,c), u \sim h(c,c),\ y \sim c, v \sim c,\ z \sim c\ \ (korak\ 2)$$

Najopštiji unifikator je:

$$\sigma = [x \to h(c,c), u \to h(c,c), y \to c, v \to c, z \to c].$$

5.6 Metod rezolucije u predikatskoj logici

Glavni cilj metoda rezolucije je, kao i u iskaznoj logici, izvođenje zaključaka (logičkih posledica) formule zapisane u klauzalnoj formi. Pravilo rezolucije u predikatskoj logici je opštije. Umesto da zahteva da u dve klauze postoje komplementni literali, pravilo zahteva da u dve klauze postoje literali A' i $\neg A''$ takvi da su atomičke formule A' i A'' unifikabilne.

Pravilo rezolucije za predikatsku logiku zapisujemo na sledeći način:

$$\frac{(\Gamma' \vee \mathcal{A}')\sigma \quad (\Gamma'' \vee \neg\mathcal{A}'')\sigma}{(\Gamma' \vee \Gamma'')\sigma} \quad (res)$$

gde su Γ' i Γ'' klauze, a σ je najopštiji unifikator za atomičke formule $\mathcal{A}'$ i $\mathcal{A}''$.

Pretpostavimo da su date formule:

$$otac(Vuk, y) \Rightarrow roditelj(Vuk, y)\ i\ roditelj(x, Jakov) \Rightarrow predak(x, Jakov).$$

Najpre ćemo navedene formule prevesti u klauzalnu formu:

$$\neg otac(Vuk, y) \vee roditelj(Vuk, y)\ i\ \neg roditelj(x, Jakov) \vee predak(x, Jakov)$$

Primetimo da su formule

$$roditelj(Vuk, y)\ i\ roditelj(x, Jakov)$$

unifikabilne. Primer supstitucije je $\sigma = [x \to Vuk, y \to Jakov]$. Nakon unifikacije, dobijamo formule:

$$\neg otac(Vuk, Jakov) \vee roditelj(Vuk, Jakov)\ i\ \neg roditelj(Vuk, Jakov) \vee predak(Vuk, Jakov).$$

Primenom pravila rezolucije zaključujemo da važi:

$$\neg otac(Vuk, Jakov) \vee predak(Vuk, Jakov).$$

Metod rezolucije, kao i u iskaznoj logici, sastoji se od uzastopnog primenjivanja pravila rezolucije nad skupom klauza. Da bismo primenili metod rezolucije, potrebno je pre svega formulu prevesti u preneks normalnu formu. Nakon toga potrebno je, koriteći postupak skolemizacije, eliminisati egzistencijalne kvantifikatore. Zatim je uz odgovarajuću supstituciju moguće primeniti metod rezolucije.

Primer 5.11 *Metodom rezolucije pokazati da iz pretpostavki:*

„Svaki čovek je smrtan."
„Sokrat je čovek."

sledi zaključak

„Sokrat je smrtan."

Pre svega, potrebno je date rečenice prevesti u predikatsku logiku.
Pretpostavke:

$$covek(sokrat)$$

$$\forall x(covek(X) \Rightarrow smrtan(X).$$

Zaključak:

$$smrtan(sokrat).$$

Dokazaćemo da je formula $smrtan(sokrat)$ logička posledica skupa formula

$$\{covek(sokrat), \forall x(covek(X) \Rightarrow smrtan(X)\}$$

tako što ćemo dokazati da je formula

$$(covek(sokrat) \wedge \forall x(covek(X) \Rightarrow smrtan(X))) \Rightarrow smrtan(sokrat).$$

valjana. Kako bismo dokazali vajanost formule, posmatraćemo njenu negaciju i metodom rezolucije pokušati da izvedemo praznu klauzu. Prevodimo negaciju formule u preneks normalnu formu:

$$\neg(\neg(covek(sokrat) \wedge \forall x(covek(X) \Rightarrow smrtan(X))) \vee smrtan(sokrat)).$$

$$\neg(\neg(covek(sokrat) \wedge \forall x(\neg covek(X) \vee smrtan(X))) \vee smrtan(sokrat)).$$

$$(covek(sokrat) \wedge \forall x(\neg covek(X) \vee smrtan(X)) \wedge \neg smrtan(sokrat)).$$

$$\forall x((covek(sokrat) \wedge (\neg covek(X) \vee smrtan(X)) \wedge \neg smrtan(sokrat))).$$

Uočavamo skup klauza $S = \{covek(sokrat), \{\neg covek(X), smrtan(X)\}, \neg smrtan(sokrat)\}$. Nakon supstitucije $[X \rightarrow sokrat]$, dobijamo redom klauze:

1. $covek(sokrat)$
2. $\{\neg covek(sokrat), smrtan(sokrat)\}$
3. $\neg smrtan(sokrat)$

   Primenom pravila rezolucije na prvu i drugu klauzu zaključujemo:

4. $smrtan(sokrat)$.

Primenom pravila rezolucije na treću i četvrtu klauzu izvodi se prazna klauza. Prazna klauza je indikator da negacija polazne formule nije zadovoljiva, što znači da je polazna formula valjana.

Napomena: Izvođenje zaključaka iz datih pretpostavki ćemo svoditi na dokazivanje valjanosti odgovarajuće formule, kao u prethodnom primeru.

Algoritam metoda rezolucije

Neka je data predikatska formula F. Da bismo ispitali zadovoljivost formule F upotrebom metoda rezolucije, pre same primene metoda potrebno je redom izvršiti sledeće korake:

- formulu F zapisati u preneks normalnoj formi;
- odrediti Skolemovu formu formule F;
- zapisati skup klauza S formule F i izvršiti preimenovanje promenljivih.

Nad definisanim skupom klauza $S = \{C_1, C_2, \dots, C_n\}$, dok god je moguće, ponavljati sledeće korake:

- ukoliko u tekućem skupu klauza postoji prazna klauza, vrati odgovor da je skup klauza S nezadovoljiv;
- ako se pravilom rezolucije može izvesti neka nova klauza, onda dodaj odgovarajuću rezolventu u tekući skup klaza;
- inače vrati odgovor da je skup klauza zadovoljiv.

U nastavku će biti reči o logičkom programiranju i Prolog-u, kao najznačajnijem predstavniku logičke paradigme. Programski jezik Prolog se zasniva na metodi izvođenja zaključaka primenom pravila rezolucije.

Glava 6

Prolog

6.1 Uvod u Prolog

Prolog je najznačajniji predstavnik logičke paradigme. Nastao je 1972. na Univerzitetu u Marseju. Naziv potiče od engleskih reči „PROgramming in LOGic". Teorijska osnova Prolog-a leži u predikatskoj logici (logici prvog reda).

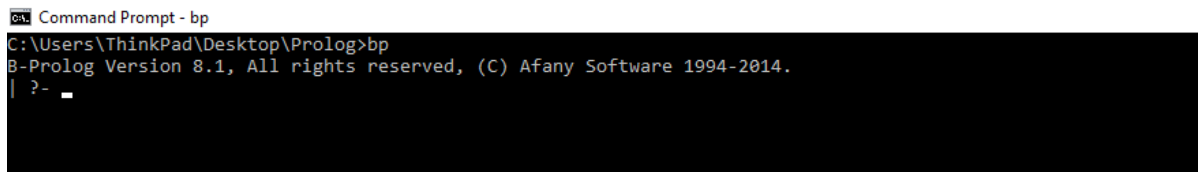
Programski jezik Prolog koristi matematičku logiku za opisivanje problema, a mehanizam za nalaženje rešenja se zasniva na metodi rezolucije. To je deklarativan programski jezik i slično kao kod upitnih jezika, u Prolog-u se definišu objekti i relacije među njima. Najveća primena Prolog-a ogleda se u:

- rešavanju problema matematičke logike;
- razumevanju prirodnog jezika;
- veštačkoj inteligenciji.

Instalacija BProlog-a

Postoje razne implementacije Prolog-a, a za potrebe ovog kursa biće korišćena implementacija pod nazivom *BProlog*. BProlog možete preuzeti sa *oficijalnog sajta pogramskog jezika*[1]. Potrebno je preuzeti i otpakovati verziju koja odgovara operativnom sistemu koji se koristi na datom računaru. U okviru foldera BProlog nalazi se izvršni fajl pod nazivom *bp*. Takođe, potreban je i *terminal*, bilo koji terminal za Linux i Mac ili Powershell ili cmd za Windows. U okviru terminala je potrebno pozicionirati se u direktorijum u okviru kog se nalazi prethodno spomenuti izvršni fajl *bp*. Nakon pokretanja izvršnog fajla prikazuje se interpreter. Pri ulasku u interpreter pojavljuje se odzivni znak *promt* `?-` pomoću kojeg se komunicira sa izvršnim fajlom. (slika 6.1)

[1] `www.picat-lang.org/bprolog/`



Slika 6.1: Prolog interpreter

Pored navedenog, potreban je i *tekst editor*. Pogodan tekst editor je Visual Studio Code jer podržava ekstenziju za označavanje sintakse. U okviru sekcije za proširenje (engl. extensions) je potrebno instalirati podršku pod nazivom VSC-Prolog. Fajlovi sa izvornim kodom u programskom jeziku BProlog se čuvaju sa ekstenzijom `.pro` ili `.pl` (preporuka je koristiti `.pro` ekstenziju jer je u operativnom sistemu Windows ekstenzija `.pl` rezervisana za programski jezik Perl).

Prolog interpreter

Neke od osnovnih komandi interpretera su:

- `help` - daje prikaz osnovnih komandi;
- `halt` - koristi se za napuštanje interpretera;
- `compile` - prevodi i generiše izvršni fajl;
- `load` - učitava izvršni fajl;
- `cl` - komanda koja objedinjuje komande *compile* i *load*.

Nakon unošenja koda u tekst editor, potrebno je izvorni kôd prevesti i učitati u okviru interpretera. Koristićemo komandu `cl` jer objedinjuje prevođenje i učitavanje. Komanda `cl` kao argument prima putanju do izvornog fajla, koja se navodi pod jednostrukim navodnicima. Ukoliko pokrenemo interpeter iz direktorijuma u kojem se nalazi izvorni fajl, dovoljno je navesti samo naziv fajla. U narednoj lekciji će biti ilustrovana uporeba komande `cl` u okviru prvog Prolog primera.

Na slici 6.2 je prikazan poziv komande `help` koja daje prikaz osnovnih komandi Prolog-a.

6.2 Osnove Prolog-a

U okviru uvoda u programski jezik Prolog, rekli smo da se Prolog zasniva na predikatskoj logici. Međutim, ne mogu se sve formule predikatske logike izraziti u Prolog-u, već samo *Hornove klauze*. To su klauze u kojima postoji najviše jedan literal koji nije pod negacijom.

```
Command Prompt - bp
C:\Users\ThinkPad\Desktop\Prolog>bp
B-Prolog Version 8.1, All rights reserved, (C) Afany Software 1994-2014.
| ?- help
 cd(Dir)              -- change the working directory to Dir. e.g. cd('c:/work')
 cl(File)             -- compile and load a program
 compile(File)        -- compile a program
 consult(F) or [F]    -- consult a proram without compiling it
 halt or ctl-z (d)    -- quit the system
 load(File)           -- load a compiled program
 notrace              -- stop the tracer
 p(T1,...,Tn)         -- start the execution of the predicate p/n
 profile              -- display the profile of the last execution
 profile_compile(F)   -- compile a program with profiling code inserted
 profile_src(File)    -- analyze predicates in a program
 read(Term)           -- read a term from a file or the keyboard
 spy(F/N)             -- add F/N to the list of spy points
 system(Com)          -- execute an OS command
 trace                -- start the tracer
 writeln(Term)        -- write a term into a file or the standard output stream
 X is Exp             -- evaluate an expression

yes
| ?-
```

Slika 6.2: Komanda `help`

Hornove klauze odgovaraju implikaciji

$$C_1, C_2, \ldots, C_n \Rightarrow C,$$

koju ćemo u Prolog-u zapisivati u obliku

$$C \Leftarrow C_1, C_2, \ldots, C_n,$$

uz zamenu znaka $\Leftarrow$ sa $:-$

$$C : -C_1, C_2, \ldots, C_n.$$

Hornove klauze zapisane na ovaj način u Prolog-u nazivamo *pravilima* koja čitamo:

$$ako\ \ C_1, C_2, \ldots, C_n\ \ onda\ \ C.$$

Klauza C predstavlja *glavu* pravila, a sve sa desne strane simbola $:-$ predstavlja *telo* pravila. Simbol zareza ',' ima ulogu operatora konjukcije.

Pravila koja nemaju telo nazivaju se *činjenice*, a pravila bez glave u Prolog-u se nazivaju *ciljevi* ili *upiti*. Pravila, činjenice i upiti se zavšavaju tačkom.

Metod rezolucije se u Prolog-u primenjuje upravo na Hornove klauze. Navedeni zapis Hornovih klauza $C : -C_1, C_2, \ldots, C_n$, ekvivalentan je sledećem zapisu u klauzalnoj formi:

$$\neg C_1 \vee \neg C_2 \vee \ldots \vee \neg C_n \vee C$$

Primena metode rezolucije se ogleda u usaglašavanju činjenica i pravila sa postavljenim ciljem. Ukoliko se cilj ispuni, Prolog daje pozitivan odgovor `yes`, a u suprotnom negativan `no`.

Početni primer

Posmatrajmo sledeće čuveno zaključivanje:

„Svaki čovek je smrtan."
„Sokrat je čovek."
Sledi, *„Sokrat je smrtan."*

U okviru početnog primera najpe ćemo dato zaključivanje prevesti u predikatsku formulu i ispitati da li iz prva dva tvrđenja sledi treće, a zatim ćemo isti problem rešiti koristeći programski jezik Prolog.

Dato zaključivanje možemo zapisati u obliku predikatske formule na sledeći način:

$$(covek(sokrat) \wedge \forall x(covek(X) \Rightarrow smrtan(X))) \Rightarrow smrtan(sokrat).$$

Sada je potrebno ispitati valjanost prethodno dobijene predikatske formule. Posmatraćemo negaciju formule i metodom rezolucije pokušati da izvedemo praznu klauzu. Prevodimo negaciju formule u preneks normalnu formu:

$$\neg(\neg(covek(sokrat) \wedge \forall x(covek(X) \Rightarrow smrtan(X))) \vee smrtan(sokrat))$$

$$\neg(\neg(covek(sokrat) \wedge \forall x(\neg covek(X) \vee smrtan(X))) \vee smrtan(sokrat))$$

$$(covek(sokrat) \wedge \forall x(\neg covek(X) \vee smrtan(X)) \wedge \neg smrtan(sokrat))$$

$$\forall x((covek(sokrat) \wedge (\neg covek(X) \vee smrtan(X)) \wedge \neg smrtan(sokrat))$$

Dobijamo skup klauza

$$S = \{covek(sokrat), \{\neg covek(X), smrtan(X)\}, \neg smrtan(sokrat)\}.$$

Nakon supstitucije $[X \rightarrow sokrat]$ i primenom pravila rezolucije izvodi se prazna klauza. Zaključujemo da negacija polazne formule nije zadovoljiva, što znači da je početna formula valjana.

Sada ćemo napisati Prolog program koji na osnovu prve dve rečenice proverava istinitost treće, tačnije da li je Sokrat smrtan. Rečenica *„Sokrat je smrtan"* se u Prolog-u tumači kao činjenica, rečenica *„Svaki čovek je smrtan"* predstavlja pravilo, dok je traženi zaključak *„Sokrat je smrtan"* zapravo cilj koji Prolog treba da ispuni.

U nastavku je prikazan primer 6.1 u okviru kog su navedeni data činjenica i pravilo.

Primer 6.1: Čuveno logičko zaključivanje

```
1 covek(sokrat).
2 smrtan(X):-covek(X).
```

```
B-Prolog Version 8.1, All rights reserved, (C) Afany Software 1994-2014.
| ?- cl('prvi.pro')
Compiling::prvi.pro
compiled in 1 milliseconds
loading...

yes
| ?-
```

Slika 6.3: Prevođenje Prolog programa

```
| ?- smrtan(sokrat)

yes
| ?- _
```

Slika 6.4: Postavljanje upita u okviru Prolog interpretera

Nakon što je u okviru tekst editora unet izvorni kôd, potrebno je isti prevesti u okviru interpretera kao na slici 6.3

Nakon uspešnog prevođenja postavljamo upit (slika 6.4).

Na osnovu zadate činjenice, paravila i upita, mehanizam Prolog-a formira skup klauza $S = \{covek(sokrat), \{\neg covek(X), smrtan(X)\}, \neg smrtan(sokrat)\}$ analogno postupku opisanom u uvodnom delu primera. Zatim primenjuje metod rezolucije, i vraća odgovor `yes`.

6.3 Sintaksa

Programski jezik Prolog ne pravi razliku između programa i podataka. Zasniva se na termu, kao osnovnoj strukturi podataka [15].

Komentari

U programskom jeziku Prolog podržane su dve vrste komentara:

- jednolinijski komentar

```
1 % Ovo je komentar
```

- višelinijski komentar

```
1  /*
2 Ovo je viselinijski
3 komentar
4 */
```

Termovi

Postoje tri vrste termova:

- **konstante**:

  * **atomi** - stringovi koji počinju malim slovom. Sastoje se od malih i velikih slova, cifara i specijalnog karaktera `_` . Atom se može navesti i kao niz slova pod apostrofom, i tada je dozvoljeno da početno slovo bude veliko. Primeri atoma:

    `sokrat, aB , c_12, 'I ovo je atom u Prolog-u';`

  * **brojevi** - podržani su celi i realni brojevi. Primeri brojeva:

    `-15 , 1, 88.9, 6.02e-23 ;`

- **promenljive** - stringovi koji se sastoje od velikih i malih slova, brojeva i donje crte. Promenljive počinju velikim slovom ili `_`. Anonimnu promenljivu označavamo sa `_` i koristimo kada vrednost promenljive nije od interesa. Primeri promenljivih:

  `X , Y, _101 , _ ;`

- **kompozitni termovi** - struktura oblika $f(t_1, t_2, \ldots, t_n)$, gde je f funktor (ili funkcijski simbol), n predstavlja arnost, a $t_1, t_2, \ldots, t_n$ su termovi. Primeri kompozitnih termova:

  `covek(sokrat) , ucitelj(sokrat,X).`

Napomena: Drugi naziv za kompozitne termove koji se često koristi je **struktura**.

Program

Prolog program je sekvenca Hornovih klauza. U uvodnom delu smo napomenuli da Hornove klauze mogu biti činjenice, pravila i upiti, a u nastavku je data formalna definicija ove tri vrste Hornovih klauza:

- **činjenice** - vrsta atomičke formule oblika $p(t_1, t_2, \ldots, t_n)$, gde je p predikat arnosti n koji se primenjuje nad termovima $(t_1, t_2, \ldots, t_n)$. Činjenice opisuju svojstva i relacije između objekata, kao na primer:

Primer 6.2: Zadavanje činjenica

```
1 covek(sokrat).
2 ucitelj(sokrat,platon).
```

- **pravila** - opšti oblik zapisujemo u formi $C : -C_1, C_2, \ldots, C_n$, gde su $C, C_1, C_2, \ldots, C_n$ atomične formule. Na osnovu činjenica koje važe sa desne strane, izvodimo zaključak koji se nalazi sa leve, kao na primer:

Primer 6.3: Zadavanje pravila

```
smrtan(X):-sokrat(X).
```

- **upiti** ili **ciljevi** - postavljaju se o objektima i odnosima među njima. Upite postavljamo iz interpretera, kao na primer:

```
|?-smrtan(sokrat).
```

Činjenice i pravila čine *bazu znanja*, sa kojom Prolog progam komunicira koristeći upite.

6.4 Sistemski operatori i predikat odsecanja

Programski jezik Prolog sadrži brojne sistemske (ugrađene) predikate koji doprinose algoritmičnosti jezika. Sistemski predikati narušavaju deklarativan stil programiranja i mogu proizvesti bočne efekte. Bez sistemskih predikata Prolog je *čist* deklarativan jezik. U Prolog-u postoje brojni sistemski (ugrađeni) predikati, a u nastavku su prikazani neki od osnovnih koji su realizovani kao operatori.

Operator unifikacije

Za razliku od proceduralnih programskih jezika kod kojih se osnovni način izvršavanja programa zasniva na dodeljivanju vrednosti promenljivama, u Prolog-u se koristi *unifikacija* kao generalni tip dodele. Unifikacija je jedna od najznačajnijih operacija nad termima. Simbol za ovu operaciju je =.

Unifikacija nad termima T i S se vrši na način koji je opisan u nastavku.

- Ako su T i S konstante, unifikuju se ako predstavljaju istu konstantu.
- Ako je S promenljiva, a T proizvoljan objekat, unifikacija se vrši tako što se termu S dodeli T. Obrnuto, ako je S proizvoljan objekat, a T promenljiva, onda T primi vrednost terma S.
- Ako su S i T strukture, unifikacija se može izvršiti ako:
  - imaju istu arnost i jednake funktore;
  - sve odgovarajuće komponente se mogu unifikovati.

Primer 6.4. *Neka su date dve strukture:*

Primer 6.4: Struktura predmeti

```
1 predmeti(programskeP,X,bazeP,Y).
2 predmeti(Z,programiranje,bazeP,racunarskeM)
```

U Prolog interpreteru postaviti upit:

```
-? predmeti(programskeP,X,bazeP,Y)=
predmeti(Z,programiranje,bazeP,racunarskeM).
```

Prolog nastoji da izjednači date strukture, tačnije da instancira promenljive odgovarajućim konstantama. Rezultat postavljenog upita je prikazan u nastavku.

```
?- predmeti(programskeP,X,bazeP,Y)=
    predmeti(Z,programiranje,bazeP,racunarskeM)
X =programiranje
Y =racunarskeM
Z =programskeP
yes
```

Unifikacija je uspešna.

Primer 6.5 *Neka su date dve strukture:*

Primer 6.5: Struktura umetnik

```
1 umetnik(glumac,X,slikar).
2 umetnik(reditelj,scenarista,Y).
```

U Prolog interpreteru postaviti upit:

```
?-umetnik(glumac,X,slikar)= umetnik(reditelj,scenarista,Y).
```

Prolog nastoji da izjednači date strukture. Rezultat postavljenog upita je prikazan u nastavku.

```
?- umetnik(glumac,x,slikar)=umetnik(reditelj,scenarista,Y)
no
```

Unifikacija je neuspešna jer argumetni `glumac` i `reditelj` ne predstavljaju istu konstantu.

Napomena: Kao što se može primetiti u prethodnim primerima, operator unifikacije se primenjuje i na promenljive koje nisu konkretizovane.

Operator inverzan operatoru unifikacije je \=.

Aritmetički operatori

Aritmetički operatori podržani u Prolog-u su:

- sabiranje(+);
- oduzimanje (-);
- množenje (*);
- deljenje (\);
- celobrojno deljenje (`div`);
- ostatak pri deljenju (`mod`).

Za izračunavanje aritmetičkog izraza, a zatim dodeljivanje njegove vrednosti promenljivoj, u Prolog-u se koristi ugrađeni predikat `is`. Sintaksa je sledeća: `X is Y`, gde je `X` promenljiva, a `Y` aritmetički izraz.

Operatori poređenja

Operatori poređenja primenjuju se na konkretizovane (brojčane) promenljive. U Prolog-u su podržani sledeći relacioni operatori: `>`, `<`, `>=` i `=<`. Pored navedenih, podržani su i operatori za proveru aritmetičke, odnosno identične jednakosti i nejednakosti:

- aritmetička jednakost (`=:=`);
- aritmetička nejednakost (`=\=`);
- identično jednaki termovi (`==`);
- nisu identično jednaki termovi (`\==`).

U nastavku je ilustrovana razlika između operatora unifikacije, operatora za proveru artitmečke i identične jednakosti kroz njihove pozive u okviru interpretera.

```
| ?-  1 + 2 == 2 + 1
no
| ?- 1 + 2 =:= 2 + 1
yes
| ?-  1 + 2 = 2 + 1
no
| ?-   1 + X = Y + 2
X = 2
Y = 1
yes
```

Operator negacije

Oznaka za operator negacije je `not`. Upotrebljava se kao prefiksni operator: `not(X)`, gde `X` predstavlja određeni cilj. Važno je napomenuti da operator `not` nije operator u smislu logičke negacije, već važi da `not(X)` uspeva, samo ukoliko cilj `X` ne uspeva.

Neka je u bazi znanja data činjenica `covek(sokrat)`. Ilustracija upotrebe operatora `not` u okviru interpretera je prikazana u nastavku.

```
| ?-  not(covek(sokrat))
 no
| ?- not(covek(platon))
yes
```

Operator sečenja

Operator sečenja (engl. cut operator) omogućava brže izvršavanje programa i uštedu memorijskog prostora kroz eksplicitnu kontrolu traženja sa vraćanjem . Oznaka za operator sečenja je znak uzvika `!` i to je cilj koji uvek uspeva. Neka je u okviru baze znanja dato sledeće pravilo:

```
H :- B1,B2, ..., Bm, !, ..., Bn.
```

Kada se neki cilj `G` unifikuje sa `H`, primenjuje se prethodno navedeno pravilo. Ukoliko su uspeli podciljevi `B1,B2, ..., Bm` , uspeva i predikat `!` i onemogućava dalje traženje alternativnih rešenja. Takođe, predikat odsecanja onemogućava i unifikovanje cilja `G` sa glavom nekog drugog predikata koji se u bazi znanja nalazi ispod navedenog pravila.

6.5 Izračunavanje odgovora u Prolog-u

Naredni primer 6.6 ilustruje kako Prolog na osnovu date baze znanja odgovara na potavljeno pitanje.

Primer 6.6: Baza znanja o teniskim turnirima

```
1 grendSlem(usOpen).                   % klauza 1
2 grendSlem(vimbldon).                 % klauza 2
3
4 masters1000(rim).                    % klauza 3
5 masters1000(sangaj).                 % klauza 4
6 masters1000(mandrid).                % klauza 5
7
8 evropa(vimbldon).                    % klauza 6
9 evropa(rolanGaros).                  % klauza 7
```

```
10 evropa(rim).                          % klauza 8
11 evropa(madrid).                       % klauza 9
12
13 azija(sangaj).                        % klauza 10
14
15 amerika(usOpen).                      % klauza 11
16
17 teniski_turnir(Y):- grendSlem(Y).     % klauza 12
18 teniski_turnir(Y):-masters1000(Y).    % klauza 13
```

U okviru interpretera postavlja se cilj `?-teniski_turnir(X), azija(X).` Od Prolog-a se očekuje da odgovori koji teniski turnir se igra u Aziji, ukoliko takav turnir postoji, u suprotnom da vrati neuspeh.

Naš glavni cilj sastoji se od dva podcilja: `teniski_turnir(X)` i `azija(X)`. Glavni cilj će biti ispunjen samo ako budu ispunjena oba potcilja.

U nastavku je prikazan način na koji Prolog teži da ispuni zadati cilj tako što pokušava da dokaže saglasnost sa bazom podataka [12].

- Definiše se polazni cilj:

  `teniski_turnir(X), azija(X).`

- Pošto se polazni cilj sastoji od dva podcilja, Prolog prvo pokušava da zadovolji prvi podcilj tražeći klauzu u programu počev od vrha ka dnu, takvu da se može unifikovati sa `teniski_turnir(X)`. Pronalazi klauzu broj 12:

  `teniski_turnir(Y):- grendSlem(Y).`

- Novi cilj postaje:

  `grendSlem(X), azija(X).`

- Prolog pokušava da zadovolji prvi podcilj `grendSlem(X)`. Pronalazi klauzu broj 1 `grendSlem(usOpen)`, pa novi cilj postaje:

  `azija(usOpen).`

- Prolog prolazi kroz bazu znanja i kako ne pronalazi klauzu koja se može unifikovati sa datim ciljem, vraća neuspeh. Dodeljena vrednost promenljivoj `X` se poništava, a Prolog se vraća na prethodni podcilj:

  `grendSlem(X), azija(X).`

- Sada pronalazi klauzu broj 2 `grendSlem(vimbldon)`, pa novi podcilj postaje:

```
azija(vimbldon).
```

- Prolog prolazi kroz bazu znanja i kako ne pronalazi klauzu koja se može unifikovati sa datim ciljem, vraća neuspeh. Dodeljena vrednost promenljivoj `X` se poništava, a Prolog se vraća na prethodni podcilj:

```
grendSlem(X), azija(X).
```

- Prolog skenira bazu znanja počev od klauze broj 3 i kako ne uspeva da ispuni podcilj `grenSlem(X)`, vraća se na inicijalni cilj:

```
teniski_turnir(X), azija(X).
```

- Sada se baza skenira počevši od prve klauze koja se nalazi ispod klauze broj 12 i Prolog pokušava da zadovolji prvi podcilj `teniski_turnir(X)` i pronalazi klauzu broj 13:

```
teniski_turnir(Y):-masters1000(Y).
```

- Novi cilj postaje:

```
masters1000(X), azija(X).
```

- Prolog pokušava da zadovolji prvi podcilj `masters1000(X)`, u bazi znanja nailazi na klauzu broj 3 `masters1000(rim)` i nakon unifikacije, novi cilj postaje:

```
azija(rim).
```

- Prolog prolazi kroz bazu znanja i kako ne pronalazi klauzu koja se može unifikovati sa datim ciljem, vraća neuspeh. Dodeljena vrednost promenljivoj X se poništava, a Prolog se vraća na prethodni podcilj:

```
masters1000(X), azija(X).
```

- Pretražuje se počev od klauze broj 4 u cilju unifikacije prvog podcilja.`masters1000(X)` se unifikuje upravo sa klauzom broj 4, pa novi cilj postaje:

```
azija(sangaj).
```

- Skenirajući klauze programa, Prolog nailazi na klauzu `azija(sangaj)`, koja predstavlja činjenicu. Činjenice su u Prolog-u uvek tačne, pa je novi cilj koji je potrebno zadovoljiti prazan (prazan cilj označavamo sa `[ ]`). Upravo je prazan cilj indikator uspeha i promenljiva `X` se postavlja na `sangaj` , tj. `X=sangaj`.

- Prolog se vraća na cilj `masters1000(X), azija(X)` i iznova će pokušati da ga zadovolji, sve dok ne iscrpi sve mogućnosti. U cilju zadovoljavanja prvog podcilja, Prolog nastavlja skeniranje baze počev od klauze broj 5 `masters1000(madrid)` i nakon unifikacije novi cilj postaje:

```
azija(madrid).
```

- Prolog prolazi kroz bazu znanja i kako ne pronalazi klauzu koja se može unifikovati sa datim ciljem, vraća neuspeh. Time su iscrpljene sve mogućnosti baze znanja, pa je jedino rešenje `X=sangaj`.

Prilikom izračunavanja odgovora u Prolog-u, može doći do nekog od naredna dva slučaja:

- proces izračunavanja se uspešno zavšava, cilj je ispunjen i Prolog daje određeno rešenje;

- proces izračunavanja se neuspešno završava i ne dobija se rešenje.

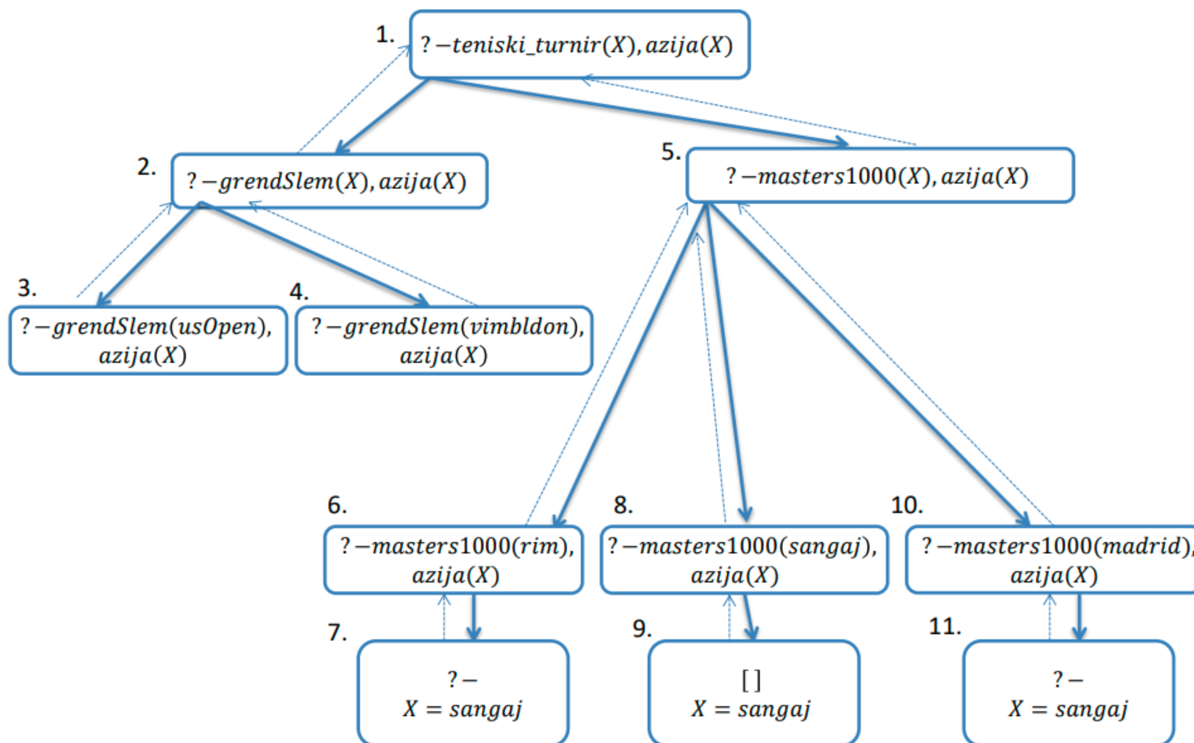
U slučaju da se glavni cilj sastoji od više podciljeva, Prolog ispunjava glavni cilj tako što pokušava da zadovolji svaki od podciljeva, počev od prvog sa leve strane. Ukoliko se neki od podciljeva završava neuspehom, Prolog se vraća na prethodni podcilj i teži da ga zadovolji za neke druge vrednosti. Ovaj postupak se ponavlja sve dok se ne iscrpe sve mogućnosti. Postupak traženja sa vraćanjem (engl. backtracking) omogućava nalaženje svih rešenja iz skupa mogućih rešenja.

Napomena: Za postavljeni upit `?- grenSlem(australianOpen)` Prolog vraća neuspeh jer nema te činjenice u bazi. U Prolog-u važi *pretpostavka zatvorenosti* (engl. Closed-World Assumption), tj. smatra se netačnim sve što eksplicitno nije navedeno kao tačno.

Stablo izračunavanja odgovora

Opisani postupak izračunavanja odgovora iz primera 6.6 u Prolog-u se može slikovito prikazati stablom pretraživanja kao na slici 6.5.

Stablo se sastoji od *grana* i *čvorova*. Početni čvor stabla koji sadrži cilj naziva se *koren* stabla. Svi čvorovi koji su označeni upitom nazivaju se nezavršnim. *Listovi* u stablu pretraživanja nazivaju se završnim čvorovima. Razlikujemo listove označene sa `[ ]`, koji



Slika 6.5: Stablo izračunavanja odgovora

predstavljaju čvorove uspeha i listove označene sa ?-, koji predstavljaju čvorove neuspeha. Proces traženja rešenja se odvija kretanjem od korena stabla ka listovima, ali i od listova ka vrhu (predstavlja traženje sa vraćanjem). Na slici 6.5 je strelicama prikazan smer traženja rešenja. Možemo uočiti jedan čvor uspeha, što znači da je Prolog uspešno pronašao samo jedno rešenje.

6.6 Rekurzija u Prolog-u

Iterativni postupci se ne mogu direktno realizovati u Prolog-u, već posredno preko rekurzije. Na primeru porodičnog stabla ilustrovaćemo primenu rekurzije u Prolog-u.

Primer 6.7 *U nastavku je data baza znanja kreirana na osnovu porodičnog stabla Nemanjića. Baza sadrži svojstvo da li je neki član porodice žena ili muškarac, kao i relaciju roditelj. Potrebno je definisati relaciju predak.*

Primer 6.7: Baza znanja o odnosima u porodici Nemanjić

```
1 /*cinjenice koje definisu svojstvo muskarca i zene*/
2 zensko(ana).
3 zensko(jefimija).
4 zensko(jevrosima).
5
6 musko(zavida).
```

```
7  musko(stefanNemanja).
8  musko(stefanPrvovencani).
9
10 /* cinjenice koje definisu odnose u porodici */
11 /* predikat roditelj(X,Y) ima znacenje da je X roditelj od Y */
12 roditelj(stefanPrvovencani,urosI).
13 roditelj(urosI,milutin)
14 roditelj(stefanNemanja, stefanPrvovencani).
15 roditelj(zavida, stefanNemanja).
```

Na osnovu date baze znanja je jasno da je `X` u relaciji `predak` sa `Z` ukoliko važi da je `X` roditelj od `Z`, što definišemo na sledeći način:

```
predak(X,Z): roditelj(X,Z).
```

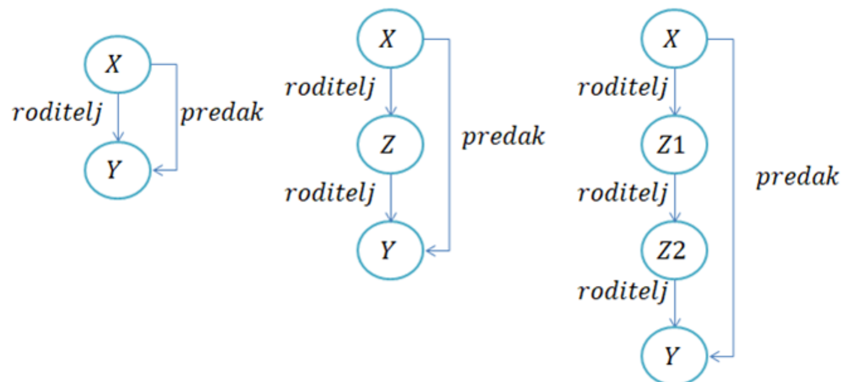
Međutim, `X` će biti u relaciji predak sa `Y` i ukoliko postoji `Z` takav da je `X` roditelj od `Z`, a `Z` roditelj od `Y`, sto definišemo na sledeći način:

```
predak(X,Y): roditelj(X,Z), roditelj(Z,Y).
```

Analogno će važiti i:

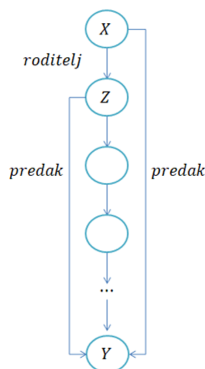
```
predak(X,Y): roditelj(X,Z1), roditelj(Z1,Z2), roditelj(Z2,Y).
```

Na slici 6.6 dat je grafički prikaz relacije `predak`.



Slika 6.6: Grafički prikaz relacije predak

Ovakav način rešavanja nas ograničava, drugim rečima ne možemo zadovoljiti predikat za proizvoljnu dubinu stabla. Rešićemo ovaj problem na elegantniji način koristeći rekurziju, tj. definisaćemo pravilo po kome je roditelj pretka takođe predak [16]. Na slici 6.7 dat je grafički prikaz rekurzivne relacije predak.



Slika 6.7: Grafički prikaz rekurzivne relacije predak

Sada je potrebno prethodno definisanoj bazi znanja dodati predikat `predak`.

Primer 6.8: Predikat `predak`

```
1 /*relacija predak ce u potpunosti biti definisana zadavanjem
     sledeca dva pravila:
2 1. roditelj je predak;
3 2. roditelj pretka je takodje predak. */
4
5 predak(X,Y):- roditelj(X,Y).
6 predak(X,Y):- roditelj(X,Z), predak(Z,Y).
```

U okviru interpretera postavljmo upit `?-predak(stefanPrvovencani,Y)`.

```
| ?- predak(stefanPrvovencani,Y).
Y = urosI ?;
Y = milutin ?;
no
```

6.7 Liste

Lista je složena struktura podataka, koja predstavlja niz uređenih elemenata proizvoljne dužine. Element liste može biti proizvoljan term, čak i druga lista. Redosled elemenata je bitan i pristup se vrši redom, počevši od prvog elementa.

Primer liste u Prolog-u: `L= [1,2,[],Y,[3,4]]`

Listu definišemo rekurzivno na sledeći način:

- lista može biti prazna u oznaci `[ ]`;
- lista se može konstruisati od glave i repa `.(G,R)`, gde `.` predstavlja konstruktor za listu, `G` je proizvoljan term, a `R` je lista termova koja može biti i prazna.

Primeri zadavanja listi koristeći rekurzivnu definiciju:

1. `[ ]` - prazna lista
2. `.(a,[ ])` - jednočlana lista, a je term i predstavlja glavu liste, `[ ]` je rep
3. `.(a,.(b,[ ]))` - lista koja ima dva argumenta

Na ovaj način možemo zadati listu proizvoljne dužine, međutim, ovakav zapis je nepraktičan. Zato ćemo koristi kraći zapis. Na primer, zapis liste `.(a,.(b,.(c,[])))` je ekvivalentan zapisu `[a,b,c]`.

Prolog podržava i zapis liste u kome su jasno razdvojene komponente glave i repa: `L=[G|R]`. Takođe, ispred karaktera `|` možemo postaviti proizvoljan broj elemenata, a zatim listu preostalih elemenata, kao na primer:

```
[1,a,3]=[1|[a,3]]=[1,a|[3]]=[1,a,3|[]]
```

Liste predstavljaju najvažniju strukturu podataka Prolog-u, a rekurzija je glavni metod koji se primenjuje prilikom rada sa listama. Obrada cele liste se najčešće svodi na obradu repa liste, koji takođe predstavlja listu, ali sa manjim brojem argumenata.

Osnovna pravila za rad sa listama

U nastavku ćemo proučiti neke od osnovnih predikata za rad sa listama:

- kako ispitati pripadnost elemenata listi;
- kako spojiti dve liste;
- kako iz liste izbrisati element;
- kako ubaciti u listu element na proizvoljnu poziciju;
- kako datu listu razdvojiti na jedan element i ostatak liste.

Svaki od navedenih zadataka ćemo rešavati primenom rekurzije.

Član liste

Primer 6.9 *Napisati Prolog program koji određuje da li je dati element član liste.*

Primer 6.9: Predikat clan

```
1 /* dati element pripada listi ukoliko moze da se unifikuje sa
     glavom ili ukoliko se nalazi u repu polazne liste */
2 clan(X,[X|_]). /* koristimo anonimnu promenljivu za oznacvanje
     repa, jer nam rep polazne liste nije od interesa */
3 clan(X,[_|R]) :- clan(X,R). /* ukoliko se dati element nije
     unifikovao sa glavom liste, rekurzivno pozivamo za rep */
```

Poziv predikata: `clan(element,lista).`

Napomena: Predikat `clan` možemo pozvati i na sledeći način: `clan(X,lista)` i time generisati sve elemente liste.

```
| ?- clan(3,[1,4,5,3])
yes
| ?- clan(2,[1,3,5])
no
| ?- clan(X,[2,3,4])
X = 2 ?;
X = 3 ?;
X = 4 ?;
no
```

Spajanje dve liste

Primer 6.10 *Date su dve liste. Napisati Prolog program koji formira listu dodavanjem druge liste prvoj.*

Primer 6.10: Predikat spoji

```
1 /* bazni slucaj predstavlja cinjenicu - ukoliko je jedna od listi
     prazna, rezultat spajanja je druga lista */
2 spoji([],X,X).
3 /* glava pocetne liste je glava i rezultujuce, koju nadovezujemo
     na listu R1 koja predstavlja rezultat rekurzivnog poziva */
4 spoji([G|R], Y, [G|R1]):-spoji(R,Y,R1).
```

Poziv predikata: `spoji(prva,druga,X).`

```
|?- spoji([1,2,3],[4,5,6],X).
X = [1,2,3,4,5,6]
yes
| ?- spoji(X,Y,[1,2,3])
X = []
Y = [1,2,3] ?;
X = [1]
Y = [2,3] ?;
X = [1,2]
Y = [3] ?;
X = [1,2,3]
Y = [] ?;
no
```

Brisanje iz liste

Primer 6.11 *Data je lista. Potrebno je napisati predikat za brisanje svih elemenata liste jednakih datom elementu.*

Primer 6.11: Predikat obrisi

```
1 /* bazni sucaj, ukoliko pokusamo da obrisemo element iz prazne
     liste, rezultat je prazna lista */
2 obrisi(X, [], []).
3 /* ukoliko se element moze unifikovati sa glavom ulazne listom,
     rekurzivno pozivamo predikat za rep ulazne liste */
4 obrisi(X, [X|R], R1):- obrisi(X, R, R1), !.
5 /* ukoliko se element ne moze unifikovati sa glavom ulazne liste,
     glava ulazne liste bice i glava rezultujuce, rekurzivno
     pozivamo predikat za rep liste */
6 obrisi(X, [G|R], [G|R1]):- G \== X, obrisi(X, R, R1).
```

Poziv predikata: `obrisi(element,lista,X).`

```
| ?- obrisi(3,[3,1,2,3],X).
X = [1,2]
yes
```

Ubacivanje elementa u listu

Primer 6.12 *Dati su element i lista. Napisati Prolog program koji formira novu listu tako što dati element ubacuje na proizvoljnu poziciju date liste.*

Primer 6.12: Predikat ubaci

```
1 /* element koji se zadaje kao prvi argument se ubacuje u listu
     koja je drugi argument, dok treci argument predikata
     predstavlja rezultujucu listu */
2 /* u praznu listu ubacujemo element, pa je rezultujuca lista
     jednoclana  */
3 ubaci(X, [], [X]).
4 /* dodajemo element na pocetak date liste */
5 ubaci(X, [G|R], [X,G|R]).
6 /* dodajemo element u rep pocetne liste */
7 ubaci(X, [G|R], [G|R1]):-ubaci(X, R, R1).
```

Poziv predikata: `ubaci(element,lista,X).`

```
| ?-  ubaci(1,[2,3,4],X)
X = [1,2,3,4] ?;
```

```
X = [2,1,3,4] ?;
X = [2,3,1,4] ?;
X = [2,3,4,1]
yes
```

Izbor

Primer 6.13 *Napisati Prolog program koji datu listu razdvaja na jedan element i ostatak liste.*

Primer 6.13: Predikat izbor

```
1 /* glava se izdvaja kao element */
2 izbor([G|R], G, R).
3 /* glava se nalazi u ostatku liste */
4 izbor([G|R], X, [G|R1]):- izbor(R,X,R1).
```

Poziv predikata: `izbor(lista,X,L).`

```
| ?-  izbor([1,2,3,4], X,L)
X = 1
L = [2,3,4]?;
X = 2
L = [1,3,4] ?;
X = 3
L = [1,2,4] ?;
X = 4
L = [1,2,3]
```

U nastavku ćemo logičke i kombinatorne probleme rešavati korišćenjem navedenih osnovnih predikata za rad sa listama.

6.8 Rešavanje kombinatornih problema

U okviru ovog poglavlja definisaćemo pravila za rešavanje osnovnih kombinatornih problema u Prolog-u.

Permutacije

Definicija 6.1 *Neka je dat konačan skup A od n elemenata. Permutacija skupa A predstavlja niz od n elemenata tog skupa.*

Pošto ćemo ovde govoriti o permutacijama bez ponavljanja, podrazumevamo da se svaki od n elemenata skupa A pojavljuje tačno jednom. Pronaći permutacije nekog skupa od n elemenata znači pronaći sve načine na koje se n elemenata može poređati u niz.

Primer 6.14 *Definisati predikat kojim se određuju sve permutacije date liste.*

Analiza rešenja

Definisaćemo predikat `permutacija` arnosti dva, takav da prvi argument predstavlja datu listu, a drugi argument je rezultat, tj. permutacija date liste. Primer poziva upita je: `?-permutacija([1,2,3], X).` Rešenje ovog problema ćemo bazirati na dva slučaja:

1. ukoliko je lista prazna, permutacija je takođe prazna lista;
2. lista je neprazna pa se može unifikovati sa šablonom `[G|R]`. Rekurzivno ćemo primeniti predikat `permutacija` na rep liste. U listu generisanu rekuzivnim pozivom ćemo ubaciti glavu polazne liste na proizvoljnu poziciju.

Primer 6.14: Permutacije bez ponavljanja

```
1   /*[1,2,3] -> 1 [2,3]      1 [3,2]   izdvojimo glavu i zatim
       rekurzivno odredimo permutacije repa
2              -> [1,2,3]      [1,3,2]     zatim ubacujemo glavu
                  redom na sve pozicije rezultujucih listi
                  rekurzivnog poziva
3              -> [2,1,3]      [3,1,2]
4              -> [2,3,1]      [3,2,1] */
5
6  /*pomocni predikat*/
7  ubaci(X, [], [X]).
8  ubaci(X, [G|R], [X,G|R]).
9  ubaci(X, [G|R], [G|R1]):-ubaci(X, R, R1).
10
11  /* permutacija prazne liste je prazna lista */
12  permutacije([], []).
13  /* rekurzivni poziv */
14  permutacije([G|R], P):- permutacije(R, R1), ubaci(G, R1, P).
```

```
| ?- permutacije([1,2,3],X).
X = [1,2,3] ?;
X = [2,1,3] ?;
X = [2,3,1] ?;
X = [2,3,1] ?;
X = [2,1,3] ?;
```

```
X = [2,3,1] ?;
X = [2,3,1] ?;
X = [1,2,3] ?;
X = [2,1,3] ?;
X = [2,3,1] ?;
X = [2,3,1] ?;
X = [2,1,3] ?;
   ...
```

Varijacije

Varijacije se koriste za rešavanje kombinatornih problema kod kojih je potrebno od n objekata izabrati k objekata. Kod varijacija je poredak elementa važan. U zavisnosti od toga da li je u varijacijama dozvoljeno ponavljanje elemenata ili ne, razmatraćemo varijacije sa ponavljanjem i varijacije bez ponavljanja.

Varijacije bez ponavljanja

Definicija 6.2 *Neka je dat konačan skup A od n elemenata. Varijacija bez ponavljanja $k-$te klase od n elemenata skupa A je $k-$točlani ($k \leq n$) niz različitih elemenata skupa A.*

Primer 6.15 *Napisati predikat koji generiše sve varijacije $k-$te klase za dati skup predstavljen listom L.*

Analiza rešenja

Definisaćemo predikat `varijacije` arnosti tri takav da prvi argument predstavlja datu listu, drugi je klasa varijacije i rezultujuća lista se čuva u poslednjem argumentu. Definisaćemo pomoćni predikat `izbor` koji datu listu razdvaja na jedan element i ostatak liste, zatim rekruzivno pozvati predikat `varijacije` za ostatak liste i na tako dobijene varijacije ćemo dodati prethodno izdvojeni element.

Primer 6.15: Varijacije bez ponavljanja

```
1 /* glava se izdvaja kao element */
2 izbor([G|R], G, R).
3 /* glava se nalazi u ostatku liste */
4 izbor([G|R], X, [G|R1]):- izbor(R,X,R1).
5 
6 /* izlaz iz rekurzije, varijacija nulte klase je prazna lista */
7 varijacije(L,0,[]).
8 /* ulazna lista se razdvaja na elemenat X i ostatk liste L1.
     Rekuzivno pozivamo predikat varijacije za listu L1, a zatim na
     sve dobijene varijacije dodamo izabrani element X */
```

```
9 varijacije(L,K,[X|R]):- K>0, izbor(L,X,L1), K1 is K-1, varijacije(
    L1,K1,R).
```

```
| ?- varijacije([1,2,3,4],2,X)
X = [1,2] ?;
X = [1,3] ?;
X = [1,4] ?;
X = [2,1] ?;
X = [2,3] ?;
X = [2,4] ?;
X = [3,1] ?;
X = [3,2] ?;
X = [3,4] ?;
X = [4,1] ?;
X = [4,2] ?;
X = [4,3] ?;
no
```

Varijacije sa ponavljanjem

Definicija 6.3 *Neka je dat konačan skup A od n elemenata. Varijacija sa ponavljanjem* $k-$*te klase od* n *elemenata skupa* A *je* $k-$*točlani* $(k \leq n)$ *niz elemenata skupa* A *koji mogu da se ponavljaju.*

Primer 6.16 *Napisati predikat koji generiše sve varijacije sa ponavljanjem* $k-$*te klase za za dati skup predstavljen listom* L*.*

Analiza rešenja

Analogno prethodnoj analizi, jedina razlika je u tome što sada imamo rekurzivni poziv za celu listu.

Primer 6.16: Varijacije sa ponavljanjem

```
1 varijacijeP(L,0,[]).
2  varijacijeP(L,K,[X|R]):-K>0, izbor(L,X,Ost), K1 is K-1,
     varijacijeP(L,K1,R).
```

```
|?- varijacijeP([1,2,3,4],2,X)
X = [1,1] ?;
X = [1,2] ?;
X = [1,3] ?;
X = [1,4] ?;
X = [2,1] ?;
X = [2,2] ?;
```

```
X = [2,3] ?;
X = [2,4] ?;
X = [3,1] ?;
X = [3,2] ?;
X = [3,3] ?;
X = [3,4] ?;
X = [4,1] ?;
X = [4,2] ?;
X = [4,3] ?;
X = [4,4] ?;
no
```

Kombinacije

Kombinacije se koriste za rešavanje problema kod kojih je od n objekata potrebno izabrati k. Kod kombinacija poredak elemenata nije važan. U zavisnosti od toga da li je u kombinaciji dozvoljeno ponavljanje elemenata ili ne, razmatraćemo kombinacije sa ponavljanjem i kombinacije bez ponavljanja.

Kombinacije bez ponavljanja

Definicija 6.4 *Neka je dat konačan skup A od n elemenata. Kombinacija bez ponavljanja $k-$te klase od n elemenata skupa A je bilo koji podskup od k ($k \leq n$) različitih elemenata.*

Primer 6.17 *Napisati predikat koji generiše sve kombinacije $k-$te klase za dati skup predstavljen listom L.*

Analiza rešenja

Definisaćemo predikat `kombinacije` arnosti tri takav da prvi argument predstavlja datu listu, drugi je klasa kombinacije i rezultujuća lista se čuva u poslednjem argumentu. Prilikom određivanja svih kombinacija $k-$te klase imamo dva slučaja:

1. ukoliko kombinacije sadrže glavu ulazne liste, tada kao glavu rezultujuće liste postavljamo glavu ulazne liste i rekurzivno pozivamo predikat za rep liste reda k-1;
2. ukoliko kombinacije ne sadrže glavu ulazne liste, tada rekurzivno pozivamo predikat za rep liste.

Primer 6.17: Kombinacije bez ponavljanja

```
1 /* ukoliko se trazi nulta kombinacija klase, rezultat je prazna
     lista */
2 kombinacije(L,0,[]).
```

```
/* u suprotnom listu razdvajamo na glavu i rep */
/* glavu ulazne liste postavljamo kao glavu i rezultujuce, dok rep
   rezultujuce liste predstavlja rezultat rekurzivnog poziva za
   kombinaciju reda k-1 */
kombinacije([G|R],K,[G|R1]):-K>0, K1 is K-1, kombinacije(R,K1,R1).
/*dodajemo kombinacije bez glave ulazne liste  */
kombinacije([G|R],K,L):-K>0,kombinacije(R,K,L).
```

```
| ?- kombinacije([1,2,3,4],2).
X = [1,2] ?;
X = [1,3] ?;
X = [1,4] ?;
X = [2,3] ?;
X = [2,4] ?;
X = [3,4] ?;
no
```

Kombinacije sa ponavljanjem

Definicija 6.5 *Neka je dat konačan skup A od n elemenata. Kombinacije sa ponavljanjem $k-$te klase od n elemenata skupa A je bilo koji podskup od A takav da se jedan element može ponoviti do k puta.*

Primer 6.18 *sve kombinacije $k-$te klase sa ponavljanjem za dati skup predstavljen listom L.*

Analiza rešenja

Analogno prethodnom rešenju, jedina razlika je u tome što sada rekurzivni poziv uključuje i glavu polazne liste.

Primer 6.18: Kombinacije sa ponavljanjem

```
/* kombinacije 0-te klase je prazna lista */
kombinacijaP(L,0,[]).
/* rekurzivno pozivamo predikat reda k-1 */
kombinacijaP([G|R],K,[G|R1]):-K>0, K1 is K-1, kombinacijaP([G|R],
   K1,R1).
/* dodajemo kombinacije bez glave */
kombinacijaP([G|R],K,L):-K>0,kombinacijaP(R,K,L).
```

```
| ?- kombinacijeP([1,2,3,4],2).
X = [1,1] ?;<br>
X = [1,2] ?;<br>
X = [1,3] ?;<br>
```

```
X = [1,4] ?;<br>
X = [2,2] ?;<br>
X = [2,3] ?;<br>
X = [2,4] ?;<br>
X = [3,3] ?;<br>
X = [3,4] ?;<br>
X = [4,4] ?;<br>
no
```

6.9 Rešavanje logičkih problema

Programski jezik Prolog je naročito pogodan za rešavanje problema u okviru kojih su za date entitete jasno definisane osobine i svojstva koja moraju da važe. Primeri takvih logičkih problema su: *Ajnštajnov problem kuća* i *problem misionara i ljudoždera.*

Ajnštajnov problem kuća

Potrebno je napisati program koji rešava zagonetku prikazanu u nastavku, poznatu kao Ajnštajnov problem kuća.

Formulacija problema

Postoji pet kuća, svaka različite boje u kojoj žive ljudi različitih nacionalnosti, koji piju različita pića, jedu različita jela i imaju različite kućne ljubimce. Važi sledeće:

- Englez živi u crvenoj kući
- Španac ima psa
- kafa se pije u zelenoj kući
- Ukrajinac pije čaj
- zelena kuća je odmah desno uz belu
- onaj koji jede špagete ima puža
- pica se jede u žutoj kući
- mleko se pije u srednjoj kući
- Norvežanin živi u prvoj kuci s leva
- onaj koji jede piletinu živi pored onoga koji ima lisicu

- pica se jede u kući koja je pored kuće u kojoj je konj
- onaj koji jede brokoli pije sok od narandze
- Japanac jede suši
- Norvežanin živi pored plave kuće

Čija je zebra, a ko pije vodu?

Analiza rešenja

Prilikom rešavanja logičkih zagonetki, zadatak predikata je drugačiji, odnosno njegov zadatak nije da proveri da li određena relacija važi ili da nam da neku dodatnu informaciju, već da izgradi neku vrstu strukture. Rešenje će biti lista kuća takvih da zadovoljavaju sve relacije date zadatkom. Za prikaz rešenja je pogodna lista jer ćemo moći da zaključimo koja kuća je levo, desno, ili u sredini (ovo ćemo moći da zaključimo jer lista sadrži neparan broj kuća). Spisak činjenica i pravila koje definišemo za potrebe rešavanja zagonetke prikazani su u nastavku.

- Shodno tome da za svaku kuću postoji tačno pet informacija koje je jednoznačno određuju, definišemo predikat:

  `kuca(boja, nacionalnost, jelo, pice, kucni_ljubimac).`

- Pomoćni predikat koji određuje da li se element `X` nalazi desno od `Y` u listi `L`:

  `desno(X, Y, L).`

- Predikat `pored` definiše da li se element `X` nalazi pored `Y` u datoj listi `L`:

  `pored(X,Y, L).`

- Predikat `clan` koji proverava da li je kuća element rezultujuće liste kuća:

  `clan(X, L).`

- Ključni predikat je predikat `kuce` koji rešava zagonetku, tj. vraca listu kuća `L` koje zadovoljavaju uslove zadatka:

  `kuce(L).`

- Predikat `odgovori` na osnovu liste `L` koja sadrži rešanja, daje odgovor na postavljeno pitanje:

  `odgovori(X,Y).`

Primer 6.19: Ajnštajnov problem kuća

```
/* pomocni predikat clan koji proverava da li je kuca X  clan liste koja je data drugim argumentom */
clan(X, [X|_]).
clan(X, [_|R]):- clan(X,R).
/* pomocni predikat desno proverava da li je kuca X desno od kuce Y u listi koja je data kao poslednji argument */
desno(X,Y,[Y,X|_]).
desno(X,Y,[_|R]):- desno(X,Y,R).
/* pomocni predikat pored proverava da li su kuce X i Y  pored u listi koja je data kao poslednji argument */
pored(X,Y,[X,Y|_]).
pored(X,Y,[Y,X|_]).
pored(X,Y,[_|R]):- pored(X,Y,R).
/*  predikat kuce resava zagonetku, tj. vraca listu kuca L koje zadovoljavaju uslove zadatka */
/*  anonimnom promenljivom cemo sugerisati da neki podatak postoji, ali na pocetku ta vrednost nije data */
kuce(L):- L =
 [ kuca(_,norvezanin ,_,_,_), /* norvezanin zivi u prvoj kuci sa leva */
   kuca(plava ,_,_,_,_),      /* plava kuca je pored kuce u kojoj zivi norvezanin */
   kuca(_,_,_,mleko ,_),     /* mleko se pije u srednjoj kuci */
   kuca(_,_,_,_,_),
   kuca(_,_,_,_,_) ],
   /* dodajemo kakve sve kuce treba da budu u listi */
   clan(kuca(crvena,englez ,_,_,_),L), /* dato je da Englez zivi u crvenoj kuci */
   clan(kuca(_,spanac,_,_,pas),L),
   clan(kuca(zelena,_,_,kafa,_),L),
   clan(kuca(_,ukrajinac,_,caj,_),L),
   /* sada zadajemo kakav odnos kuce treba da zadovolje koristeci pomocne predikate pored i desno */
   desno(kuca(zelena,_,_,kafa,_),kuca(bela,_,_,_,_),L),
   clan(kuca(_,_,spagete,_,puz),L),
   clan(kuca(zuta,_,pica,_,_),L),
   pored(kuca(_,_,piletina,_,_),kuca(_,_,_,_,lisica),L),
   pored(kuca(_,_,pica,_,_),kuca(_,_,_,_,konj),L),
   clan(kuca(_,_,brokoli,narandza ,_),L),
   clan(kuca(_,japanac,susi ,_,_),L),
```

```
32      /* kako bismo izgradili celokupno resenje, tj da bi do
           unifikacije svih anonimnih promenljivih, potrebno je
           iskoristiti i dva podatka koja se kriju u postavljenom
           pitanju */
33      clan(kuca(_,_,_,_,zebra),L),
34      clan(kuca(_,_,_,voda,_),L).
35  /* predikat koji na osnovu resenja koje dobijamo pozivom predikata
        kuce, daje informaiju ko drzi zebru, a ko pije vodu */
36  odgovori(X,Y) :- kuce(L), clan(kuca(_,X,_,_,zebra),L), clan(kuca(_
      ,Y,_,voda,_),L).
```

```
| ?- odgovori(X,Y).
X = japanac
Y = norvezanin ?;
no
```

Problem misionara i ljudoždera

Formulacija problema

Tri misionara i tri ljudoždera (engl. missionaries and cannibals) koji se nalaze na levoj obali reke je potrebno prevesti na desnu obalu čamcem koji prima dve osobe. Pri tome, ne sme u jednom trenutku na obali biti više ljudoždera od misionara, jer će u suprotnom ljudožderi pojesti misionare. Potrebno je naći raspored prevoženja, tako da svi bezbedno pređu reku.

Analiza rešenja

U nastavku su definisana pravila i činjenice koja omogućavaju bezbedno prelaženje reke.

- Potrebno je opisati činjenice koje će govoriti o broju misionara i ljudoždera na obalama i položaju čamca. Dovoljno je zadati broj misionara i ljudoždera na levoj strani, jer je time jednoznačno određen broj misionara i ljudoždera i na desnoj strani:

  `stanje(br_misionara, br_ljudozdera, levo).`

  Početno stanje je `stanje(3,3,levo)`, a završno `stanje(0,0, desno)`.

- Potrebno je zadati legalne prelaze imajući u vidu da se čamcem mogu prevesti najviše dve osobe. Primer legalnog prelaza je `legalan_prelaz(2, 0)`, sa značenjem da se u čamcu prevoze dva misionara i nijedan ljudožder.

- Pored legalnih prelaza, potrebno je zadati i legalna stanja imajući u vidu da broj ljudoždera ni na jednoj obali, ni u jednom trenutku, ne sme biti veći od broja

misionara. Prilikom zadavanja legalnih stanja položaj čamca je irelevantan i zbog toga taj podatak izostavljamo. Primer legalnog stanja je `legalno_stanje(X, X)`, sa značenjem da se na obali nalazi jednak broj misonara i ljudoždera.

- Predikat `novo_stanje` na osnovu trenutnog stanja formira legalno sledeće stanje:

  `novo_stanje(stanje(M1, Lj1, levo), stanje(M2, Lj2, desno)).`

- Na osnovu trenutnog i sledećeg stanja, predikat `izbor_poteza` nalazi odgovarjući potez, takav da ljudožderi ne mogu pojesti misionare:

  `izbor_poteza(stanje(M1, Lj1, levo), stanje(M2, Lj2, desno), potez(M, Lj, desno)).`

  Primer jednog poteza je `potez(1,1,desno)`, sa značenjem da se jedan misionar i jedan ljudožder prevoze čamcem na desnu stranu obale.
  Lista ovako izabranih poteza će upravo biti rešenje zadatka.

- Ključni predikat, `rasporedi`, generiše prethodno spomenutu listu poteza koja omogućava bezbedan raspored prevoženja. Ovaj predikat kao argumente prima trenutno stanje, listu do tada primenjenih stanja i kao treći argument, listu koja predstavlja rešenje:

  `rasporedi(TrenutnoStanje, Do_sada, [Potez | PreostaliPotezi]).`

Primer 6.20: Problem misionara i ljudoždera

```
1  /*legalni prelazi*/
2  legalan_prelaz(2, 0).
3  legalan_prelaz(1, 0).
4  legalan_prelaz(1, 1).
5  legalan_prelaz(0, 1).
6  legalan_prelaz(0, 2).
7
8  /*legalna stanja*/
9  legalno_stanje(X, X).
10 legalno_stanje(3, X).
11 legalno_stanje(0, X).
12
13 rasporedi(stanje(0, 0, desno), _, []).
14 rasporedi(TrenutnoStanje, Do_sada, [Potez | PreostaliPotezi]) :-
15         novo_stanje(TrenutnoStanje, SledeceStanje),
16         not(clan(SledeceStanje, Do_sada)),
```

```
17          izbor_poteza(TrenutnoStanje, SledeceStanje, Potez),
18          rasporedi(SledeceStanje, [SledeceStanje | Do_sada],
               PreostaliPotezi).
19
20
21  izbor_poteza(stanje(M1, Lj1, levo), stanje(M2, Lj2, desno), potez(
       M, Lj, desno)) :-
22          M is M1 - M2,
23          Lj is Lj1 -Lj2.
24  izbor_poteza(stanje(M1, Lj1, desno), stanje(M2, Lj2, levo), potez(
       M, Lj, levo)) :-
25          M is M2 - M1,
26          Lj is Lj2 - Lj1.
27
28  novo_stanje(stanje(M1, Lj1, levo), stanje(M2, Lj2, desno)) :-
29          legalan_prelaz(M, Lj),
30          M =< M1,
31          Lj =< Lj1,
32          M2 is M1 - M,
33          Lj2 is Lj1 - Lj,
34          legalno_stanje(M2, Lj2).
35  novo_stanje(stanje(M1, Lj1, desno), stanje(M2, Lj2, levo)) :-
36          legalan_prelaz(M, Lj),
37          M2 is M1 + M,
38          Lj2 is Lj1 + Lj,
39          M2 =< 3,
40          Lj2 =< 3,
41          legalno_stanje(M2, Lj2).
42
43
44  ispisredosleda([G|R]):-write(G), nl, ispisredosleda(R).
45  ispisredosleda([]).
46
47  clan(X,[X|_]).
48  clan(X,[_|R]):-clan(X,R).
49
50  misionari:-rasporedi(stanje(3,3,levo),[],X), ispisredosleda(X).
```

Nakon prevođenja, u okviru interpretera postavljamo upit `?- misionari` i kao rezultat dobijamo listu poteza koji omogućavaju bezbedno prevoženje.

```
|?- misionari.
```

```
potez(1,1,desno)
potez(1,0,levo)
potez(0,2,desno)
potez(0,1,levo)
potez(2,0,desno)
potez(1,1,levo)
potez(2,0,desno)
potez(0,1,levo)
potez(0,2,desno)
potez(1,0,levo)
potez(1,1,desno)
```

Glava 7

Haskell

7.1 Uvod u Haskell

Haskell je funkcionalni programski jezik nastao 1990. godine. Njegovim tvorcem se smatra Haskell Brooks Curry. Neke od osnovnih karakteristika programskog jezika Haskell su:

- transparentnost referenci - funkcija uvek daje isti rezultat za iste ulazne vrednosti i zato za Haskell kažemo da je *čist* funkcionalni jezik;
- lenja evaluacija - izbegavaju se nepotrebna izračunavanja;
- poseduje bogat sistem tipova;
- podržan je parametarski polimorfizam i preopterećivanje (engl. overloading), što za posledicu ima sažeto i generičko progamiranje.

Haskell koristi sve koncepte koje funkcionalna paradigma pruža i kao takav se dosta koristi u edukativne svrhe.

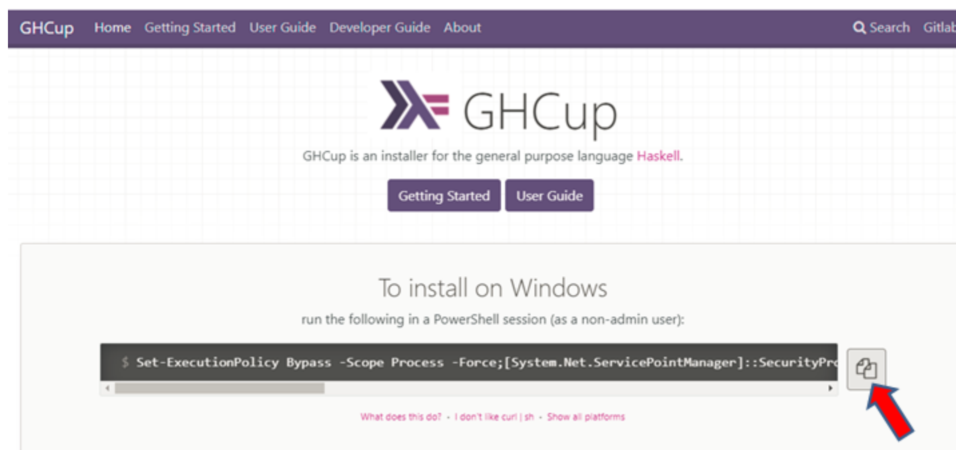
Instalacija

Postoje razne implementacije Haskell-a, a najpoznatija je implementacija GHC (engl. Glasgow Haskell Compiler), koja ujedno predstavlja i kompajler i interpreter. Za instalaciju GHC-a na operativnim sistemima Linux i Mac potrebno je u okviru terminala uneti sledeći niz komandi:

```
> sudo apt install ghc
```

Za Windows operativni sistem je potrebno sa *oficijalnog sajta pogramskog jezika Haskell*[1] prekopirati komandu kao na slici 7.1.

[1] www.haskell.org/ghcup/



Slika 7.1: Oficijalni sajt GHC-a

Zatim je datu komandu potrebno izvršiti u okviru Windows Power Shell-a (slika 7.2). Na svako od postavljenih pitanja se može odgovoriti klikom na taster enter. Instalacija može potrajati nekoliko minuta.

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\ThinkPad> Set-ExecutionPolicy Bypass -Scope Process -Force;[System.Net.ServicePointManager]::SecurityProtoco
l = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072;Invoke-Command -ScriptBlock ([ScriptBlock]::Create((Inv
oke-WebRequest https://www.haskell.org/ghcup/sh/bootstrap-haskell.ps1 -UseBasicParsing))) -ArgumentList $true
Picked C:\ as default Install prefix!
Where to install to (this should be a short Path, preferably a Drive like 'C:\')
Press enter to accept the default [C:\]:

Setting env variable GHCUP_INSTALL_BASE_PREFIX to 'C:\'
Preparing for GHCup installation...
Specify Cabal directory (this is where haskell packages end up)
Press enter to accept the default [C:\\cabal]:


Install HLS
Do you want to install the haskell-language-server (HLS) for development purposes as well?
[Y] Yes  [N] No  [A] Abort  [?] Help (default is "N"):
```

Slika 7.2: Instalacija GHC-a

Takođe, potreban je i tekst editor za pisanje izvornog koda. Pogodan tekst editor je Visual Studio Code jer podržava ekstenziju za označavanje sintakse. Fajlovi sa izvornim kodom u programskom jeziku Haskell se čuvaju sa ekstenzijom `.hs`.

Interpreter *ghci*

Nakon uspešno završene instalacije na raspolaganju imamo i kompajler i interpreter. Za potrebe ovog kursa koristićemo interpreter.

Pokrećemo interpreter kao na slici 7.3, nakon čega se prikazuje verzija interpretera i poruka da je učitana standarda biblioteka `Preclude.hs`. Pored toga, `Preclude>` označava početnu liniju u okviru koje se ispisuju komande.

Windows PowerShell

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\ThinkPad> ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude>
```

Slika 7.3: Pokretanje interpretera

Sa interpreterom možemo interaktivno komunicirati kao što je prikazano u nastavku. Pritiskom na enter vrši se evaluacija unetog izraza [14].

```
Prelude> 3+15
18
Prelude> 49 * 10
490
Prelude> 2342 - 1267
1075
Prelude> 7 / 2
3.5
Prelude>
```

Pored nevedenih aritmetičkih operatora, mogu se primenjivati i logički operatori: `&&` (logičko i), `||` (logičko ili) i `not` (negacija logičke vrednosti).

```
Prelude> True && False
False
Prelude> False || True
True
Prelude> not False
True
Prelude>
```

Takođe, vrednosti je moguće porediti i na jednakost, odnosno različitost.

```
Prelude> 5 == 5
True
Prelude> 5 /= 5
False
Prelude> "zdravo" == "zdravo"
True
Prelude>
```

Sve naredbe u interpreteru počinju sa operatorom dve tačke, tj. :. U tabeli 7.1 su prikazane neke od osnovnih komandi interpretera.

Tabela 7.1: Osnovne komande interpretera

Komanda	Značenje
`:load` *naziv*	učitava .hs fajl
`:reload`	osvežava već učitani fajl
`:type` *izraz*	pokazuje tip podatka proizvoljnog izraza
`:?`	prikazuje sve komande
`:quit`	za napuštanje interpretera

U okviru interpretera *ghci* moguće je imenovati izraz koristeći ključnu reč `let` kao što je prikazano u nastavku.

```
Prelude> let a = 5 * 3
Prelude> a
15
```

Prvi program

Pisanje programa u Haskell-u sastoji se pre svega od unošenja izvornog koda u okviru tekst editora. U primeru 7.1 definisana je funkcija `ispis_poruke`, koja ispisuje na izlaz odgovarajuću poruku.

Primer 7.1: Zdravo svete! :)

```
1 ispis_poruke = print "Zdravo svete! :)"
```

Zatim je potrebno prethodno uneti program sačuvati u fajlu sa ekstenzijom `.hs` i pomoću komande `load` učitati u okviru interpretera kao na slici 7.4.

```
Prelude> :load 1.hs
[1 of 1] Compiling Main             ( 1.hs, interpreted )
Ok, one module loaded.
*Main>
```

Slika 7.4: Učitavanje programa

Napomena: Program je učitan navođenjem komande `load` i samog naziva fajla koji je sačuvan u tekućem direktorijomu. Ukoliko se program učitava iz direktorijuma koji nije tekući, potrebno je navesti putanju do fajla.

Početna linija `Preclude>` nakon učitavanja programa postaje `*Main>` u okviru koje se učitani program izvršava pozivanjem odgovarajuće funkcije kao na slici 7.5.

```
*Main> ispis_poruke
"Zdravo svete! :)"
*Main>
```

Slika 7.5: Izvršavanje programa pozivanjem fukncije `ispis_poruke`

7.2 Funkcije u Haskell-u

Pojam funkcije u programskom jeziku Haskell se ne razlikuje od matematičkog pojma funkcije. Međutim, postoji razlika u samom zapisu funkcije. Na primer, u matematici izraz

$$f(a, b) + c \cdot d$$

znači primeniti funkciju na argumente a i b i rezultatu dodati proizvod vrednosti c i d. Prethodni izraz se u Haskell-u zapisuje na sledeći način:

$$f \;\; a \;\; b + c * d.$$

U narednoj tabeli 7.2 navedeni su još neki primeri matematičkih funkcija i njima ekvivalentni zapisi u Haskell-u.

Tabela 7.2: Zapisi funkcija u matematici i Haskell-u

U matematici	U Haskell-u
$f(x, y)$	$f \;\; x \;\; y$
$f(g(x))$	$f(g \;\; x)$
$f(x, g(y))$	$f \;\; x \;\; (g \;\; y)$
$f(x)g(y)$	$f \;\; x * g \;\; y$

Važno je napomenuti da u okviru izraza funkcijski simbol ima najveći prioritet. I zato je nekada neophodno koristiti zagrade. Na primer, ukoliko bismo u izazu $f(g(x))$ izostavili zagrade, Haskell bi izraz $f \;\; g \;\; x$ tumačio kao funkciju dva argumenta [13].

Za funkcije u Haskell-u kažemo da su **građani prvog reda** (engl. first class citizen), što u kontekstu programskog jezika ima sledeća značenja: funkcije se mogu čuvati u promenljivima, prosleđivati drugim funkcijama, kreirati u okviru funkcija i mogu biti povratna vrednost funkcije. **U Haskell-u je sve funkcija**. Na primer, operator množenja $*$ se u Haskell-u smatra funkcijom koja prima dva argumenta (parametra) i kao rezultat vraća njihov proizvod. Operatori se najčešće zapisuju u infiksnoj notaciji (između argumenata). Moguće je zapisati operatore i u prefiksnoj notaciji tako što se dati operator ogradi zagradama i nakon toga se navedu argumenti. Tako zapisani operatori u prefiksnoj notaciji nazivaju se **sekcije**.

```
Prelude> 15 * 25
```

```
375
Prelude> (*) 15 25
375
Prelude>
```

Funkcije su obično definisane prefiksno, a ukoliko se radi lakšeg razumevanja koristi inifiksa notacija, potrebno je naziv funkcije ograditi pomoću karaktera ‘ kao što je prikazano u nastavku.

```
Prelude> min 9 4
4
Prelude> 9 ‘min‘ 4
10
Prelude> div 5 2
2
Prelude> 5 ‘div‘ 2
2
Prelude> mod 5 2
1
Prelude> 5 ‘mod‘ 2
1
Prelude>
```

7.3 Sistem tipova

Haskell je statički i strogo tipiziran programski jezik. To znači da se tip svakog izraza određuje pre izračunavanja funkcije i svi tipovi se moraju poklapati, nema implicitnih konverzija. Statička provera tipova omogućava da većina grešaka bude otkrivena pri samoj kompilaciji, što znatno doprinosi procesu debagovanja. U okviru ovog poglavlja govorićemo o tipovima i klasama podataka kao osnovnim konceptima Haskell-a. Pre toga, uvešćemo liste i n-torke kao osnovne strukture podataka u Haskell-u.

Liste i n-torke

Liste u funkcionalnom programiranju nalaze široku primenu i predstavljaju ključan alat za transformaciju podataka. Elementi se zapisuju u okviru uglastih zagada i razdvajaju se zarezima.

Primer 7.2: Primeri listi u Haskell-u

```
1 lista1 = [1,4,7,9]
2 lista2 = [True,False,True]
```

U tabeli 7.3 su prikazani osnovni operatori za rad sa listama.

Tabela 7.3: Operatori nad listama

Operator	Značenje
`:`	Dodaje pojedinačne elemente na početak liste
`!!`	Izvlači element na zadatom indeksu
`++`	Operator konkatenacije dve liste

U nastavku su u okviru interpretera ilustrovane prethodno navedene operacije nad listama.

```
Prelude> 1 : [2,3]
[1,2,3]
Prelude> [4,5,1,8,9] !! 2
1
Prelude> [1,2,3,4] ++ [5,6]
[1,2,3,4,5,6]
```

Napomena: Liste su u Haskell-u indeksirane počev od 0.

Niske predstavljaju listu karaktera. U primeru su prikazani ekvivalentni zapisi niske „paradigma”.

```
Prelude> let niska1 = "Paradigma"
Prelude> let niska2 = ['P','a','r','a','d','i','g','m','a']
Prelude> niska1
"Paradigma"
Prelude> niska2
"Paradigma"
```

Neke od ugrađenih funkcija za rad sa listama prikazane su u tabeli 7.4

Tabela 7.4: Ugrađene funkcije za rad sa listama

Funckija	Povratna vrednost
`head`	Vraća prvi element liste
`tail`	Vraća rep liste
`length`	Vraća broj elemenata liste
`null`	Vraća `True` ukoliko je lista prazna, `False` inače

U nastavku je u okviru interpretera ilustrovana upotreba navedenih funkcija.

```
Prelude> head [5,4,3,2,1]
5
```

```
Prelude> tail [5,4,3,2,1]
[4,3,2,1]
Prelude> length [5,4,3,2,1]
5
Prelude> null []
True
Prelude> null [5,4,3,2,1]
False
```

U Haskell-u se pored listi za manipulaciju podacima koriste i n-*torke*, koje za razliku od listi mogu da sadrže elemente različitih tipova. Jedna n-torka ima unapred zadat broj elemenata. Elementi se zapisuju u okviru oblih zagada i razdvajaju se zarezima.

Primer 7.3: Primeri n-torki u Haskell-u

```
1 ntorka1 = ('p',1,[1.2, 2.3],"paradigma")
2 ntorka2 = (1,2,3,4)
```

Ugrađene funkcije definisane za parove (n=2) su:

- funkcija `fst` - vraća prvi element;
- funkcija `snd` - vraća drugi element.

U nastavku je u okviru interpretera ilustrovana upotreba navedenih funkcija.

```
Prelude> let t = (2.4,2)
Prelude> fst t
2.4
Prelude> snd t
2
```

Liste i n-torke mogu biti parametri i povratne vrednosti funkcija.

Osnovni tipovi podataka u Haskell-u

U okviru tabele 7.5 navedeni su osnovni tipovi podataka.

Koristeći komandu `type` u okviru interpretera možemo proveriti tipove navedenih vrednosti:

```
Prelude> :type True
True :: Bool
Prelude> :type 'a'
'a' :: Char
Prelude> :type "abc"
"abc" :: [Char]
Prelude>
```

Tabela 7.5: Osnovni tipovi podataka

Tip	Značenje	Primer
Bool	Logički tip	True, False
Char	Karakter	'a', 'c'
String	Niz karaktera	"abc"
Int	Celi brojevi ograničenog raspona	235
Integer	Celi brojevi proizvoljnog raspona	2425689052
Float	Realni brojevi	1.4142135

Liste i n-torke takođe postaju tipovi, ukoliko su poznati tipovi vrednosti koje liste i n-torke sadrže. Primeri tipova listi i n-torki su: `[Bool]`, `[Char]`,`(String,Bool,Char)` ...

Tipovi funkcija

Funkcija argumente jednog tipa preslikava u argumente drugog tipa. Na primer, u nastavku su data dva tipa funcije:

```
Bool -> Bool
Char -> Bool
```

Prvi primer predstavlja tip funkcije koji uzima i vraća logičku vrednost. Dok u okviru drugog primera je zadat tip funkcije koji uzima argument tipa karakter, a povratna vrednost je logičkog tipa. Iako je Haskell strogo tipiziran programski jezik, nije potrebno navesti tip funkcije pre same definicije jer je kompajler u stanju da automatski sam zaključi tipove. Međutim, da bi kôd bio jasniji, tip funkcije ćemo navoditi pre same definicije.

U nastavku su prikazani pozivi funkcije `head` u okviru interpretera.

```
Prelude> head[True,False,True]
True
Prelude> head["Yes", "No"]
"Yes"
Prelude> head[1.2, 3.3, 5.4, 8.1]
1.2
Prelude>
```

U okviru interpretera proveravamo tip funkcije `head`.

```
Prelude> :type head
head :: [a] -> a
```

Tip `head :: [a] -> a` znači da funkcija `head` uzima argumente tipa `a` i vraća vrednost istog tipa, gde `a` predstavlja tipsku promenljivu koja se odnosi na proizvoljan tip. Ovakve funkcije, čiji tipovi argumenata mogu biti različiti i ne moraju unapred biti poznati, nazivamo **polimorfnim funkcijama**.

Klase tipova

Svaka klasa tipova definiše skup funkcija koje moraju biti implementirane za tip koji je predstavnik date klase. U nastavku su date neke od osnovnih klasa tipova u Haskell-u.

Klasa `Eq`

Tipovi podataka koji se mogu porediti na jednakost i različitost pripadaju klasi `Eq`. Klasa `Eq` definiše sledeće dve funkcije: `(==)` i `(/=)`.

Svi osnovni tipovi podataka su instance klase `Eq`.

U nastavku je u okviru interpretera ilustrovana upotreba prethodno navedenih funkcija.

```
Prelude> True == False
False
Prelude> 3 == 3
True
Prelude> 3 /= 3
False
Prelude> 'a'=='b'
False
Prelude> "abc" == "abc"
True
Prelude> [1,2] == [1,2,2]
False
Prelude> (1,True) == (1, True)
True
```

Na osnovu prethodnog primera zaključujemo da je funkcija `==` definisana za različite tipove podataka: `Int`, `Bool` i `String`. U okviru interpretera proveravamo tip funkcije `==`.

```
Prelude> :type ==
(==) :: Eq a => a -> a -> Bool
```

Tip `Eq a => a -> a -> Bool` znači da funkcija `==` uzima dva argumenta proizvoljnog tipa `a` koja se mogu porediti na jednakost i različitost, a vraća vrednost tipa `Bool`. Ovakve polimorfne funkcije čiji tip sadrži jednu ili više tipskih promenljivih koje pripadaju određenoj klasi tipova, nazivamo **preopterećenim funkcijama** (engl. overloaded).

Klasa Ord

Uređeni tipovi podataka pripadaju klasi Ord. Klasa Ord definiše sledeće funkcije: <, >, <=, >=. Svi osnovni tipovi podataka su instance klase Ord. U nastavku je u okviru interpretera ilustrovana upotreba prethodno navedenih funkcija.

```
Prelude> 3 >= 1
True
Prelude> "abc" < "def"
True
Prelude> ('a',2) < ('b', 1)
True
```

Klasa Show

Klasa Show sadrži one tipove čije se vrednosti mogu predstaviti stringovnom reprezentacijom koristeći funkciju show. Svi osnovni tipovi podataka su instance klase Show.

U nastavku je u okviru interpretera ilustrovana upotreba funkcije show.

```
Prelude> show 3.14
"3.14"
Prelude> show False
"False"
Prelude> show [1,2,3]
"[1,2,3]"
```

Klasa Num

Klasa Num sadrži sve numeričke tipove podataka. Neke od osnovnih funkcija definisanih u klasi Num su: +, -, *,abs, signum.

Tipovi Int, Integer i Float su instance klase Num.

U nastavku je u okviru interpretera ilustrovana upotreba navedenih funkcija klase Num.

```
Prelude> 1 + 5
6
Prelude> 1.1 + 5.5
6.6
Prelude> negate 5
-5
Prelude> abs(-1)
1
Prelude> signum(-1)
```

```
-1
Prelude>
```

Klasa `Integral`

Celobrojni tipovi podataka pripadaju klasi `Integral`. Funkcije definisane u klasi `Integral` su: `div` i `mod`.

Tipovi `Int` i `Integer` su instance klase `Integral`.

U nastavku je u okviru interpretera ilustrovana upotreba prethodno navedenih funkcija, koje ćemo zadati i u infiksnoj i u prefiksnoj notaciji.

```
Prelude> div 5 2
2
Prelude> 5 ‘div‘ 2
2
Prelude> mod 5 2
1
Prelude> 5 ‘mod‘ 2
1
```

Klasa `Fractional`

Razlomački tipovi podataka pripadaju klasi `Fractional`. Funkcije definisane u klasi `Fractional` su: `/` i `recip`.

Osnovni tipovi `Float` i `Double` su instance klase `Fractional`. U nastavku je u okviru interpretera ilustrovana upotreba prethodno navedenih funkcija.

```
Prelude> 9.0 / 2.0
4.5
Prelude> recip 2.0
0.5
Prelude>
```

7.4 Definisanje funkcija

Kao i prilikom definisanja funkcije u matematici, definicija funkcije u Haskell-u uključuje zadavanje domena, kodomena i preslikavanja [7]. Preslikavanje se zadaje izrazima. Haskell omogućava nekoliko različitih načina za definisanje funkcija koje iz datog niza mogućih rezultata biraju odgovarajući. Nakon uvodnog primera u okviru kog će biti prikazan način zadavanja funkcije jednostavnim izrazom, biće reči o korišćenju uslovnih izraza i ograđenih jednačina (engl. guarded equation) pri definisanju funkcija. Biće objašnjen i

moćan koncept podudaranja obrazaca (engl. pattern matching). Na samom kraju ćemo govoriti o lambda izrazima kao još jednom načinu definisanja funkcija u Haskell-u.

Uvodni primer

Definisati funkciju `duplo x` koja računa dvostruku vrednost celog broja `x`.

U okviru tekst editora definišemo funkciju `duplo` kao u primeru 7.4.

Primer 7.4: Definicija funkcije `duplo`

```
1 duplo x = x + x
```

Nakon samog naziva funkcije navode se parametri funkcije odvojeni razmacima. U ovom primeru funkcija `duplo` ima jedan parametar `x`. Sa desne strane znaka jednakosti je izraz koji definiše funkciju. Nakon što smo definisali funkciju, potrebno je sačuvati je u okviru fajla sa ekstenzijom `.hs` i nakon toga je učitati u interpreteru koristeći komandu `load` kao što je prikazano u nastavku.

```
GHCi, version 8.10.7: https:\\www.haskell.org/ghc/  :? for help
Prelude> :load duplo.hs
[1 of 1] Compiling Main             ( duplo.hs, interpreted )
Ok, one module loaded.
*Main>
```

Nakon učitavanja pozivamo funkciju navodeći njen naziv i odgovarajući parametar.

```
*Main> duplo 2
4
*Main> duplo 2.2
4.4
```

Primetimo da funkcija `duplo` ispravno računa i dvostruku vrednost realnog broja. Koristeći komandu `info` u okviru interpretera se može proveriti tip funkcije `duplo`.

```
*Main> :info duplo
duplo :: Num a => a -> a
```

Haskell na osnovu same definicije funkcije `duplo` zaključuje da funkcija preslikava neki numerički tip u taj isti tip. Pošto se zadatkom traži da funkcija računa dvostruku vrednost celog broja, navešćemo i potpis funkcije i time naglasiti da je u pitanju funkcija koja kao argument prima ceo broj, i da je povratna vrednost funkcije takođe ceo broj.

Primer 7.5: Definicija i potpis funkcije `duplo`

```
1 -- potpis funkcije
2 duplo :: Int -> Int
3 -- definicija funkcije
```

```
4 duplo x = x + x
```

Ukoliko sada pokušamo da pozovemo funkciju `duplo` sa realnim argumentom, interpreter će prijaviti grešku. Zbog čitljivosti ćemo uvek pre same definicije funkcije, navesti i potpis funkcije.

Napomena: Prilikom zadavanja naziva funkcije, jedino ograničenje je da naziv funkcije mora početi malim slovom.

Kao i u prethodnom primeru, često se prilikom pisanja izvornog koda navode komentari. U Haskell-u su podržani jednolinijski i vešelinijski komentari.

Primer 7.6: Sintaksa komentara

```
1 -- jednolinijski komentar
2 {-
3  ovo je
4  viselinijski komentar -}
```

Uslovni izrazi

Uslovni izrazi predstavljaju najjednostavniji način definisanja funkcije. Ilustrovaćemo upotrebu na primeru 7.7.

Primer 7.7: Maksimum dva cela broja

```
1 maks :: Int -> Int -> Int
2 maks x y =
3     if x > y
4         then x
5         else y
```

Za razliku od imperativnih jezika kod kojih je `else` grana opciona, u Haskell-u je neophodna. Takođe, Haskell je striktan što se tiče poravnanja, tj. treba voditi računa o poravnanju blokova naredbi. Primetimo da u situaciji kada u okviru funkcije imamo više uslova, kao u primeru funkcije `sgn`, kôd postaje nepregledan.

Primer 7.8: Funkcija signum

```
1 sgn::Float->Int
2 sgn x=
3     if x>0
4         then 1
5         else
6             if x==0
```

```
7                     then 0
8                     else -1
```

To je razlog zašto ćemo se u nastavku upoznati sa sintaksom ograđenih jednačina, kao još jednim načinom definisanja funkcija, koji znatno doprinosi preglednosti koda.

Ograđene jednačine

U Haskell-u su ograđene jednačine (engl. guarded equation) alternativa za uslovne izraze. U primeru 7.9 je navedena ilustracija upotrebe ograđenih jednačina prilikom definisanja funkcije `max`.

Primer 7.9: Maksimum dva cela broja definisan pomoću ograđenih jednačina

```
1 maks :: Int -> Int -> Int
2 maks x y
3     | x > y = x
4     | otherwise = y
```

Što se tiče sintakse ograđenih jednačina, nakon navođenja imena funkcije i njenih argumenata, sledi uspravna crta, a zatim uslov za koji postoji odgovarajuća definicija. Uslove navodimo jedan ispod drugog dokle god ne iscrpimo sve uslove karakteristične za definiciju date funkcije. Zbog preglednosti koda, sintaksa ograđenih jednačina se češće koristi.

Primer 7.10: Funkcija signum definisana pomoću ograđenih jednačina

```
1 sgn::Float->Int
2 sgn x
3     |x > 0 = 1
4     |x == 0 = 0
5     |otherwise = (-1)
```

Važno je napomenuti da se uvek prvo navode specifični uslovi, a na kraju najopštiji jer se provera uslova vrši redom počev od prvog. Za razliku od `else` grane u `if-then-else` izrazu, kod ograđenih jednačina uslov `otherwise` se može izostaviti.

Poklapanje obrazaca

Poklapanje obrazaca (engl. pattern matching) predstavlja jednostavan način definisanja funkcija u Haskell-u. Ideja je da pokušamo da poklopimo vrednost funkcije prema obrascu, i, po potrebi, vežemo promenljivu sa uspešniml poklapanjem [11]. Na primer, funkciju koja računa negaciju logičke vrednosti jednostavno možemo definisati koristeći poklapanje šablona kao u primeru 7.11.

Primer 7.11: Logička negacija

```
negacija :: Bool -> Bool
negacija True = False
negacija False = True
```

Ukoliko funkciju `negacija` pozovemo za parametar `True`, doći će do uspešnog poklapanja sa prvim obrascem i rezultat će biti `False`. A ako pozovemo funkciju sa parametrom `False`, doći će do poklapanja sa drugim obrascem. Na sličan način možemo definisati i logičku konjukciju.

Primer 7.12: Logička konjunkcija

```
konjunkcija :: Bool -> Bool -> Bool
konjunkcija True True = True
konjunkcija True False = False
konjunkcija False True = False
konjunkcija False False = False
```

Obrasci nam omogućavaju da definisemo funkciju za konkretne parametre. Takođe, možemo koristiti simbol `_` koji nazivamo džoker (engl. wildcard). Simbol `_` se može poklopiti sa bilo čime, što nam omogućava da dođemo do rezultata bez korišćenja vrednosti svih parametara kao u primeru 7.12.

Primer 7.13: Logička konjunkcija korišćenjem džokera

```
konjunkcija :: Bool -> Bool -> Bool
konjunkcija True True = True
konjunkcija _ _ = False
```

Međutim, postoji još jedan, efikacniji način za definisanje logičke konjukcije. Vodimo se time da ukoliko je prvi parametar `True`, koja god da je vrednost drugog parametra, rezultat će biti jednak drugom parametru. Ukoliko je prvi parametar `False`, koja god da je vrednost drugog parametra, rezultat će biti `False`.

Primer 7.14: Logička konjunkcija korišćenjem džokera i identifikatora

```
konjunkcija :: Bool -> Bool -> Bool
konjunkcija True x = x
konjunkcija False _ = False
```

Ovde je važno objasniti razliku između identifikatora `x` i džokera `_`. Ključna razlika je u tome što džoker ne čuva vrednost. U prvom slučaju je nephodno koristiti identifikator jer on figuriše u samoj definiciji funkcije. U drugom slučaju je dozvoljeno koristiti džokera jer je vrednost funkcije uvek jednaka `False`, tj. džoker se ne pojavljuje u samoj definiciji.

Važno je napomenuti i da se poklapanje obrazaca pokušava redom počev od prvog šablona koji je naveden. Zato je potrebno voditi računa da neki prethodno navedeni

obrazac nije nadskup kasnije navedenog obrasca, jer u toj situaciji interpreter prijavljuje grešku u vidu poruke da je neki obrazac nedostižan.

7.5 Rekurzivne funkcije

U okviru uvodnog dela rekli smo da se izvođenje programa u Haskell-u svodi na evaluaciju izraza i to bez stanja. Posebno, nepostojanje naredbe dodele i promenljivih koje menjaju stanje ima za posledicu da iterativne konstrukcije nisu moguće. Jedini način iteracije u Haskell-u je *rekurzija*.

Funkcija koja izračunava faktorijel nekog broja je u matematici definisana na sledeći način:

$$n! = \begin{cases} 1, & n = 0 \\ n * (n-1)!, & n \geq 0 \end{cases}$$

U nastavku je prikazan primer 7.15 definicije funkcije koja računa faktorijel prirodnog broja u Haskell-u.

Primer 7.15: Faktorijel broja

```
faktorijel n =
    if n == 0
        then 1
        else n*faktorijel(n-1)
```

Primetimo da na osnovu matematičke definicije funkcije jednostavno možemo definisati istu funkciju u programskom jeziku Haskell, kao u primeru 7.16.

Primer 7.16: Faktorijel broja primenom poklapanja obrazaca

```
faktorijel' 0 = 1
faktorijel' n = n*faktorijel' (n-1)
```

Rekurzivne funkcije se najčešće definišu poklapanjem obrazaca, kao što je prikazano u primeru 7.16.

7.6 Funkcije višeg reda

Funkcije višeg reda su funkcije koje kao parametar ili kao povratnu vrednost imaju funkciju. U okviru ovog poglavalja ćemo najpre definisati *Karijev postupak* koji koristi činjenicu da povratna vrednost funkcije može biti funkcija. Karijev postupak se koristi za prevođenje funkcija sa više argumenata u funkciju jednog argumenta. Zatim ćemo uvesti pojam *lambda izraza* kao anonime funkcije koja je često parametar funkcija višeg reda.

Na kraju poglavlja ćemo se upoznati sa tri najznačajnije funkcije višeg reda u Haskell-u: `map`, `filter` i `fold`.

Karijev postupak

U Haskell-u se prilikom evaluacije funkcija sa više parametara primenjuje Karijev postupak. Osnovna ideja Karijevog postupka jeste da se funkcija primeni na svoj prvi argument i time dobije nova funkcija koja se zatim primeni na sledeći argument i tako redom sve dok se ne primeni na sve zadate argumente. U nastavku ćemo ilustrovati Karijev postupak na jednostavnom primeru funkcije koja sabira dva cela broja.

Primer 7.17: Funkcija `saberi`

```
1 saberi :: Int -> (Int->Int)
2 saberi x y = x + y
```

Funkcija `saberi` se primenjuje najpre na parametar `x`, a kao rezultat se dobija interna funkcija `saberi x` kojom se računa zbir broja `x` i bilo koje druge celobrojne vrednosti. Zatim se tako dobijena interna funkcija primeni na parametar `y` i kao rezultat se dobija `x + y`.

Napomena: Po konvenciji u Haskell-u strelica `->` je desno asocijativna, što za posledicu ima da je zapis `Int -> Int -> Int` ekivalentan zapisu `Int -> (Int -> Int)`. Sa druge strane, primena funkcije je levo asocijativna, pa je zapis `saberi x y` ekvivalentan zapisu `(saberi x) y`.

Funkcije koje uzimaju parametre jedan po jedan nazivamo *Karijevim funkcijama*. Karijeve funkcije su korisne jer se njihovom parcijalnom primenom definišu nove funkcije kojima su neki od parametara početne funkcije fiksirani. Primer 7.18 ilustruje parcijalnu primenu funkcije.

Primer 7.18: Parcijalna primena funkcije

```
1 pomnozi :: Int -> Int -> Int -> Int
2 pomnozi x y z = x*y*z
3 {-- funkcija pomnozi' dobijena fiksiranjem  prvog parametra sa
    leve strane --}
4 pomnozi' = pomnozi 2
```

```
PS C:\Users\ThinkPad\Desktop\Haskell> ghci
GHCi, version 8.10.7: https: \\www.haskell.org/ghc/  :? for help
Prelude> :load 1.hs
[1 of 1] Compiling Main             ( 1.hs, interpreted )
Ok, one module loaded.
*Main> pomnozi 2 3 4
```

```
24
*Main> pomnozi' 3 4
24
```

Lambda izrazi

U Haskell-u se funkcija može zadati korišćenjem *lambda izraza*. Lambda izraz definiše preslikavanje funkcije i njene parametre, ne navodeći ime funkcije. Zato se lambda izraz naziva i *anonimnom funkcijom*. U Haskell-u se lambda izraz koji definiše funkciju za izračunavanje dvostruke vrednosti datog parametra zadaje na sledeći način:

`\x -> x + x`.

Oznaka za lambda izraz je `\`, a zatim se navode parametri funkcije razdvojeni razmacima. Nakon parametara funkcije zadaje se simbol `->`, za kojim sledi definicija funkcije.

Svaki lambda izraz je ograničen na samo jedan argument. Međutim, i funkcije više argumenata se mogu definisati pomoću lambda izraza ukoliko se na njih prethodno primeni Karijev potupak. Lambda izrazi su pogodni za definisanje funkcija koje se pozivaju samo jednom i obično se koriste kao parametri funkcija višeg reda.

Funkcije višeg reda map, filter i fold

Posebno važne funkcije višeg reda su ugrađene funkcije standardne biblioteke `map`, `filter` i `fold`. Navedene funkcije se koriste za manipulaciju kolekcijama podataka. U nastavku ilustrujemo upotrebu ovih funkcija u Haskell-u.

Funkcija `map`

Funkcija `map` se koristi prilikom rešavanja zadataka u okviru kojih je potrebno primeniti jednu istu operaciju nad svim elementima neke kolekcije podataka. Kao parametre funkcija `map` uzima funkciju i listu, dok je povratna vrednost lista dobijena transformacijom ulazne liste koristeći datu funkciju, kao što je prikazano u primeru 7.19.

Primer 7.19: Provera tipa funkcije *map* u okviru interpretera

```
Prelude> :type map
map :: (a -> b) -> [a] -> [b]
```

Na osnovu potpisa funkcije `map` zaključujemo da je prvi parametar funkcija koja prima vrednost tipa `a`, a vraća vrednost tipa `b`. Drugi parametar je lista elemenata istog tipa kao i ulazni parametar date funkcije, dok povratna vrednost predstavlja listu tipa `b`, koji odgovara tipu povratne vrednosti date funkcije.

Funkcija koja se zadaje kao parametar funkcije `map` se najčešće definiše koristeći lambda izraze ili sekcije, kao što je prikazano u primeru 7.20.

Primer 7.20: Ilustracija funkcije *map* u okviru interpretera

```
Prelude> map (\x -> x + 1) [1, 2, 3, 4]
[2,3,4,5]
Prelude>  map (+1) [1, 2, 3, 4]
[2,3,4,5]
```

Funkcija `filter`

Funkcija `filter` se koristi prilikom rešavanja problema u okviru kojih je potrebno izdvojiti sve elemente neke kolekcije podataka koji zadovoljavaju dati uslov. Kao parametre funkcija `filter` uzima funkciju uslova i listu, dok je povratna vrednost lista elemenata koji zadovoljavaju dati uslov. Provera tipa funkcije *filter* je prikazana u primeru 7.21.

Primer 7.21: Provera tipa funkcije *filter* u okviru interpretera

```
Prelude> :type filter
filter :: (a -> Bool) -> [a] -> [a]
```

Na osnovu potpisa funkcije `filter` vidimo da je prvi parametar funkcija koja prima vrednost tipa `a`, a vraća vrednost tipa `Bool`. Drugi parametar je lista elemenata takođe tipa `a`, dok je povratna vrednost lista onih elemenata ulazne liste za koje uslovna funkcija vraća `True`.

Kao i kod funkcije `map`, funkcija koja se zadaje kao parametar funkcije `filter` se definiše koristeći lambda izraze ili sekcije, kao što je prikazano u primeru 7.22.

Primer 7.22: Ilustracija funkcije *filter* u okviru interpretera

```
Prelude> filter (\x -> x > 1) [1, 2, 3, 4]
[2,3,4]
Prelude> filter (>1) [1, 2, 3, 4]
[2,3,4]
```

Funkcija `fold`

Funkcija `fold` je pogodna za rešavanje onih problema koji zahtevaju iteraciju kroz listu elemenata i njihovo kombinovanje u jednu vrednost. U zavisnosti od toga da li se iteracija vrši s leva na desno, ili s desna na levo, u Haskell-u su implementirane funkcije `foldl` (*engl. left fold*) i `foldr` (*engl. right fold*). Parametri funkcija `foldl` i `foldr` su: binarna funkcija, početna vrednost (akumulator) i kolekcija podataka, kao što je prikazano u primeru 7.23. Binarna funkcija se redom primenjuje na elemente kolekcije i akumulator, a krajnji rezultat te primene je povratna vrednost odgovarajuće `fold` funkcije.

Primer 7.23: Provera tipa funkcija *fold* u okviru interpretera

```
Prelude> :type foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
Prelude> :type foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

Iz potpisa funkcija `fold` vidimo da postoji oganičenje po kome tip `t` pripada klasi `Foldable`, u okviru koje su definisane funkcije `foldl` i `foldr`. Kao prvi parametar data je binarna funkcija. Drugi parametar je akumulator tipa `b`, treći kolekcija `t` koja prima elemente tipa `a` i na kraju povratna vrednost čiji je tip isti kao i tip akumulatora. Primetimo da je u potpisu funkcija `foldl` i `foldr` jedina razlika u rasporedu parametara binarnih funkcija. Drugim rečima, funkcija `foldl` prima akumulator kao prvi argument binarne funkcije,a funkcija `foldr` kao drugi argument. U nastavku je u okviru interpretera na primeru funkcije koja računa zbir svih elemenata liste ilustrovana upotreba funkcije `foldl` (7.24).

Primer 7.24: Ilustracija funkcije *foldl* u okviru interpretera

```
Prelude> foldl (\acc x -> acc + x) 0 [1,2,3,4]
10
```

Prvi parametar funkcije `foldl` je binarna funkcija `(\acc x -> acc + x)`, `0` je početna vrednost, a poslednji parametar je lista `[1,2,3,4]` nad kojom se primenjuje funkcije `foldl`. Prvo se `0` koristi kao prvi parametar binarne funkcije, a za `x` se uzima prvi element iz liste. Dobijeni međurezultat `(0 + 1)` postaje novi akumulator i binarna funkcija se primenjuje na akumulator i naredni element iz liste, nakon čega novi akumulator postaje `(1 + 2)`, a `3` nova trenutna vrednost. Primenom binarne funkcije vrednost akumulatora postaje `6` i finalno se kao trenutni element uzima poslednji element iz liste, sabira sa akumulatorom i kao kranji rezultat se dobija `10`. Navedeni opis izračunavanja možemo zapisati na sledeći način:

```
(((0 + 1)+ 2) + 3) + 4.
```

Slično za funkciju `foldr` (7.25).

Primer 7.25: Ilustracija funkcije *foldl* u okviru interpretera

```
Prelude> foldr (\x acc  -> x + acc) 0 [1,2,3,4]
10
```

Jedina je razlika u tome što je sada prvi parametar binarne funkcije poslednji element iz liste, dok je akumulator drugi parametar. Nakon primene binarne funkcije tako dobijeni međurezultat se opet uzima kao drugi parametar, dok je prvi parametar naredni element iz liste:

```
1 + (2 + (3 + (4 + 0))).
```

Rezultat je u oba slučaja `10` jer je sabiranje komutativna operacija. Kao parametar funkcija `foldl` i `foldr` možemo proslediti i funkciju zapisanu koristeći sekcije (7.26).

Primer 7.26: Prvi parametar funkcija *fold* je funkcija + definisana pomoću sekcija

```
Prelude> foldl (+) 0 [1,2,3,4]
10
Prelude> foldr (+) 0 [1,2,3,4]
10
```

Za razliku od sabiranja, kod oduzimanja nije svejedno da li ćemo primenjivati funkciju `foldl` ili `foldr` (7.27).

Primer 7.27: Prvi parametar funkcija *fold* je funkcija - definisana pomoću sekcija

```
Prelude> foldl (-) 0 [1, 2, 3]
-6
Prelude> foldr (-) 0 [1, 2, 3]
2
```

Objašnjenje izračunavanja u prvom slučaju:

```
((0 - 1) - 2) - 3.
```

Dok u drugom slučaju imamo:

```
1 - (2 - (3 - 0)).
```

7.7 Korisnički tipovi i klase tipova

U okviru poglavlja 7.3 upoznali smo se sa osnovnim klasama tipova u Haskell-u, kao i sa tipovima koji predstavljaju instance osnovnih klasa tipova. U ovom poglavlju biće opisana dva mehanizma za deklarisanje novih tipova podataka u programskom jeziku Haskell, kao i mehanizam za deklarisanje korisničkih klasa tipova.

Korisnički tipovi

U Haskell-u se novi tipovi podataka mogu deklarisati korišćenjem ključnih reči `type` i `data`.

Deklaracija `type`

Deklaracija tipa podataka pomoću ključne reči `type` podrazumeva uvođenje sinonima za već postojeći tip podataka. Time kôd postaje čitljiviji, odnosno pregledniji. Na primer, u okviru standardne biblioteke tip `String` je deklarisan kao u primeru 7.28.

Primer 7.28: Definicija tipa `String` u okviru standardne biblioteke

```
type String = [Char]
```

Na ovaj način je samo postavljeno novo ime - `String`, za niz karaktera, kao već postojeći tip podataka. Nakon ključne reči `type` uvodi se ime novog tipa koje mora početi velikim slovom, koji se zatim izjednačava sa već postojećim tipom podataka.

Tipovi mogu imati i parametre, kao u primeru 7.29.

Primer 7.29: Tip podataka definisan sa parametrom `a`

```
type Par a = (a,a)
```

`Par` predstavlja novo ime za par dva elemenenta proizvoljnog tipa. Parametar `a` se uvodi kao identifikator za neki proizvoljan tip ili tipsku klasu. U primeru 7.30 je definisana funkcija za množenje elemenata instance tipa `Par Int`.

Primer 7.30: Funkcija pomnoži definisana nad korisničkim tipom `Par`

```
type Par a = (a,a)
pomnozi :: Par Int -> Int
pomnozi (a,b) = a*b
```

Deklaracija `data`

U Haskell-u se potpuno novi tip podataka uvodi koristeći ključnu reč `data`. Takvi tipovi podataka drugačije se nazivaju i *algebarskim tipovima podataka*.

Primer 7.31: Definicija tipa `Bool` u okviru standardne biblioteke

```
data Bool = False | True
```

Prilikom definicije novog tipa podataka nakon ključne reči `data` navodi se ime tipa za kojim sledi proizvoljan broj parametara, koji se takođe mogu i izostaviti. Sa desne strane znaka jednakosti `=` navode se konstruktori rays razdvojeni znakom `|`. Konstruktori, koji opisuju način na koji se kreiraju instance datog tipa, moraju početi velikim slovom i mogu imati parametre. Kostrunktori `True` i `False` u navedenom primeru se pozivaju bez parametara i predstavljaju konstante.

Primer 7.32 ilustruje definiciju tipa sa parametrima.

Primer 7.32: Definicija tipa `Trougao`

```
data Trougao a b c = Jednakostranicni a
  | Jednakokraki a b
  | Raznostranicni a b c
```

Nakon navođenja imena tipa, sledi lista parametara, odnosno identifikatora `a, b` i `c` koji označavaju tipske promenljive. Tip `Trougao` je definisan konstruktorima `Jednakostranicni`, `Jednakokraki` i `Raznostranicni`, koji se zadaju sa odgovarajućim brojem parametara. Pomoću jednog od navedenih konstruktora možemo kreirati instancu tipa `Trougao`, međutim nije moguće ispisati kreiranu instancu, jer za tip `Trougao` nije podržana nijedna funkcionalnost. Prilikom poziva konstruktora `Jednakostranicni 10` interpreter prijavljuje grešku, jer tip `Trougao` ne pripada klasi `Show`.

```
Prelude> :load trougao.hs
[1 of 1] Compiling Main             ( trougao.hs, interpreted )
Ok, one module loaded.
*Main> Jednakostranicni 5

<interactive>:3:1: error:
    * No instance for (Show (Trougao Integer b0 c0))
        arising from a use of ‘print’
    * In a stmt of an interactive GHCi command: print it
*Main>
```

Ovaj problem se jednostavno može rešiti. Potrebno je naglasiti da novodeklarisani tip pripada odgovarajućoj klasi koristeći ključnu reč `deriving`. Time novodeklarisani tip nasleđuje sve podrazumevane implementacije funkcija koje su podržane u navedenoj klasi.

Primer 7.33: Ključna reč `deriving`

```haskell
data Trougao a b c  = Jednakostranicni a
            | Jednakokraki a b
            | Raznostranicni a b c
            deriving Show
```

```
Prelude> :load trougao.hs
[1 of 1] Compiling Main             ( trougao.hs, interpreted )
Ok, one module loaded.
*Main> Jednakostranicni 5
Jednakostranicni 5
*Main> Raznostranicni 5 4 3
Raznostranicni 5 4 3
```

U primeru 7.34 je prikazana implementacija funkcije koja računa obim instance tipa `Trougao`.

Primer 7.34: Funkcija `obim`

```haskell
obim :: Trougao Float Float Float -> Float
```

```
2 obim (Jednakostranicni a) = 3*a
3 obim (Jednakokraki a b) = 2*a + b
4 obim (Raznostranicni a b c) = a + b + c
```

Funkcija `obim` kao ulazni parametar dobija instancu tipa `Trougao`, čiji su parametri tipa `Float`. Time je naglašeno da će realne vrednosti biti prosleđene konstruktorima. Povratna vrednost funkcije `obim` je takođe realnog tipa. Funkcija se definiše poklapanjem obrazaca i time se za svaki od konstruktora opisuje način izračunavanja obima.

```
Prelude> :load trougao.hs
[1 of 1] Compiling Main             ( trougao.hs, interpreted )
Ok, one module loaded.
*Main> obim(Jednakostranicni 5)
15.0
*Main> obim(Raznostranicni 5 4 3)
12.0
```

Konstruktor korisnički definisanog tipa podataka kao parametar može imati instancu nekog drugog korisnički definisanog tipa. Svakom parametru konstruktora je moguće dodeliti ime. Tako imenovani parametri se mogu koristiti za dobijanje određenih podataka iz konkretne instance (slično `get` metodi u objektno-orijentisanom programiranju).

Primer 7.35: Korisnički definisan tip `Zivotinja`

```
1  {- definicija tipa podataka Zivotinja koji
2    moze biti Pas, Macka ili Papagaj -}
3  data Zivotinja = Pas
4                  | Macka
5                  | Papagaj
6                  deriving Show {- nad tipom Zivotinja
7                                   se instancira
8            klasa Show -}
9
10 {- tip Ljubimac se karakterise imenom,
11  godinom i tipom zivotinje -}
12 {- MkLj je konstruktor
13   ciji se parametri navode imenovano
14   u okviru viticastih zagrada -}
15 data Ljubimac = MkLj {
16     ime :: String, -- nazivi parametara su imenovani
17     godine :: Int, -- za svaki parametar se navodi tip
18     vrsta :: Zivotinja }
19     deriving Show
20
```

```
21 -- funkcija stariji poredi po godinama dva ljubimca
22 stariji :: Ljubimac -> Ljubimac -> Bool
23 stariji ljubimac1 ljubimac2 =
24   godine ljubimac1 > godine ljubimac2
```

```
Prelude> :load  zivotinja.hs
[1 of 1] Compiling Main             ( zivotinja.hs, interpreted )
Ok, one module loaded.
*Main> let dzeki = MkLj {ime = "Dzeki", godine = 3, vrsta = Pas}
*Main> let lui = MkLj {ime = "Lui", godine = 5, vrsta = Macka}
*Main> stariji dzeki lui
False
```

Korisničke klase tipova

Klasa u Haskell-u ima ulogu sličnu kao intefejs u objektno-orijentisanom programiranju. Klasa definiše funkcije koje tip treba da implementira da bi bio instanca te klase.

Primer 7.36: Definicija klase `Eq` u okviru standardne biblioteke

```
1 class Eq a where
2   (==) :: a -> a -> Bool
3   (/=) :: a -> a -> Bool
4   {-# MINIMAL (==) | (/=) #-}
```

Nakon ključne reči `class` navodi se ime klase, a zatim tipski parametar `a` koji će se prikom instanciranja klase zameniti nekim konkretnim tipom. Proizvoljan tip označen tipskom promenljivom `a` će biti instanca klase `Eq` ukoliko implementira neku od navedene dve funkcije `==` ili `/=`.

Primer 7.37 ilustruje kreiranje nove klase `Oblik` čije su instance dva korisnički definisana tipa `Krug` i `Pravougaonik`. Tipovi podataka `Krug` i `Pravougaonik` deklarišu se kao instance klasa `Eq` i `Show` tako što predefinišu operator `==`, odnosno funkciju `show` koristeći ključnu reč `where`. Drugi način je da se koristeći ključnu reč `deriving` naglasi da tipovi podataka `Krug` i `Pravougaonik` pripadaju klasama `Eq` i `Show` i time nasleđuju podrazumevane implementacije operatora `==` i funkcije `show`. Pomoću ključne reči `where` zadaju se osnovne funkcionalnosti klase `Oblik`. Takođe, prilikom instanciranja tipova podataka, nakon ključne reči `where` zadaje se implementacija potrebnih funkcionalnosti.

Primer 7.37: Klasa `Oblik`

```
1 data Krug =
2     MkK {r :: Float}
```

```
3  data Pravougaonik =
4      MkP {a:: Float, b :: Float}
5  {- a je tipski parametar koji se
6    prilikom instanciranja klase
7    postavlja na konkretan tip -}
8  class Oblik a where
9   {- povrsina i obim predstavljaju dve osnovne
10   funkcionalnosti klase Oblik-}
11     povrsina :: a -> Float
12     obim     :: a -> Float
13 {- zadajemo da je Krug instanca klase Oblik tako sto
14   tipski parametar a menjamo sa konkretnim
15   tipom Krug, a nakon toga se implemetiraju
16   funkcije obim i povrsina -}
17 instance Oblik Krug where
18 {-na osnovu potpisa funkcija u okviru klase Oblik,
19  zakljucujemo da je x zapravo Krug-}
20     povrsina   x = (r x) * (r x) * pi
21     obim       x = 2 * (r x) * pi
22 {- slicno za pravougaonik -}
23 instance Oblik Pravougaonik where
24     povrsina x = (a x) * (b x)
25     obim x = 2 * ((a x) + (b x))
26 {- da bi krugovi, mogli da se porede,
27    tip Krug mora biti instanca klase Eq,
28   tacnije potrebno je predefinisati operator == -}
29 instance Eq Krug where
30     (==) k1 k2 = (r k1) == (r k2)
31 {- slicno za pravougaonik -}
32 instance Eq Pravougaonik where
33     (==) p1 p2 = (a p1) == (a p2) &&  (b p1) == (b p2)
34 instance Show Krug where
35     show x = "(r=" ++ show (r x) ++ ")"
36
37 instance Show Pravougaonik where
38     show x = "(a=" ++ show (a x) ++ "," ++
39      "b=" ++ show (b x) ++ ")"
```

Klasa u Haskell-u je moćan koncept koji omogućava da korisnički definisan tip može da koristi neke već implementirane funkcionalnosti. Važno je naglasiti da klasa ne predstavlja tip podataka kao u objektno-orijentisanom programiranju. U funkcionalnom

programiranju instanca određene klase predstavlja tip podataka.

Glava 8

Zaključak

U okviru elektronskih lekcija o programskim paradigmama kreiranih za potrebe ovog rada, obrađeni su i sistematizovani osnovni koncepti funkcionalne i logičke paradigme, kao i matematičke teorije koja se nalazi u osnovi logičke paradigme programiranja.

Lekcije su pisane sa ciljem da budu pogodne za sve one koji se prvi put susreću sa funkcionalnom i logičkom paradigmom. Njihova prednost u odnosu na druge programske paradigme je način rešavanja problema, koji podrazumeva da programer treba precizno da opiše dati problem, a mehanizam programskog jezika pronalazi traženo rešenje. U okviru elektronskih lekcija logička paradigma je uvedena kroz programski jezik Prolog, koji je ujedno i najpopularniji jezik ove paradigme. Funkcionalna paradigma je prikazana kroz Haskell, čist funkcionalni jezik, koji koristi sve koncepte funkcionalne paradigme. Kao takav, Haskell predstavlja dobru osnovu za učenje drugih funkcionalnih jezika. Funkcionalna paradigma postaje sve popularnija, te sve veći broj programskih jezika koristi osnovne funkcionalne koncepte, a u okviru elektronskih lekcija je dat sistematičan pristup za njihovo izučavanje.

Literatura o funkcionalnoj i logičkoj paradigmi je obimna i lako dostupna na internetu. Jedan od glavnih motiva za pisanje ovog rada bio je da se omogući pogodna literatura na srpkom jeziku, s obzirom da su dostupni materijali najčešće na engleskom jeziku. Važno je napomenuti da sadržaj elektronskih lekcija u potpunosti prati plan nastavnog predmeta Programske paradigme, pa tako učenicima četvrtog razreda specijalizovanog IT odeljenja može poslužiti kao literatura za lakše savladavanje gradiva. Sve elektronske lekcije javno su dostupne na linku `http://edusoft.matf.bg.ac.rs/eskola_veba/#/course-details/pp`.

Literatura

[1] Jurić Nemanja, Marić Miroslav, *eŠkola Veba*, Matematički fakultet, Beograd, 2016

[2] A. Tucker, R. Noonan, *Programming Languages: Principles and Paradigms*, McGraw-Hill Science, 2001

[3] Predrag Janičić, Filip Marić, *Programiranje 2*, Matematički fakultet, Beograd, 2021

[4] Predrag Janičić, Mladen Nikolić, *Veštačka inteligencija*, Matematički fakultet, Beograd, 2021

[5] Milena Vujošević Janičić, *Odnos programskih jezika i programskih paradigmi*, http://www.programskijezici.matf.bg.ac.rs/ppR/2020-1/predavanja/01_odnosPJiPP_tekst.pdf

[6] Milena Vujošević Janičić, *Logičko programiranje*, http://www.programskijezici.matf.bg.ac.rs/ppR/2020-1/predavanja/logicko_programiranje.pdf

[7] Milena Vujošević Janičić, *Funkcionalno programiranje*, http://www.programskijezici.matf.bg.ac.rs/ppR/2020-1/predavanja/funkcionalno_programiranje_tekst.pdf

[8] Bird Richard, Philip Wadler, *Introduction to functional programming*, Prentice Hall International (UK), 1988, ISBN 0-13-484189-1

[9] P. Miličić, V. Stojanović, Z. Kadelburg, B. Boričić, *Matematika za prvi razred srednje škole*, Zavod za udžbenike, Novi Sad, 1991, ISBN 86-23-81095-3

[10] Predrag Janičić, *Matematička logika u računarstvu*, Beograd, 2008, ISBN 86-7589-040-0

[11] Miran Lipovača, *Learn You a Haskell for Great Good!*, 2011, ISBN 978-1-59327-283-8 www.learnyouahaskell.com

[12] Dušan Tošić, Radivoj Protić, *Prolog kroz primere*, 1991, ISBN 86-325-0306-5

[13] Graham Hutton, *Programming in Haskell*, Cambridge University Press, 2007, ISBN-13 978-0-511-29218-7

[14] Zvanična stranica jezika Haskell, `www.haskell.org`

[15] *B-Prolog user's manual*, `www.picat-lang.org/bprolog/download/manual.pdf`

[16] Ivan Bratko, *Prolog programming for artificial intelligence*, Addison-Wesley, 1990, ISBN 0-20r-14224-4

[17] P. Van Roy, S. Haridi, *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2003.