0 Uvod

Programiranje i algoritmi

Pojam programa, programiranja i algoritma nimalo nije jednostavno definisati. Po Vebsterovom rečniku, program se definiše kao kratke, uglavnom odštampane, smernice koje treba pratiti, a progamiranje kao pisanje takvog programa. Naravno, iako tačna, ovakva definicija nam nimalo ne pomaže da konkretno opišemo šta je to kompjuterski program. Ispostavlja se da odgovor na pitanje šta je programiranje i šta je algoritam ni ne može dati jednoznačno, pa se zbog toga uvodi čitava teorija algoritama, koja nam ne pruža definiciju ovih pojmova, već opisuje kako jednoznačno i bez prostora za sumnjom, možemo da ih identifikujemo. Pojam algoritma[1] ili efektivne procedure dugo je bio prisutan bez ikakvog nastojanja za njegovim definisanjem. Neformalno, pod algoritmom se podrazumeva mehanicki izvodljiv postupak, definisan *konačnim* brojem instrukcija, koji se izvodi korak po korak nad *konačnim* skupom polaznih podataka, pričemu je svaki korak nedvosmisleno definisan i završava se u *konačnim* vremenskim i prostornim okvirima.

O algoritmima se pričalo odavno, još od doba starih Vavilonaca. Jedan od najstarijih algoritama iz tog perioda je algoritam svodjenja razlomka na uzajamno proste činioce, kojeg svi učimo u 3. razredu osnovne škole. Kulturu odredivanja algoritma su dalje unapredili stari grci, medju kojima je i popularan algoritam odredjivanja prostih brojeva, ali pravi procvat algoritama će tek nastati u XX veku.
Na II svetksom kongersu matematičara koji je održan 1900. godine u Parizu, Hilbert[2] je izneo niz problema (23, tačno) za koje nije

imao rešenja i smatrao ih vrlo interesantnim, ako ne i bitnim. Jedan od tih problema je tražio da se precizno opiše postupak koji daje da/ne odgovor na to da li jedna specifična jednačina ima celobrojnih rešenja. Do tada, o sličnim postupcima se pričalo samo na intuitivnom nivou, te mnogi nisu razumeli šta je Hilbert imao na umu. Srećom, upravo u tom periodu, matematika je prolazila kroz, tako reći, fazu samoistraživanja i pitanja o preciznom definisanju same matematike je bila aktuelna tema. Nakon toga je usledio dug niz godina gde idalje nikakav formalni koncept ovih termina nije usvojen kao ni bilo kakav odgovor na Hilbertov problem. 1928. godine, Hilbert, zajedno sa još par kolega, postavlja sličan problem pod nazivom Entscheidungsproblem, što u prevodu znači problem odlučivosti. Ovaj problem je ponovo tražio da se konstruiše algoritam koji daje da/ne odgovor na jedno takode strogo formalno pitanje. Sve ovo je upalo u oči dvojici brilijantnih matematičara, koji su uporedo, potpuno nezavistno jedan od drugog, došli do rešenja tek 1936. godine. Doktor matematike i profesor na Univerzitetu Prinston, Alonzo Čerč[3] dolazi do odgovora pomoću formalnog računa kojeg je sam razvio i nazvao lambda račun (λ račun), a mladi student sa Univerziteta Kembridžu, Alan Tjuring[4] pomoću mnogo 'prizemnijeg' načina, konstrukcijom jedne idejne mašine. Zadivljen ovim poduhvatom, Čerč poziva Tjuringa da mu se pridruži na Univerzitetu Prinston i naziva ovu mašinu Tjuringovom Mašinom (TM). Formalna definicija TM se uvodi preko uredjene sedmorke i dosta je tehnički zahtevna, stoga ćemo ovde pričati samo o glavnoj iterpretaciji nje.

TM se može zamisliti kao beskonačno duga traka koja je izdeljena na polja (ćelije). U svaku ćeliju trake moˇze biti upisan samo jedan simbol nekog unapred izabranog alfabeta (pisma, npr. pisma sastavljenog samo od nula i jedinica) i jednog specijalnog simbola,

takozvanog blanko simbola ”␣”. Pored toga, zamišljamo i da postoji mehanizam, koga nazivamo glava (head), koji može da čita sdržaj samo jedne ćelije i da izvršava samo jednu od instrukcija Tjuringovog programa. Ove instrukcije govore glavi šta da upiše u datu ćeliju i, u zavnisnosti šta se u toj ćeliji nalazilo pre upisivanja, govori glavi u koju stranu trake da se pomeri za jedno mesto (levo, desno ili da se ne pomera). Ovako definisana mašina može da rešava čitav niz problema (one koje je postavio Hilbert, a i više) i omogućava nam da definišemo algoritam kao postupak rešavanja zadatog problema putem Tjurignove Mašine, program kao niz Tjuringovih instrukcija, a programiranje kao pisanje tog programa. Zbog ovog revolucionarnog otkrića, mnogi smatraju Alana Tjuringa ocem programiranja.

Reprezentacija brojeva i brojevni sistemi

Broj je jedan od prvih matematičkih koncepata koji je čovek osmislio tj. otkrio i služio je da se enumerišu razni objekti i pojmovi. Njihovo zančenje varira od konteksta u kome se koriste. Na primer, '5' može da označava broj ovaca u stadu, ili koliko puta je zemlja opisala potpunu elipsu oko sunca od nečijeg rodjenja, ili pak da obeleži peron na stanici. Svaki od ovih primera validno koristi broj '5', ali taj broj reprezentuje različite stvari, pa je stoga bitno primetiti da jedan isti broj moze da označava različite stvari u zavisnosti od njegove reprezentacije.
Pored toga, jedan isti pojam, odnosno ideja kao što je broj pet može da se žapiše na ražličite načine, korišćenjem različitih simbola ili skupa simbola. Tako na primer, 'pet' je išto što i '5' i isto što i 'five', 'V', '五' ...

Ovo vezivanje simbola za brojeve potiče od ljudske potrebe da brojeve koriste lakše i nedvosmisleno u komunikaciji. Stari Rimljani, recimo, su izdvojili brojeve jedan, pet, deset, pedeset, sto, petsto i hiljadu kao najvažnije i dodelili im simbole 'I', 'V', 'X', 'L', 'C', 'D' i 'M' redom i sada ovaj skup simbola nazivamo Rimskim brojevima. U ovakvom skupu, pojam broja četiri nije moguće prikazati jednim simbolom već moramo da ga prikažemo kombinacijom dva ili više osnovna simbola, tj. kao 'IV'. Za sisteme poput Rimskog, svaki simbol je jednoznačno definisan i ne zavisi od pozicije na kojoj se nalazi u broj i takve sisteme nazivamo nepozicionim sistemima. Nasuprot njima, Arapski brojevni sistem, koji koristi simbole 0 1 2 3 4 5 6 7 8 9 kao osnovne, je pozicioni sistem i kod njih važi da se jedan isti simbol drugačije interpretira u zavisnosti od njege pozicije u broju. Tako na primer, 1 u 123 označava broj sto, dok 1 u 313 označava broj deset. Ono što razlikuje pozicionog od nepozicionog sistema je to što dolazi sa pojmom o **osnovi sistema**.

Definicija 0.1 (Osnova brojevnog (pozicionog) sistema)**.**

Osnova brojevnog sistema je broj različitih brojeva koje možemo da zapišemo korišcenjem samo jednog osnovnog simbola sistema.

Gornju definiciju možemo i da preformulišemo kao: broj jednocifrenih brojeva. S toga, Arapski brojevni sistem je dekadni brojevni sistem, osnova mu je 10.

Čitanje brojeva

Fokusirajmo se trenutno samo na Arapske brojeve. Ma kako čudan broj napisali, ubedjen sam da bi ga svi uspešno pročitali. Recimo, 2465 bi pročitali kao ”dve hiljade četristo šezdeset pet”, što je

naravno skroz tačno. Medjutim, odgovor na pitanje "A kako ste uspeli to da pročitate?" uopšte nije tako jednostavan!
Razmislimo kako to mi tačno čitamo brojeve. Odgovor na ovo pitanje možda delovati malo začudjujuće, ili još bolje, kontra intuitivno! Iako u broju 2465 mi prvo izgovorimo kranje levi broj, dvojiku, zapravo brojeve čitamo sa desna na leva! I to na sledeći način:

- Prvo pročitamo broj 5 i zapamtimo ga.
- Zatim pročitamo broj 6, pomnožimo ga sa 10 i saberemo sa zapamćenim brojem 5, da dobijemo 65. Sada njega pamtimo.
- Sledeći broj na koji nailazimo je 4. Njega pomnožimo sa 100 i ponovo sabiramo sa prethodno zapamćenim brojem, što je 65. Taj novodobijeni broj, 465, ponovo pamtimo
- Konačno, čitamo broj 2, množimo ga sa 1000 i sabiramo sa zapamćenim brojem da dobijemo 2465

Dozvolite da ilustrujem i ubedim vas u ovaj postupak sa malo težim brojem 23987299792458. Pretpostavljam da, čak iako ste uočili da poslednjih 9 cifara čine brzinu svetlosti u vakuumu koja je izražena u milionima metara u sekundi, niste uspeli istom brzinom da odgovorite koji je to broj, kao za broj 2465. Mislim da nije loše pretpostaviti da ste verovatno koristili tehniku sa odvajanjem tri-po-tri broja zarezima sa leva na desno, kako bi ste preglednije videli da je reč o "dvadeset tri biliona devetsto osamdeset sedam milijardi dvesta devedeset devet miliona sedamsto devedeset dve hiljade četristo pedeset osam".
Slično gore opisanoj metodi, i ovde ste čitali brojeve s desna na levo, s tim da ste to radili nad grupama sa po najviše 3 broja. Kada pa žljivije izgovorite neki
broj, možete videti da baš opisujete ovaj postupak: ”dve hiljade

četiri stotina šezdeset i pet”, dakle 2*1000 + 4*100 + 6*10 + 5 = 2465.

Pre nego što uopštimo postupak čitanja brojeva za bilo koju osnovu, prodjimo još jednom detaljno kako čitamo brojeve u osnovi (10), tj. u onoj koju koristimo u svakodnevnom životu.

Čitanje brojeva u osnovi (10)

Prisetimo se jednog jednostavnog ali važnog matematičkog koncepta - stepenovanje brojeva. Uprošćena definicija[5] bi glasila:

Definicija 0.2 (Stepenovanje)**.**

- Stepen nekog zadatog broja je broj dobijen množenjem zadatog broja samim sobom onoliko puta koliko iznosi stepen.
- Nulti stepen bilo kog broja je 1

Primeri:

- $2^3 = 2 \cdot 2 \cdot 2 = 8$ - tri puta pomnožimo dvojiku samu sa sobom (dižemo dvojiku na kub)
- $5^2 = 5 \cdot 5 = 25$ - dva puta pomnožimo peticu samu sa sobom (dižemo peticu na kvadrat)
- $10^4 = 10 \cdot 10 \cdot 10 \cdot 10 = 10000$ - četiri puta pomnožimo desetku samu sa sobom
- $16^1 = 16$ - samo šesnaest
- $2^0 = 1$
- $5^0 = 1$
- $1234^0 = 1$
  Poslednje pravilo o tome da bilo koji broj dignut na nulti stepen

daje broj jedan je samo dogovor matematičara (radi lakšeg i potpunog definisanja stvari) i ne treba mnogo razbijati glavu zašto je tako.

Konačno možemo preći na iznošenje postupka (tj. algoritma) za čitanje brojeva:
Potrebno je pamtiti 3 broja. Jedan označava tekuću sumu, drugi označava brojač koji nam služi za stepenovanje i treći je trenutna cifra na kojoj se nalazimo.

1. Krenuti čitanje sa desna na levo cifru po cifru, postaviti sumu i brojač na 0.
2. Osnovu brojevnog sistema (u našem slučaju 10) dići na stepen trenutnog brojača
3. Tako dobijen broj pomnožiti sa trenutnom cifrom
4. Sada, taj novodobijeni broj sabrati sa tekućom sumom i zapamtiti ga kao novu tekuću sumu
5. Pomeriti se jednu cifru u levo i uvećati brojač za 1
6. Vratiti se na korak 2) sve dok se ne obrade sve cifre
7. Kada su sve cifre obradjene, traženi broj je ono što je pamtimo u sumi

Ilustrujmo ovaj postupak sa brojem 2465:

- (Korak 1) Osnova = 10, brojač = 0, suma = 0, treunta cifra = 5
- (Korak 2) $10^0 = 1$ - dižemo osnovu na trenutni brojač
- (Korak 3) $1 \cdot 5 = 5$ - dobijen broj množimo sa trenutnom cifrom
- (Korak 4) $5 + 0 = 5$ | suma = 5 - taj broj smo sabrali sa tekućom sumom, što je bilo 0, i dobijeni broj zapamtili kao novu tekuću sumu
- (Korak 5) trenutna cifra = 6, brojač = 1, suma = 5

- (Korak 2) $10^1 = 10$
- (Korak 3) $10 \cdot 6 = 60$
- (Korak 4) $60 + 5 = 65$ | suma = 65
- (Korak 5) trenutna cifra = 4, brojač = 2, suma = 65
- (Korak 2) $10^2 = 100$
- (Korak 3) $100 \cdot 4 = 400$
- (Korak 4) $400 + 65 = 465$ | suma = 465
- (Korak 5) trenutna cifra = 2, brojač = 3, suma = 465
- (Korak 2) $10^3 = 1000$
- (Korak 3) $1000 \cdot 2 = 2000$
- (Korak 4) $2000 + 465 = 2465$ | suma = 2465
- (Korak 5) nema više cifara
- (Korak 7) Resenje = 2465

Dakle, dobili smo tačno onaj broj koji je i trebalo. Sasvim logično, broj 2465 zapisan u osnovi (10) je broj 2465 pročitan u osnovi (10). Koristeći ovaj postupak možemo prevesti broj zapisan u bilo kojoj osnovi u broj zapisan u osnovi (10)!

Čitanje brojeva u proizvoljnoj osnovi

Neka je brojevni sistem zadat sa n cifara (osnove n) i neka su te cifre $c_0, c_1, c_2, \ldots, c_{n-2}, c_{n-1}$. Tada svaki ceo broj k možemo prikazati kao[6]:

$$k = c_{i_r} c_{i_{r-1}} \ldots c_{i_1} c_{i_0}$$

$$k = c_{i_r} \cdot n^r + c_{i_{r-1}} \cdot n^{r-1} + \cdots + c_{i_1} \cdot n^1 + c_{i_0} \cdot n^0, \quad \text{tj.} \qquad (3$$

$$k = c_{i_r} \cdot n^r + c_{i_{r-1}} \cdot n^{r-1} + \cdots + c_{i_1} \cdot n + c_{i_0}, \quad \text{jer } n^1 = n \text{ i } n^0 = 1$$

Gde je svaki $c_{i_z}, z \in 0, \ldots r$ jedan od (jednocifrenih) brojeva $c_0, c_1, \ldots, c_{n-2}, c_{n-1}$.
Drugim rečima, svaki broj može da se prikaže kao zbir umnožaka jednog od osnovnih cifara i stepena date osnove. ($2465 = 2 \cdot 10^3 + 4 \cdot 10^2 + 6 \cdot 10 + 5$)

Uvedimo konačno **binarni sistem**, tj. sistem osnove (2)[7]

Definicija 0.3 (Binarni sistem)**.**

Binarni sistem je sistem sa samo dve cifre: $0 \quad 1$

Dakle, za binarni sistem imamo samo dve cifre na raspolaganju i osnova tog sistema je (2). Iako na prvi pogled ovo možda deluje restriktujuće, može se pokazati da su dve cifre sasvim dovoljne da zapišu bilo koji broj. Ovo ćemo potvrditi tako što ćemo naći neki algoritam koji može da prebaci bilo koji broj zapisan u dekadnom sistemu u binarni i obratno.

Već smo naveli jedan način kako možemo da prebacimo (odnosno pročitamo) broj zapisan u binarnom sistemu u dekadni, modifikujući korak 2) tako da umesto broja 10, dižemo broj 2 na stepenove. Druga, mnogo brža i lakša varijanta je da iskoristimo (*). Tako recimo, prebacivanje broja 1011_2 u dekadni sistem je:

$$1011 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 8 + 2$$

$1011_2 = 11_{10}$

Za vežbu, uverite se da je:

- $111_2 = 7_{10}$
- $1_2 = 1_{10}$

- $1001_2 = 9_{10}$

Prebacivanje brojeva iz dekadnog u binarni zapis je nešto teži postupak. Prvo tražimo najveći mogući stepen na koji možemo da dignemo dvojiku tako da dobijemo broj manji ili jednak zadatom broj u dekadnoj osnovi. Nakon toga, idemo redmo za svaki stepen manje od najvećeg mogućeg i pitamo se da li treba da ga uključimo u sumu ili ne. Ako treba, na tu poziciju pišemo 1, a ako ne onda 0. Drugačije, možemo da delimo zadati broj u dekadnoj osnovi dvojikom sve dok se ne dobije nula, pamteći ostatak prilikom takvog deljenja. Ukoliko je ostatak 1, pamtimo 1, a ukoliko je ostatak 0, pamtimo 0[8] . Traženi broj zapisan u binarnoj osnovi se dobija tako se pročita taj niz ostatak u obrnutnom smeru. Najlakše je videti na primeru 1203:

```
1203 / 2 = 601 | ostatak = 1
601 / 2 = 300  | ostatak = 1
300 / 2 = 150  | ostatak = 0
150 / 2 = 75   | ostatak = 0
75 / 2 = 37    | ostatak = 1
37 / 2 = 18    | ostatak = 1
18 / 2 = 9     | ostatak = 0
9 / 2 = 4      | ostatak = 1
4 / 2 = 2      | ostatak = 0
2 / 2 = 1      | ostatak = 0
1 / 2 = 0      | ostatak = 1
```

Kada se ovi ostatci pročitaju odozdo nagore dobijamo:
$1203_{10} = 10010110011_2$

Binarni brojevi su od ključne važnosti u svetu računara. Svaki računar je sastavljen od miliona tranzistora, poluprovodnog materijala sačiinjenog od nekog

metal-oksida (najčešće silicijum-oksida). Tranzistori su dalje sačinjeni od sitnih (uglavnom bakarnih ili platinastih) žica kroz koje može da protiče struja. Ukoliko postoji strujanje elektrona kroz neku posmatranu žicu ili skup računarskih komponenti, mi možemo da joj dodelimo brojčanu vrednost 1, a ukoliko ono ne postoji (tačnije ukoliko napon nije u odredjenom intervalu), onda joj dodeljujemo brojčanu vrednost 0[9]. Dakle, iako koriste drugačiju osnovu, binarni i dekadni
sistemi su ekvivalentni i sposobni da izvršavaju iste operacije! Naravno, ovim nismo rešili problem da kad-tad može da nam ponestane žica, te računar ne može baš svaki broj koji poželimo da zapiše (mada može njih dovoljno mnogo!).

Sabiranje brojeva u proizvoljnoj osnovi

Ispostavlja se da postupak sabiranja brojeva ne zavisi od osnovne u kojoj radimo. U ovo možemo lako da se uverimo ako malo bolje porazmislimo o tome da je sabiranje ništa drugo nego ukupno prebrojavanje elemenata neke dve grupe. Zbir tih elemenata neće zavisiti od toga šta su ti elementi (brojevi, babe, žabe ...) kao ni od toga kako želimo da zapišemo taj zbir. Zbog toga, sabiranje binarnih, a i brojeva u bilo kojoj drugoj osnovi je trivijalno i možemo koristiti metod potpisivanja, sa dodavanjem jedinice na narednu veću cifru kada je zbir prethodnih dveju veći ili jednak osnovi u kojoj radimo. Npr:

```
1
2         1
3  -------------
4      100101
5  +   010110
6  -------------
```

```
7       111011
```

Rešenje prethodnog primera smo dobili tako što smo poravnali brojeve po najmanjoj cifri i zatim krenuli da sabiramo s leva na desno cifru po cifru, prateći šablon:

- $0 + 0 = 0$
- $0 + 1 = 1 + 0 = 1$
- $1 + 1 = 0$ sa prebacivanjem 1 na narednu cifru

Bitovi, Bajtovi i dalje ...

Kao što smo rekli, apsolutno sve je u računaru zapisano preko cifara iz binarnog sistema, a kompjuter sam (uz pomoć programera koji su ga napravili) tumači te cifre na različite načine u zavisnosti od konteksta. Već smo videli da za jedan četvorocifren dekadni broj (poput 1203) nam je potrebno 11 cifara da prikažemo taj isti broj u binarnoj osnovi, što dovodi do zaključa da se informacije u sistemu računara tumače po većim grupama, te su programeri nazvali neke najčešće grupacije u zavisnosti od broja cifara koje uzimamo kao celinu:

- Bit - jedna cifra (jedna žica) i najmanja jedinica u memoriji. [Oznaka: b]
- Nibble - 4 Bit-a, ne koristi se toliko često. [Oznaka: N]
- Byte - 8 Bit-a, uglavnom čini osnovnu jedinicu u memoriji. [Oznaka: B]
- Word - Zavisi od arhitekture računara i predstavlja dužinu podatka koje proces obradjuje; stariji računari su uglavnom bili 16bit-ni, a današnji su najčešće 32bit-ni ili 64bit-ni

Pored ovih oznaka, postoji o osnovna konvecija o tipovima podataka. Dužina tih tipova je uglavnom standardnizovana ali može da varira, te ovde navodimo dužinu koju pretstavljaju u Javi:

- int - integer, tj. ceo broj. Iznosi 4B = 32b
- char - character, tj. karakter odnosno simbol (slovo, broj ili specijalan simbol). Iznosi 1B
- double - double precision, tj. realani broj sa pokretnim zarezom dvostruke tačnosti. Iznosi 8B
- String - niska, tj. niz karaktera i zavisi od duzine karakter.
- itd ...

Formalni jezici i paradigme programskih jezika

Kao i kod neformalnih (govornih) jezika, formlani jezici se definišu nad nekim rečnikom čije reči spajamo u smislene rečenice prateći odgovarajuca gramatička
pravila. Dakle, sastoji se od sintaksnog i gramatičkog dela.
Programski jezici su samo uži deo formalnih jezika, gde su ključne reči rečnik, sintaksa označava gramatčka pravila a jedna instrukcija programskog jezika je jedna rečenica. Opet, kao i kod neformalnih jezika, da bi smo savladali programski jezik, potrebno je da naučimo njenu sintaksu i da dosta vežbamo pisanje programa (analogno pričanju) kako bi postali fluentni.
Kod neformalnih jezika imamo klasifikaciju po nekim grupama, podgrupama ... dok programske jezike klasifikujemo u zavnisnosti od načina izvršavanja odredjenih funkcija. Neke od tih podela su na:

1. Imperativne i deklarativne

- *Imperativni* programgski jezici su oni jezici kojima se govori tačno šta na koji način da izvši. Drugim rečima, oni očekuju konkretne naredbe koje postupno opisuju kako se dolazi do rešenja. Primer ovakvih prograskih jezika su Java, C#, C++, C, Python, JavaScript ...
- *Deklarativni* programski jezici su oni jezici kojima se izlaže tačna definicija problema, a programski jezik sam, u pozadini, dolazi do rešenja. Ovavkim programskim jezicima govorimo šta želimo da dobijemo kao rezultat i pod kakvim to uslovima treba da se dogodi. Primer je SQL, Prolog, Haskell ...

2. Viskog i niskog nivoa
   - *Jezici visokog nivoa* su oni jezici koji ne komuniciraju direktno sa procesorskom memorijom, tj. registrima. Ovoj grupi spada veći broj programskih jezika, poput Java, C++, C, C#, Python, Haskell ...
   - *Jezici niskog nivoa* komuniciraju direktno sa procesorskom memorijom i to su asemblerski jezici.
3. Opšte oblastne i oblasno-specifične svrhe
   - *Jezici opšte oblastne svrhe* su jezici koji su dizajnirani da mogu da (nešto malo lakše a nešto malo teže) reše različite tipove problema. Ovoj grupi takode pripada veći programskih jezika, pa se tako Java koristi za programiranje igrica[10], sistema u ku´cnim aparatima, web apleta ...
   - *Jezici oblasno-specifične svrhe* su jezici koji su dizajnirani da reše samo odredjenu vrstu problema i često tu vrstu problema rešavaju znatno lakše od jezika opšte svrhe. Tako se na primer Verilog ili HDVL koristi za dizajniranje i testiranje mikročipova.

4. Kompajlirane i Interpretirane
    - *Kompajlirani* programski jezici su oni koji pre izvršavanja njihovih programa moraju da se kompajliraju (prevedu) u mašinski (ili izvršni) kod. Primer ovakvih programskih jezika je C.
    - *Interpretirani* programski jezici su oni koji se ne prevode u mašinski kod, vec interpretator u hodu izvršava i prevodi kod. Primer ovavih programskih jezika su JavaScript i Python.
    - Ovakva karakterizacija polako gubi smisao jer sve više i više programskih jezika nastoji da inkorporira obe ove paradigme, pa je tako, izmedu ostalih, Java i kompajliran i interpretiran jezik. Java se prvo prevodi pomo ću Javinog kompajlera do Byte koda, a dalje se pomo ću interpretatora interpretira i izvršava.
5. Objektno orijentisane i neobjektno orijentisane.
   Prosto rečeno, *Objektno orijentisani* programski jezici su oni koji dozvoljavaju konstrukciju objekata, a *neobjektno orijentisani* oni koji to ne zadovoljavaju. Većina prograskih jezika danas nastoji da bude objektno orijentisan jer već niz decenija ova paradigma predstavlja standard na koji su se programeri navikli. Primer objektno orijentisanih jezika su Java, C, C++, JavaScript, Python, Ruby ... dok od neobjektno orijentisanih izdvajamo C i Pascal.
6. Statički i dinamički tipizirane
    - Kod *statički tipiziranih* programski jezika svakoj promenjivoj se unapred zna tačan tip i njen tip nemože ni na koji način da se promeni, prilikom kompilacije programa. Iako restriktivna, ova paradigma nam (koliko toliko) garantuje sigurnost pravilnog deklarisanja promenjivih. Ovakvi programski jezici su Java, C#, C++.

- Kod *dinamički tipiziranih* programskih jezika promenjivima se dodeljuje tip tek prilikom izvršavanja programa. Ovo nas znatno manje limitira i ne tera nas da obraćamo dodatnu pažnju na tačno tipiziranje svake promenjive, ali zato nam ništa ne garantuje sigurnost prilikom pokretanja programa. Ova paradigma u poslednje vreme raste po poularnosti, delom jer je lakše novajlijama da savladaju znatno smanjen skup ključnih rečni. Ovavki jezici su Python, Lua, JavaScript.

7. Medju ostalim paradigmi izdvajamo:
    - *Funkcionalni* programski jezici su oni jezici čiji se program sastoji od definicija funkcija i njihovih kompoicija. Cisto funkcionalni jezici dozvoljavaju samo ovakav način pisanja programa dok većina drugih jezika teži da omogući programeru da se služi ovakvom paradigmom po potrebi. Primer čisto funkcionalnog jezika je Haskell, dok Java i C# imaju mogu´ćnost korišćenja ove paradigme iako nisu čisto funkcionalni.
    - *Generični* programski jezici su oni jezici koji dopuštaju deklarisanje generičkih promenjivih. Ovo nam dopušta da jedan problem apstraktujemo i opišemo njegovo rešenje na visokom nivou koji pokriva sve konkretne slučajeve.
    - *Višenitni* progremski jezici omogu ćavaju paralelno izvršavanje više naredbi. Svaki moderan računar ima ovu sposobnost, pa tako u isto vreme možemo da čitamo knjigu, slušamo muziku i pomeramo miša. Logično, ovo je jedan od najbitnijih odlika s obzirom da bez paralelnosti, kompjuteri se ne bi previše razlikovali od običnih digitrona.
    - *Logički* programski jezici su najčešće deklarativni jezici koji su bazirani na formalnoj logici. Nažalost ovakva paradigma

nije doživela značajani uticaj u svetu programiranja. Najpoznatiji primer ove paradigme je Prolog.
- *Prirodni* prograski jezici su jezici čija sintaksa više poce ća na svakodnevni, govorni jezik nego na jezik računara. Najpoznatiji primer ove paradigme je COBOL (COmmon Business Oriented Language). Ni ova paradigma nije u velokoj upotrebi, mada sve više i više programskih jezika pokušavaju da se odvoje od hardvera i budu što čitljiviji za prosečnu osobu (poput Pythona).

Naravno, jedan prograski jezik ne mora samo da pripada jednoj paradigmi (kao što možemo videti iz primera). Tako, Javu definišemo kao imperativan, objektno orijentisan, opšte namenski jezik visokog nivoa sa mogućnošću paralelnog programiranja; drugim rečima, višeparadigmalan jezik. [11]

Istorija programskog jezika Java

Prvi kompjuteri, u prvoj polovini XX veka, bili su kreirani tako da obavljalju vrlo specifične (i za današnji standard) jednostavne operacije, bili su veličine tipične dnevne sobe i sastavljeni od velikih magnetnih komponenti. Zato, nije bilo moguće odvojiti pojmove "pisanje programa" i "konstrukcija računara" jer su se oni obavljali uporedo. No, kako je tehnologija rapidno napredovala, ovakvi računari su zamenjenji mnogo sitnijim kompjuterima baziranim na električnim komponentima. Ovakvi računari su bili programibilni, tj. jedan isti računar je mogao da izvršava različite programe u zavisnosti od konfiguracije na koju je namešten. Ovakve konfiguracije su se nameštale ručno, preraspodelom načina na koje su glavne komponente medjusobno povezane. Ali ni nešto ovako nije bilo dovoljno efikasno za to doba, te su se ubrzo pojavile

perforirane trake. Te dugačke trake su bile podeljene na sitne ćelije i svaka ćelija je označavala najsitniju programibilnu jedinicu (u zavisnosti od računara to je bio bit, bajt ...). Ukoliko je ćelija izbušena, to je označavalo 1, a ukoliko nije onda je taj deo bio markiran kao 0. Bušenjem te trake pisao se program koji je onda fizički ubacivan u kompjuter koji ju je dalje gledao i tražio te rupe. Čitav ovaj postupak je dosta pocećao na mehanizam zvučne kutije.

Posle ovoga je usledila predominacija asemblerskih programskih jezika. Ovakvi programski jezici su komunicirali direktno sa registrima, memorijom sadržanom u procesoru, srcu svakog računara. Iako bi za današnji standard ovi jezici bili jako tekši za programiranje (mada se idalje često koriste), u to vreme su bili znatna olakšica svakom programeru, jer je kod kucan pomoću računara a nije pravljen na fizičuki način.

Primer programa 'HelloWorld' napisanog u x86 NASM[12] asembleru za Linux operativne sisteme:

```
section .data
    msg db 'Hello World!\n', 0xa
    len equ & - msg

section .text
    globabl _start

_start:
    mov edx, len
    mov ecx, msg
    mov ebx, 1
    mov eax, 4
    int 0x80

    mov ebx, 0
```

```
16      mov eax, 1
17      int 0x80
```

Kao što se da videti, ni ovavki programski jezici nisu toliko pogodni za pisanje malo ozbiljnijih programa. To je navelo Denisa Ričija[13] da osmisli programski jezik koji je uspeo toliko da olakša proces pisanja programa, da se sada svi programski jezici (neki manje, a neki više) pozivaju na isti princip definisanja sintakse i na
taj način se proširilo i interesovanje za pisanjem prograskih jezika i to je upravio bio C programski jezik. C programski jeziki je prvi put izašao u javnost 1972. godine i pripadao je imperativnoj, strukturiranoj, rekurzivnoj i statički-tipiziranoj paradigmi, ali nije bio objektno orijentisan. Odlikovalo ga je to što programer nije direktno komunicirao sa registrima, već je memorijom upravljao pomoću pokazivačke aritmetike. Primetimo da pisanje programa ”Hello world”u C programskom jeziku se ne razlikuje toliko od istog programa napisanog u modernim jezicima:

```c
#include <stdio.h>

int main(int argc, char* argv[], char* env[]) {
    printf("Hello World!\n");
    return 0;
}
```

Osamdesetih godina, nakon što je C programski jezik maksimalno iskoristio svoj potencijal, ideje o klasam i objektima su počele sve više i više da kruže i 1985.
godine izašao je programski jezik koji je želeo da nadogradi C. U njegovu čast ovaj novi programski jezik je nazvan C++ koji je imao sve odlike C programskog jezika sa mnogo dodatih stvari, od kojih je jedna i mogućnost kreiranja klasa i objekata. O tome kako je C++ napisan da bude lak za korišćenje, brz i efikasan u

izvršavanju komandi i bogat korisnim funkcionalnostima svedoči to što je i dan danas jedan od najkorišćenijih programskih jezika, a glavni izbor kod 99%
programera koji se takmiče na medjunarodnim takmičenjima i olimpijadama.

Otprilike u isto vreme kada se razvija C++, dolazi do procvata informacionih tehnologija i sve više mašina i električnih aparata je počelo da se proizvodi. Svaka kompanija je pokušavala da nadmaši svoje suparnike i sve više i više različitih arhitektura računara je počelo da se proizvodi. Kako je svaka od tih arhitektura dovoljno različita od prethodnih, to je za svaku iznova morao da se piše kompajler za svaki jezik koji bi se na njoj koristio. Ideju za programskim jezikom koji bi bio arhitektonski neutralan (čiji se kompajliran kod ne bi razlikovao od arhitekture do arhitekture) je dobio Džejms Gozling[14], ko je u to vreme radio u Sun Microsystems kompaniji. Dana 23. maja 1995. godine izašla je prva verzija programskog jezika Oak, ali to ime je ubrzo zamenjeno imenon Indenežanskog ostrva sa kojeg potiče Džejmsova omiljena kafa, ostrva Jave.
Prateći svoj moto, ”Write once, run everywhere”[15] prograski jezik Java se rapidno
razvijao u razne oblasti poput programiranja ugradjenih sistema, pisanje web appleta ... Godine 2010. održavanje Jave je preuzela kompanija Oracle, koja je idalje
unapredje, čineci verziju 24 najnovijom verzijom koja je izašla krajem 2024. godine.

Primer 'Hello World' programa napisanog u Javi:

```
1
2  public class HelloWorld {
```

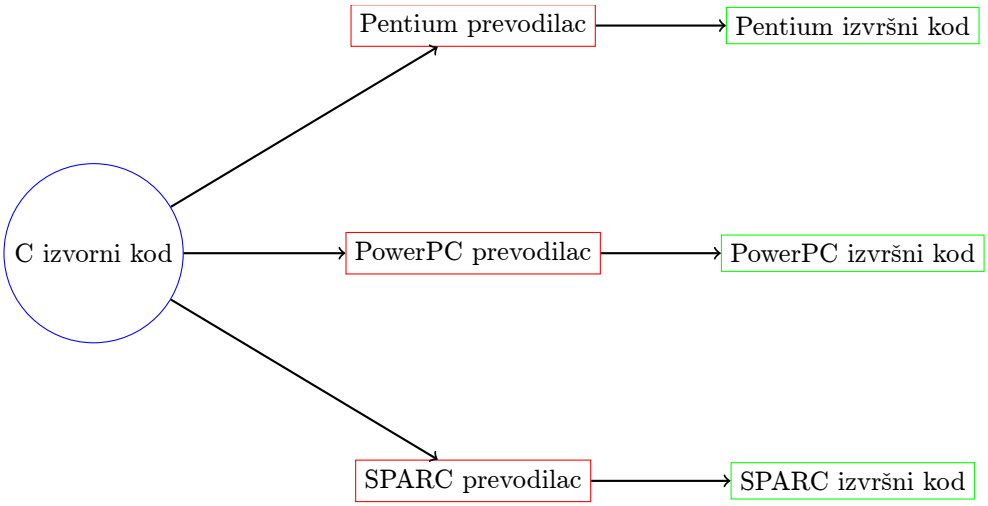
```
3
4       public static void main(String[] args) {
5           System.out.println("Hello world!");
6       }
7   }
```

Specifikacije Jave

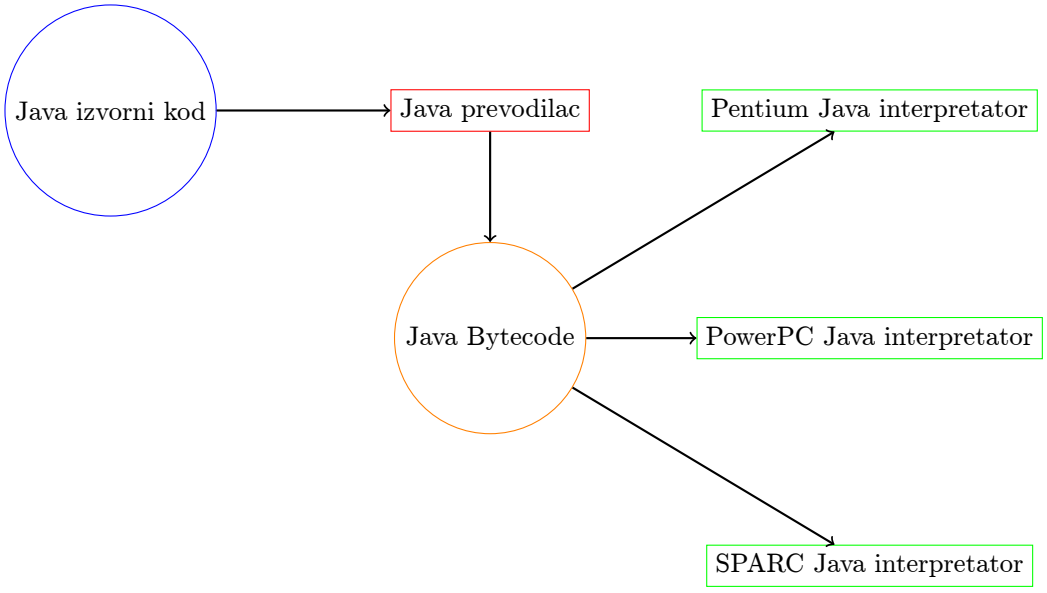
Java je objektno-orijentisan, opšte-namenski, imperativan, kompajliran pa interpretiran jezik visokog nivoa baziran na principu "Write once, run anywhere", kojeg je kreirao Džejms Gozling 23. maja 1995. godine u Sun Microsystems kompaniji. Java se može preuzeti u različitim oblicima u zavisnosti od upotrebe:

- Java SE (Standard Edition) - Standardni Java paket
- Java ME (Micro Edition) - Umanjena verzija zasnovana na SE verziji
- Java EE (Enterprise Edition) - Namenjena ogromnim projektima
- Java Card - Namenjen za pametne kartice

Java je takodje obogoćena raznim bibliotekama i API-ima (Application Programming Interface) koji je čine jednim od najrazvijenijih programskih jezika. Ono što odlikuje Javu od ostalih sličnih programskih jezika poput C# jezika je to što je Java `cross-platform`, odnosno da ne zavisi od operativnog sistema ili uredjaja, tj. portabilna je. Ova portabilnost je omogućena činjenicom da se Java prvo prevodi (kompajlira) do Java Bytecode-a a zatim se taj Bytecode dalje interpretira u Javinoj virtuelnoj mašini. Naime, tradicionalan način prevodjenja koda napisanog od strane programera (npr. C koda) u mašinski, tj. izvršni kod je izgledao nešto poput:



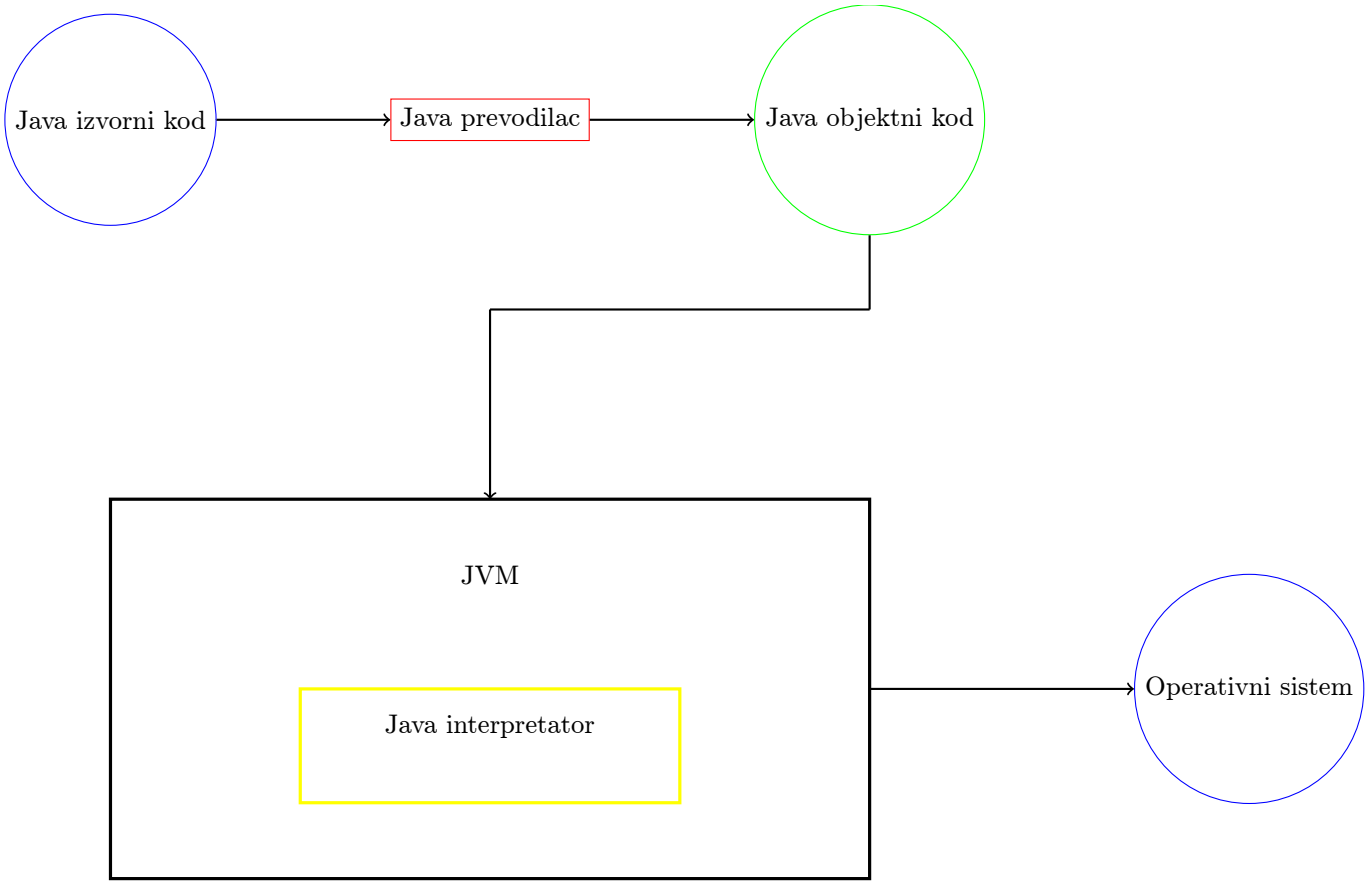
Dakle, za svaku računarsku arhitekturu (poput Pentium, PowerPC, SPARC ...), u ovom tradicionalnom načinu moramo da pišemo zaseban prevodilac za C jezik, koji će prevoditi C izvorni kod u izvršni kod za tu zadatu arhitekturu. Kod Javing Bytecode pristupa imamo sledeći slučaj:



Dakle, umesto da imamo 3 različita prevodilaca, imamo jedan isti za svaku arhitekturu koji na isti način prevodi javin izvršni kod u bajt-kod za kojeg kažemo da je binaran i arhitektonski neutralan koji se dalje interpretira posebno za svaku arhitekturu. Ali sačekajmo malo, ako pogledamo malo bolje, u prvom - tradicionalnom pristupu imamo jedan čvor manje u našem grafu nego u drugom pristupu, jer na kraju krajeva, opet moramo za svaku arhitekturu da pišemo poseban interpretator. Šta smo ovim

dobili?
Odgovor na ovo ne tako trivijalno pitanje leži u tome sa je pisanje interpretator znatno jednostavnije od pisanje kompajlera, odnosno prevodioca. I ne samo to, zamislimo da koristimo C programski jezik na našem računaru da napravimo neki program. Ukoliko bi smo taj program kompajlovali i tako ga preneli na računar sa drugačijom arhitekturom ili operativnim sistemom, naš kod ne bi radio. Umesto toga, morali bi da prenesemo ceo izvršni kod (možda hiljade i hiljade fajlova) na taj drugi računar i tamo ga ponovo kompajlovati zasebnim C kompajlerom. S druge strane, ako isti program napišemo u Javi i kompajliramo ga, dovoljno je da samo taj kompajlirani fajl prenesemo na drugi računar, jer dokle god taj drugi računar ima javin interpretator on može pročitati tako kompajliran bajt kod. U oba slučaja pretpostavljamo da drugi računar ima neophodnu tehnologiju da pročita zadate fajlove - ili C kompajler ili Java interpretator, ali u drugom slučaju šaljemo samo bajt kod a ne ceo izložen kod. Naravno ovako dobra prenosivost mora da povuče sa sobom i par mana, a u ovom slučaju to je gubitak u performansama jer je interpretiranje (po pravilu) sporije od čitanje kompajlovanog koda30. Još od ranih faza razvića Jave, ovakav gubitak u performansama je koliko - toliko uklonjen uvodjenjem JIT kompajlera (Just-In-Time), kojeg nećemo detaljno opisivati ali ćemo definitivno uživati u njegovim pogodnostima tokom kursa.

Izvšavanje Java programa se vrši pomoću Java virtuelne mašine (JVM) i bazira se na sledećem principu:



Jezgro Jave čini JVM. JVM je mali virtuelni kompjuter koji živi samo u memoriji računara odnosno hardvera na kome se realizuje, ali poseduje sve komponente kao i pravi računar (Aritmetiču logičku jedinicu - ALU, stek memoriju, heap ...) i sadrži sistem za čitanje klasa, sistem za izvršavanje, automatski sakupljač otpadaka (nekorišćenje i zastarele memorije), regulator niti i procesa i još mnogo toga. Svaki put kada pokrećemo Java program, kreira se mala virtualna mašina u našoj RAM (Random Access Memory) memoriji koja izvršava naš program.

Pored JVM-a, ključnu ulogu u pisanju Java programa igraju i Java API klase koje je razvio Sun Microsystems. Klase unutar API-a se koriste da omoguće Java programu da komunicira sa raznim računarskim procesima i one su grupisane po paketima (folderima), pri čemu jedna paket sadrži ogroman broj klasa. Najznačajniji API je `Core API` koji sadrži pakete sa klasama za koje se garantuje da su uvek dostupni bez obzira na verziju Jave koju koristimo. Paketi Core API-a su sledeći:

- java.lang (language) - Sastoji se od centralnih klasa poput klasa-omotača za prote tipove podataka i nikada se ne mora explicitno uvoditi jer je automatski uveden za svaku Java klasu.
- java.io (input output) - Ulazno izlazna biblioteka za rad sa periferijskim hardverom.
- java.util (utility) - Sastoji se od klasa za razne strukture podataka, klasa za parsiranje ulaznog toka i još mnogo toga. Na ovu klasu ćemo se najviše uzdati tokom trajanja kursa.
- java.net (network) - Sastoji se od klasa koje čine Javu mrežno zasnovanim jezikom.
- i mnogi drugi

Kao što možemo videti u ovom delu teksta, često korišćena reč je `klasa`, čije značenje ne možemo trenutno detaljno objasniti i definisati (ali ćemo tome težiti od samog početka). Za sada je prihvatamo kao pojam koji potiče iz činjenice da je Java objektno-orijentisan programski jezik i to ne bilo kakav vec pravi[16] objektno orijentisani programski jezik. Ova reč `pravi` se odnosi na činjenicu da svaki program u Javi koji napišemo mora činiti jednu klasu. Tako na primer, za realizaciju programa "Zdravo svete" potrebno je kreirati posebnu klasu:

```
1  public class ZdravoSvete {
2      public static void main(String[] args) {
3          System.out.println("Zdravo svete!");
4      }
5  }
```

dok kod programskog jezika poput C#-a, koji je vrlo sličan Javi, kreiranje ovakvog programa može da se uradi i bez kreiranja klase:

```
1  using System;
```

```
2
3  Console.WriteLine("Zdravo svete!");
```

Elementarne konstrukcije Jave

U elementarne konstrukcije Jave (tokene) ubrajamo elemente jezika koje kompajler izdvaja kao nedeljive celine prilikom prevodjenja programa i te elementarne konstrukcije su:

1. Identifikatori
2. Ključne reči
3. Literali
4. Separatori
5. Operatori
6. Komentari
7. Beline

Hajde da prodjemo svaku konstrukciju ponaosob:

1. Identifikatori
   Identifikatori služe za imenovanje promenjivih, objekata, funkcija, klasa ... Prilikom identifikacije, mi dodeljujemo lokaciji u memoriji (nezavisno od njene dužine) jedinstveno ime koje možemo da koristimo u našem kodu da pristupimo toj memorijskoj lokaciji. Tako da umesto da pričamo o memorijskoj lokaciji na poziciji `n` u koju smo upisali našu e-mail adresu, mi tu lokaciju možemo da imenujemo tako da ima smisla, npr sa `nasaEmailAdresa`, te svaki put kada želimo da joj pristupimo, to možemo da uradimo sa tim identifikatorom. Svaki identifikator će sadržati jedan ili više karaktera tako da prate sledeći šablon: i. U obzir dolaze samo: A-Z, a-z, tj. sva velika i mala slova engleske abecede (ANSII slova).

ii. 0-9, tj. svi arapski brojevi tako da **ne počinju brojem** (poželjno ih je svakako izbegavati).
iii. $, _, tj. simbol za dolar i donja crta.
iv. Ostali znakovi poput @ i blanko znaka (" ") ne smeju da se koriste!

2. Ključne reči
   Ključne reči su identifikatori koji imaju specijalnu namenu u jeziku Java i ne mogu se koristiti za imenovanje drugih stvari. U Javi razlikujemo sledeće ključne reči:

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

3. Literali
   Literali su reči koje predstavljaju neku vrednost. U Javi, to su konstante primitvnog tipa ili instance klase `String`. Postoje sledeci literali:
   a) Celobrojni
   b) Realni
   c) Znakovni
   d) Stringovni

e) Logički

a) Celobrojni literali
Celobrojni literali mogu biti zapisani kao binarni, oktalni, dekadni i heksadecimalni. Tako na primer, broj 21 može da se zapiše kao:
- 0b10101 -> 0b je prefiks za binarni broj
- 025 -> 0 je prefiks za oktalni broj (sistem baze 8)
- 21 -> dekadni brojevi nemaju prefiks
- 0x15 -> 0x je prefiks za heksadecimalni broj

b) Realni literali
Realni literali opisuju realne brojeve. Razlikujemo `float` i `double` literale koji označavaju brojeve u pokretnom zarezu sa jednostrukom, odnosno dvostrukom preciznošću, kao i pozicioni i eksponencijalni zapis. Primeri su: 3.14, -2.71, 0.555312, 123E-5 ...

c) Znakovni literali
Znakovni literali (karakteri) su bilo koji znakovi osim apostrofa `'` i obrnute kose crte `\`. Karakteri se navode izmedju dva apostrofa. Neki primeri su: 'a', '2', '@', 'X' ...
Specijalno, postoje takozvane eskejp sekvence. One započinju obrnutom kosom crtom prateći je neki simbol i imaju specijalne uloge. Eskpejp sekvence mogu biti:
- `\'` - ispisuje apostrof
- `\"` - ispisuje navodnik
- `\\` - ispisuje obrnutu kosu crtu
- `\r` - znak za povratak na početak reda (Home)
- `\n` - znak za novi red (Enter)
- `\t` - znak za tabulator (Tab)
- `\s` - znak za razmak (Space)

- `\b` - znak za povratak jedno mesto ulevo (Left Arrow)
- `\f` - znak za prelazak na novu stranicu

d) Stringovni literali

Stringovni literal je niz znakova (kraće, niska) koji se navodi izmedju dva znaka (duplih) navodnika `"`. Kao znak stringovnog literala može da se pojavi bilo koji znak osim apostrofa, znaka navodnika i obrnute kose crte. Da bi smo njih prikazali u našem stringovnom literalu potrebno je da ih izbegnemo pomoću eskejp sekvenci. Neki primeri stringovnih literala su "Hello World!\n", "" (prazna niska), "Ana ima 53 jabuke.", "x", ceo tekst Šekspira itd.

e) Logički literali

Logički literala ima samo dva: `true` i `false`, koji redom označavaju tačno i netačno. Prilikom bilo kakvog poredjenja uvek se dobija ovaj literal.

4. Separatori

   U Javi postoji nekoliko znakova koji služe za razdvajanje jedne vrste elementarnih konstrukcija od drugih. Na primer, u separatore spada simbol `;` koji služi za razdvajanje naredbi. Sledeći simboli čine separatore: `( ) [ ] { } , . ; :`. Separatori služe samo za razdvajanje i ne odredjuju operacije nad podacima.

5. Operatori

   Operatori omogućavaju operacije nad podacima. Podaci nad kojima se vrši neka operacija se nazivaju operandi. Prema broju operanada razlikujemo operacije kao:

   - Unarne operacije - samo jedan operand
   - Binarne operacije - dva operanda
   - Ternarne operacije - tri operanda
   - n-arne operacije - n operanada

     Prema poziciji operanada u odnosu na operaciju

razlikujemo:

- Prefiksne operacije - pišu se pre operanada
- Infiksne operacije - pišu se izmedju operanada
- Postfiksne operacije - pišu se posle operanada
  Postoje sledeći tipovi operatora:
  a) Operator dodele
  b) Aritmetički operatori
  c) Relacioni operatori
  d) Logički operatori
  e) Bitovski operatri
  f) Uslovni operator dodele
  g) Instancni operator

a) Operator dodele `=`
Operator dodele je infiksni operator koji se piše znakom jednakosti i on dodeljuje `literak` koji se nalazi sa desne strane jednakosti memorijskoj lokaciji označenoj `identifikatorom` sa desne strane jednakosti. Npr `x = 5` dodeljuje celobrojni literal 5 memorijskoj lokaciji označenoj simbolom `x`.
b) Aritmetički operatori
U aritmetičke operatore ubrajamo:
- Operator sabiranja `+` -> može biti prefiksni ili infiksni
- Operator oduzimanja/negiranja `-` -> infiksni/prefiksni
- Operator množenja `*` -> infiksni
- Operator delejanja `/` -> infiksni
- Operator modua `%` -> infiksni. Služi za računanje ostatka pri deljenju. Npr. $23\%4 = 3$
- Operator sledbenika `++` -> prefiksni ili postfiksni. Uvećava vrednost za 1.
- Operator prethodnika `--` -> prefiksni ili postfiksni. Umanjuje vrednost za 1.

c) Relacioni operatori

Relacioni operatori, odnosno operatori poredjenja služe za poredjenje vrednosti i kao rezultat uvek daju `logički literal` U relacione operatore ubrajamo:

- Operator jednakosti `==` -> Dvostruka jednakost služi za poredjenje literala operanda sa leve strane dvostruke jednakosti sa literalom sa desne strane dvostruke jednakosti. Ukoliko su ta dva literala po vrednosti jednaka, dobija se literal `true` inače dobija se literal `false`. Naravno, kao što zdrav razum nalaže ne možemo porediti babe i žabe, te nešto poput `3.14 == "Hello World!"` nema smisla jer ne možemo da poredimo realni literal sa stringovnim. Treba napomenuti da ako je neki operand `identifikator`, Java ce uzeti literalsku vrednost smeštenu u tu memorijsku lokaciju dokle god se ona poklapa sa tipom literala drugog operanda.
- Operator nejednakosti `!=` -> Ispituje da li literali operanada nisu jednaki. Ako nisu jednaki vraća literal `true` inače, ako su jednaki, vraća literal `false`.
- Operatori stogo manje `<`, manje (ili jednako) `<=`, strogo veće `>`, veće ili jednako `>=` rade na očekivan način.

d) Logički operatori

Postoje tri logička operatora: `!`, `&&` i `||` koji redom označavaju operator negacije, konjukcije i disjunkcije. Operator negacije je unaran i prefiksan dok su operatori konjukcije i disjunkcije infiksni i binarni. Kao operandi kod logičkih operatora mogu se pojavljivati samo logički literali i identifikatori čuje su vrednosti logički literali. Detaljan opis ovih operatora možete videti u narednom poglavlju: `1 Sintaksa jave`

e) Bitovski operatori

Operator po bitovima može biti logički ili operator pomeraja. Bitovske operatore čine:

- Bitovska negacija `~` -> prefiksan i unaran

- Bitovska konjukcija `&` -> infiksan i binaran
- Bitovska disjunkcija `|` -> infiksan i binaran
- Bitovska ekskluzivna disjunkcija `^` -> ifiksan i binaran
- Logičko pomeranje ulevo `<<` - infiksan i binaran
- Logičko pomeranje udesno `>>` - infiksan i binaran
- Aritmetičko pomeranje ulevo `<<<` - ifiksan i binaran

Detaljan opsi ovih operatora možete videti u narednom poglavlju: `1 Sintaksa jave`

f) Uslovni operator dodele

Uslovni operator dodele je ternarni i piše se pomoću znaka pitanja i dvotačke `?:`. Koristi se u formi

$$\text{logički\_izraz} \ ? \ \text{prvi\_ishod} : \text{drugi\_ishod}$$

gde je prvi ishod deo koda koji se izvršava samo ako je logički izraz bio tačan a drugi ishod se izvršava samo ako je logički izraz bio netačan.

g) Instancni operator

Pomoću instancnog operatora koji se piše kao `instanceof` proverava se da li je konkretan primerak objekat neke klase i takodje vraća logički literal. Ovaj operator ćemo koristi u drugom delu kursa.

6. Komentari

U Javi, kao i svim drugim programskim jezicima, komentari su namenjeni samo programeru koji radi sa datim kodom radi dodatnog pojašnjena koda ili neke napomene i oni se ne kompajliraju kao naredbe (kompajler ih preskače i zanemaruje). U javi postoje 3 vrste komentara:

- Jednolinijski -> pišu se iza duple kose crte `//`
- Višelinijski -> pišu se izmedju `/*` i `*/`
- Dokumentacioni -> pišu se izmedju `/**` i `*/` i omogućavaju dodatan markup.

7. Beline

Belina je znak koji nema grafički prikaz na izlaznom uredjaju. Beline služe za razdvajanje elementarnih konstrukcija i za lakše oblikovanje koda. Beline mogu biti:

- Razmak
- Horizontalni tabulator
- Znak za kraj reda
- Znak za novu stranu
- Znak za kraj datoteke
  Javi je potpuno nebitno koliko ima znakova beline izmedju dve elementarne konstrukcije i pored toga znakovi beline ne moraju da se piše nakon simbola za označavanje završetka naredbe `;` ili otvaranja/zatvaranja blokova koda `{ }`.

Treba dodatno napomenuti da operatore dodele možemo kombinovati sa aritmetičkim i bitovskim operatorima radi lakšeg zapisa, te imamo: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `<<<=`. Tako da umesto da pišemo

```
1   x = x + 5;
```

možemo da pišemo

```
1   x += 5;
```

Tipovi podataka u Javi

Tip podatka predstavlja jedan od osnovnih pojmova u strogo tipiziranim programskim jezicima. U Javi se pravi stroga razlika izmedju pojedinačnih tipova i nije dozvoljeno njihovo mešanje. U Javi razlikujemo dve vrste tipova:

1. Primitivni tip (prost tip)
2. Objektni tip (referencni / složeni tip)

Primitivni tipovi podataka su već unapred definisani i odmah stoje na raspolaganju programeru. Proste promenjive predstavljaju lokaciju u memoriji u koju će biti smešteni odgovarajući literal. Kao primitivni tipovi pojavljuju se `brojevni tipovi` i `logički tipovi`.

1. Brojevni tipovi
   Brojevni tipovi, kao što ime ime nalaže, čuvaju vrednosti koje predstavljaju nekakav broj. Oni se dele na celobrojne i realne tipove. Celobrojni tipovi mogu biti:
   - `byte` - 8-bitni tip za promenjive koje ne zahtevaju veliku memoriju.
   - `char` - 8-bitni tip za karaktere. Kako su karakteri celobrojni tipovi podataka, sve aritmetičke operacije koje možemo vršiti nad celim brojevima možem koristiti i sa karakterima. Tako na primerm 'a' + 1 će dati 'b'.
   - short - 16-bitni tip za male brojeve.
   - int - 32-bitni tip koji je najkorišćeniji u Javi.
   - long - 64-bitni tip za velike brojeve.
     Realni tipovi se dele na `float` i `double` koji zauzimaju redom 32 bita i 64 bita.
     Prilikom vršenja operacija nad brojevima može se dobiti nešto sto nije broj (npr. kada delimo broj nulom), s toga postoji posebna vrednost koju označavamo sa `NaN` (Not a Number).
2. Logički tipovi
   Logički, `boolean` tipovi kao vrednost mogu imati samo logičke konstante i one zauzimaju 1 bit u memoriji. Ovaj tip je dobio ime po Irskom matematičaru Džordžu Bulu (1815 - 1864), koji

je osnovao granu matematike, Bulovu algebru/logiku, koja igra klučnu ulogu u razvoju računarstva.

Pojam objekta je ključan u svakom objektno-orijentisanom programskom jeziku kao što je Java. Objekat predstavlja konkretizaciju, odnosno realizaciju/instancu odredjene klase. Objekti u Javi mogu biti korisnički, nizovni ili nabrojivi. O objektima ćemo više pričati u drugom delu kursa.

---

*Negativni brojevi

Do sada nismo pomenuli nijedan negativan broj, no sve što smo već naveli radi apsolutno isto i za negativne brojeve, barem što se tiče matematike. U matematici, negativni bojevi se uvode vrlo prosto: samo se doda znak `-` na vodeću cifru.[17] Ako malo bolje razmislimo, taj znak `-` moramo nekako da prikažemo u našoj memoriji, preko nekih bitova, odnosno brojeva (zapisanih u binarnoj osnovi), te naizlged nailazimo na kvaku 22[18]. Da bi se iz nje izvadili, možemo da izaberemo jednu cifru iz našeg broja da reprezentuje znak `+` odnosno znak `-`. Kako imamo dva znaka u opticaju, to nam tačno odgovara sa binarnim zapisom, te se uzima kao pravilo:

- Vodeća cifra (najlevlja cifra) se interpretira kao znak na sledeći način:
  - Ako je ta cifra 0, onda je u pitanju pozitivan broj
  - Ako je ta cifra 1, onda je u pitanju negativan broj
    Primeri (u komplementu dvojike):
- 0011_2 - pozitivan broj (+3)
- 1001_2 - negativan broj (-7)

Čitanje negativnih brojeva se radi na potpuno isti način kao i čitanje pozitivnih brojeva, s time što se vodeća cifra čita kao negativan broj, tj:

$$k = (-1) \cdot c_{i_r} \cdot n^r + c_{i_{r-1}} \cdot n^{r-1} + \cdots + c_{i_1} \cdot n^1 + c_{i_0} \cdot n^0$$

Tako je

$$1001_2 = (-1) \cdot 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = -8 + 1 = -7$$

$$1001_2 = -7_{10}$$

Prebacivanje negativnih brojeva iz dekadne u binarnu osnovu je nešto komplikovaniji. Pogledajmo prvo kako funkcioniše taj algoritam, a zatim ćemo analizirati zašto smo iskoristi upravo takav način. Naime, algoritam u pitanju je vrlo jednostavan:

1. Pretvoriti broj u binarni zapis smatrajući ga pozitivnim
2. Ukoliko je vodeća cifra 1 dopisati cifru 0 kao vodeću
3. Obrnuti svaku cifru u zapisu $-$ svaku jedinicu pretvoriti u nulu i svaku nulu u jedinicu
4. Tako dobijen broj sabrati sa 1
   Prodjijmo ovaj algoritam jednom da prebacimo broj -1203 u binarni zapis:

- (Korak 1) $1203_{10} = 10010110011$
- (Korak 2) Pošto je vodeća cifra 1 dodajemo 0 kao vodeću cifru: 010010110011
- (Korak 3) 101101001100
- (Korak 4)

```
101101001100
+           1
```

```
3  ------------
4  101101001101
```

Nikada nemojte da zaboravite da proverite resenje! Prebacite ovako dobijeno rešenje postupkom iznad nazad u dekadni zapis; trebalo bi da se dobije ponovo isti broj!
Sada kada smo se uverili da algoritam zaista radi, zapitajmo se **zašto** radi, tj. zašto baš 'obrćemo znak svakom bitu' i zašto na kraju dodajemo jedinicu.
Da ne znamo za ovaj algoritam, možda bi smo pomislili da naivni način može da nam reši problem: samo prebaciti vodeću cifru u jedinicu a ostatak broja pročitati normalno. Tada bi smo imali (radimo sa četvorobitnim brojevima):

- $1001_2 = -1_{10}$
- $1010_2 = -2_{10}$
- itd
  No ako malo bolje pogledamo nailazimo na dva ogromna problema. Prvi problem leži u tome da kada negativne brojeve sabiramo sa pozitivnim, efektivno se približavamo nuli, što ovde nije slučaj. Ako bih sabrao -1 sa 1 dobio bih -2 umesto 0:

```
1  1001
2  +  1
3  ----
4  1010 = -2
```

A drugi problem u tome što imamo dva načina da zapišemo broj $0 = 0000 = 1000$, što nam ne bi smetalo za račun, ali time smanjujemo opseg brojeva koje možemo da prestavimo sa odredjenom dužino bitova. Oba ova problema mogu da se otklone obrtanjem znakova i sabiranjem tog broja sa jedinicimo. Time efektivno krećemo da 'brojimo' negativne brojeve od pozadi:

- $1111_2 = -1_{10}$
- $1110_2 = -2_{10}$
- $\dots$
- $1000_2 = -8$
  Sada kada pogledamo bolje, vidimo da dodavanjem jedinice na svaki broj se tačno približavamo nuli.
  Pre nego što predjemo na realne brojeve primetimo činjenicu da ovakvim zapisom ćemo uvek imati jedan više negativan broj nego strogo pozitivnih, tj. interval neće biti simetričan oko nule. Za $n-$bitni broj naš opseg će biti $\left[\frac{-2^n}{2}, \frac{2^n}{2} - 1\right]$. Tako za četvorobitne brojeve imamo:
- $-8, -7, -6, -5, -4, -3, -2, -1, 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7 = [-8, 7]$

*Realni brojevi

Sada kada smo videli da negativne brojeve možemo da prikažemo isto kao i pozitivne uz dodatan bit koji predstavlja znak broja, prirodno se nameće pitanje da li i kako možemo da predstavimo realne brojeve. Realni brojevi se razlikuju od celih po tome što oni čine `kontinualni`, a ne `diskretni` skup.[19]Dakle čak i kada se ogardimo na neki konačan interval brojeva koje želimo da prikažemo tako što ćemo izabrati sa koliko bitova raspoložemo, opet imamo beskonačno brojeva koji spadaju u taj interval. Ovu činjenicu nećemo moći nikako da zaobidjemo pa ćemo uvek imati neki prividan skup realnih brojeva koje opisujemo, tj. nećemo moći da se približimo proizvoljno blizu zadatom broju.

Formiranje realnih brojeva može da se uradi na različite načine, a mi ćemo ovde izložiti najkorišćeniji - IEEE 754[20] interpretaciju realnih brojeva u pokretnom zarezu. Realni brojevi u pokretnom zarezu ovog standarda imaju naredni opšti oblik:

$$(-1)^s \cdot 1.m \times 2^{e+b}$$

, gde je:

Oznaka	Ime	Značenje	32 bit	64 bit
s	znak	znak	1	1
m	mantisa	preciznost	23	52
e	eksponent	ospeg	8	11
b	pomak	uvećanje	127	1023

Ovakav zapis se koristi jer ima niz pogodnosti koji vuče sa sobom:

- Svaki decimalni broj možemo da zapišemo u obliku $0.m \times 10^p$, pa tako na primer 125.77 možemo da zapišemo kao 0.12577×10^3 broj 30000 kao 0.3×10^5, jednocifrene brojeve kao $8 = 0.8 \times 10^1$ itd. U binarnom zapisu imamo to pogodnost sto svaki broj različit od nule počinje sa vodećim brojem 1, pa umesto $0.m$ kao kod decimalnih brojeva, imamo $1.m$.
- Pomak stoji tu da bi eksponent bio simetričan oko nule posto su nam potrebni i negativni eksponent za male vrednosti.

Pošto se od eskponent oduzima pomak, pravi eksponent kojeg nazivamo `normalizovani eksponent`, u oznaci E je vrednost takva da je zadovoljeno $E = e + b$, i on je u intervalu $[1, 254]$ za jednostruku, odnosno u intervalu $[1, 2046]$ za dvostruku preciznost.

Algoritam za prebacivanje realnih brojeva iz decimalnog u IEEE 754 normalizovani zapis je sledeći:

1. Konvervtovati apsolutnu vrednost celog dela broja na standardni način.

2. Decimalni deo posmatrati zasebno kao broj čiji je ceo deo nula.
3. Množiti tako posmatran broj dvojikom pamteći ceo deo dobijenog broja
4. Posmatrati samo decimalni deo novodobijenog broja i vratiti se na korak 3) sve dok se ne zapamti onoliko brojeva koliko ide u mantisu.
5. Formirati broj od zapamćenih brojeva čitajući ga od prvog ka poslednjem.
6. Zapisati decimalni broj kao binarni na isti način: ceo deo pa tačka pa broj dobijen koracima 2) - 5). Sada smo dobili denormalizovani zapis.
7. Pomeriti tačku onoliko mesta u levo ili desno tako da ispred zareza ostane samo jedna jedinica i zapamtiti taj broj. Time smo dobili mantisu.
8. Ukoliko smo tačku pomerali u levo, broj mesta koliko smo zarez pomerili sabrati sa pomakom, a ukoliko smo ga pomerila u desno, od pomaka oduzeti broj mesta koliko smo zarez pomerili i pretvoriti ga u binaran broj. Time smo dobili eksponent.
9. U prvi bit upisati odgovarajući znak, zatim upisati eksponent pa na to dodati onoliko brojeva mantise koliko je potrebno da se broj bitova poravna sa standardom koji koristimo.

Dakle, u praksi:

```
Pretvoriti broj -5.25 u IEEE 754 zapis sa jednostrukom preciznošću (32-bit)

(Korak 1) 5 -> 101

(Korak 2) Posmatramo 0.25
(Korak 3) 0.25 * 2 = 0.5 | pamtimo 0
(Korak 4) 0.5
```

```
(Korak 3) 0.5 * 2 = 1.0 | pamtimo 1
(Korak 4) 0.0
(Korak 3) 0.0 * 2 = 0.0 | pamtimo 0
(Korak 4) 0.0

(Korak 5) Ovde možemo da vidimo da će sve ostale cifre koje treba da zapamitmo da budu 0. Dakle naš broj je: 01000...0

(Korak 6) 101.010...
(Korak 7) m = 1.01010... | pomereno 2 mesta u levo
(Korak 8) 2 + 127 = 129 -> e = 10000001
(Korak 9) Znak je bio - tako da u znak se upisuje 1 -> s = 1
1 10000001 01010000000
s     e          m
```

```
Pretvoriti broj 0.1 U IEEE 754

(Korak 1) 0 -> 0
(Korak 2) Posmatramo 0.1
(Korak 3) 0.1 * 2 = 0.2 | pamtimo 0
(Korak 4) 0.2
(Korak 3) 0.2 * 2 = 0.4 | pamtimo 0
(Korak 4) 0.4
(Korak 3) 0.4 * 2 = 0.8 | pamtimo 0
(Korak 4) 0.8
(Korak 3) 0.8 * 2 = 1.6 | pamtimo 1
(Korak 4) 0.6
(Korak 3) 0.6 * 2 = 1.2 | pamtimo 1
(Korak 4) 0.2

Sada možemo da vidimo šablon pa imamo:
(Korak 5) 0001100110011...
(Korak 6) 0.0001100110011...
(Korak 7) m = 1.100110011... | pomereno 4 mesta u desno
```

```
(Korak 8) 127 - 4 = 123 -> e = 01111011
(Korak 9)
0 01111011 10011001100
```

Algoritam za konvertovanje brojeva iz IEEE 754 natrag u decimalan broj je jednostavan ali treba paziti na to da eksponenti mogu biti pozitivni i negativni. Pozitivni eksponenti će biti svi oni koji prilikom konvertovanje eksponenta nazad u decimalan zapis daju broj veći od pomaka, dok oni koji su manji od pomaka označavaju negativne eksponente. Pored toga, treba primetiti da brojevi u mantisi stoje nakon zareza, što znači da se oni čitaju na sledeći način:

$$m = c_{i_1} c_{i_2} \dots c_{i_{n-1}} c_{i_n}$$

$$m = c_{i_1} \cdot 2^{-1} + c_{i_2} \cdot 2^{-2} + \dots + c_{i_{n-1}} \cdot 2^{-(n-1)} + c_{i_n} \cdot 2^{-n} \qquad (**)$$

jer svako c_{i_k} stoji tačno k mesta udesno od zareza, a to se dobija tako sto 2 podignemo na negativnu vrednsot tag stepena. Naravno, pošto radimo sa binarnim zapisom, svako to c_{i_k} može biti samo 0 ili 1.

Dakle, algoritam bi glasio:

1. Zanemariti prvi bit i pročitati onoliko narednih bitova koliko je predvidjeno da čine eksponent i konvertovati ga u decimalan zapis.
2. Od tako dobijenog broja oduzeti pomak. Time dobijamo eksponent.
3. Ostatak bitova čitatati i kreirati od njih sumu na gore opisan način (**). Dobijen broj predstavlja decimalni deo.
4. Dobijen decimalni broj sabrati sa brojem 1 i pomnožiti ga sa dvojikom dignutom na dobijen eksponent. Time dobijamo krajnji

broj.

5. Gledajući prvi, odnosno vodeći bit, odrediti znak tom broju.

Uverimo se da ovaj algoritam radi tako što ćemo za prethodni primer odraditi obrnuti postupak:

```
Konvertovat i 0 01111011 100110011001 natrag u binarni broj.

(Korak 1) 01111011 -> 123
(Korak 2) 123 - 127 = -4 -> e = -4
(Korak 3) m = 1 * 2^(-1) + 0 * 2^(-2) + 0 * 2^(-3) + 1 * 2^(-4) + 1 * 2^(-5) + ...
          m = 0.60009765625
(Korak 4) 1.60009765625 * 2^(-4) = 0.1 ?
(Korak 5) 0 = + -> +0.1 ?
```

Znak pitanja ne stoji bez veze tu. Ako pogledamo bolje, imamo ovaj sitan ostatak 0.00009765625 što nam kvari računicu jer time dobijamo broj koje je ipak malo veći od 0.1. Ovo je jednostavno greška koja nastaje pri konverziji broja 0.1 iz decimalnog u binarni zapis. Kao što smo već rekli, nismo u stanju sve realne brojeve da pretvorimo u binarne što nam prouzrokuje da gubimo preciznost. Ovaj gubitak preciznosti se manifestuje na dva načina - više različitih realnih brojeva će se prikazati istim binarnim zapisom i pri konverziji možemo da dobijemo malo manji, odnosno malo veći broj nego što bi to trebalo, kao što je ovde slučaj. Ono što je zamiljivo je da u Javi na primer ako saberemo 0.1 + 0.1 dobijamo 0.2 što i očekujemo, ali ako saberemo 0.1 + 0.1 + 0.1 dobijamo broj 0.30000000000000004!

*Matematičke osnove - Iskazna logika

Već smo bili napomenuli da su upravo matematičari bili ti koji su doprineli procvatu računarstvu pokusavajući da odgonetnu rešenja nekih interesantnih hipoteza, te je potpuno prirodno da je jezik programiranja jedan jezik matematike zasnovan na principima dobrog definisanja stvari i strogog dokazivanja iskaza. Ova korelacija je bidirektna, kako se računarstvo razvijalo tako su se pojavljivale potrebe za novim matematičkim teorijama, no mi nećemo ovde pričati o teoriji kategorija, vodećoj teoriji koja se koristi da se opišu objekti koji najbolje modeliraju pojmove iz računarstva, već cemo se služiti samo osnovnim principima objašnjenim na intuitivnom nivou bez nekih strogih definicija.
Pre nego što opišemo neophodne matematičke pojmove, prvo moramo da se pozabavimo iskaznom i predikatskom logikom, što je osnova svake matematičke teorije.

Definicija 0.3 (Iskaz)**.**

Iskaz je bilo koja tvrdnja. Svaki iskaz je ili tačan ili netačan i nikad oba.

Iskaz je centralni objekat i može prestavljati bilo šta, od dela rečenice, punih rečenica, čitavog teksta. Svaki iskaz ima svoju istinitnu vrednost, o kojoj se (na neki način) dogovaramo ranije, i ta vrednost **se ne može menjati.** Primer iskaza su:

- Srbija je na planeti Zemlji
- 2 + 2 = 4
- Svi ljudi imaju plave oči
- $3 \mid 7$
- Danas je utorak
- Crvena je najlepša boja

- Spin je intrinzično svojstvo čestica
  Svaki od ovih iskaza ima svoju istinitnu vrednost. Srbija je na planeti Zemlji jeste jedan tačan iskaz, kao i da je $2+2=4$, dok iskazi poput svi ljudi imaju plave oči i iskaz da 3 deli 7 nisu tačni; za prvi iskaz imamo kontra primere dok drugi jednostavno nije tačan uvek. Iskaz da je danas utorak je nekada tačan a nekada nije $-$ sve u zavisnosti od toga kada se taj iskaz protumači, no u datom momentu taj iskaz će imati samo jednu i tačno jednu istinitnu vrednost. Da je crvena najlepša boja je subjektivni doživljaj koji će važiti za neke ljude dok za druge neće, te ovaj istinitna vrednost ovog iskaza zavisi od dogovora. Ali ovo ne bi trebalo da vas zbunjuje, ako malo bolje pogledamo i prethodni iskazi su tačni, odnosno netačni, samo po nekom dogovoru. Dogovoreno je da je ono što se podrazumeva kada se kaže reč 'Srbija' na onome što se podrazumeva kada se kaže izraz 'Planeta Zemlja'. Isto tako, dogovoreno je da je drugi iskaz tačan. Da smo zamenili značenje simbola 4 i 5, taj iskaz ne bi bio tačan (ili da znak `+` tretiramo kao `nadovezivanje`, tada bi $2+2=22$). Činjenica da 3 ne deli 7 je posledica nekih drugih činjenica. Poslednja rečenica je takodje podležna interpretaciji, doduše manje od svih ostalih.
  Upravo zbog ove nesigurnosti u interpretaciji od sada ćemo iskaze označavati, tj. obeležavati iskaznim slovima i to koristeći *mala* slova engleske abecede[21].

Definicija 0.4 (Iskazni jezik)**.**

Iskazni jezik čine sledeći simboli:

1. Iskazni veznici: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$
2. Skup iskaznih slova $P = \{p, q, r, s, t, \ldots\}$

3. Pomoćni simboli zagrada: ()

Definicija 0.5 (Iskazna formula)**.**

Iskazne formule se grade na sledeći način:

1. Konstante $\top$ (tačno) i $\bot$ (netačno)
2. Iskazno slovo je iskazna formula
3. Ako su P i Q neke iskazne formule, tada su i $\neg P$, $P \wedge Q$, $P \vee Q$, $P \Rightarrow Q$ i $P \Leftrightarrow Q$ iskazne formule
4. Svaka iskazna formula je dobijena primenom koraka 1. i 2. konačnim brojem puta

Prethodne dve definicije nam govore o tome šta čini iskazni jezik i kako se ofrmiraju iskazne formule, tj. rečenice tog jezika. Ukoliko iskaze "Danas je sunčano" i "Sada je 5 sati" označimo sikaznim slovima p i q redom, tada je naš skup iskaznih slova $P = \{p, q\}$ i neke od mogućih iskaznih formula su:

- $p \wedge q$ - "Danas je sunčano i sada je 5 sati", dobijenom primenom pravila 3. jedanput na skup P.
- $(p \wedge q) \vee \top$ - dobijena primenom pravila 3. dva puta, prvo da se kreira iskazna formula $p \wedge q$, a zatim korišcenjem pomoćnih simbola ta dobijena iskazna formula je uparena sa iskaznom formulom $\top$.
- $p \vee \neg p$ - "Danas je sunčano ili danas nije sunčano"
- $p \Rightarrow (q \Rightarrow (p \Rightarrow p))$ - "Ako je danas sunčano onda ako je sada 5 sati, danas je sunčano ako je danas sunčano"
- $p \Leftrightarrow p$ - "Danas je sunčano ako i samo ako[22] danas je sunčano"

- itd

Kao što već vidimo, naša iskazna slova kombinujemo sa drugim iskaznim slovima korišćenjem zadatih iskaznih veznika. Istinitost tako novodobijenog iskaza zavisiće u potpunosti od istinitosti iskorišćenih sikaznih slova i veznika koji koristimo. Pozabavimo se svakim od ovih veznika ponaosob. Radi lakše ilustracije, definišimo značenje nekih iskaznih slova koje ćemo koristiti kao primere za narednu sekciju:

- $p =$ "Posedujemo ličnu kartu"
- $q =$ "Posedujemo vozačku dozvolu"
- $r =$ "$\pi \in \mathbb{R}$"
- $s =$ "$1 < 0$"

1. Negacija $\neg$
   Operator, odnosno veznik, negacije, u oznaci $\neg$, prima jednu iskaznu formulu i 'obrće' joj znak - ukoliko je ta formula bila istinitna, onda ce biti neistinitna, a ukoliko je bila neistinitna, onda će postati istinitna. Primeri:

- $\neg p$ - "Ne posedujemo ličnu kartu" ili "Nije tačno da posedujemo ličnu kartu" ...
- $\neg(\neg(p))$ ili kraće $\neg\neg p$ - "Nije tačno da nije tačno da posedujemo ličnu kartu"
- $\neg(r \vee s)$ - "Nije tačno da je pi realan broj ili da je 1 manje od nula", odnosno drugim rečima "pi nije realana broj i 1 je veće ili jednako nuli"

p	$\neg p$
$\top$	$\bot$

p	$\neg p$
$\bot$	$\top$

2. Konjukcija $\wedge$
   Operator konjukcije, u oznaci $\wedge$, koji se čita kao veznik `i`, prima dve iskazne formule i dodeljuje istinitnu vrednost novodobijenoj formuli na sledeći načnin:

p	q	$p \wedge q$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$

Drugim rečima, ceo iskaz ce biti tačan ako i samo ako su oba konjukta tačna. Ovakvo ponašanje je potpuno prirodno i očekivano. Ukoliko na šalteru piše da je potrebno priložiti i ličnu kartu i vozačku dozvolu, šalterski radnik će nas uslužiti samo u slučaju kada priložimo oba dokumenta. Dovoljno je da zaboravimo da ponesemo jedan od ta dva dokumenta (ili oba) da šalterski službenik odbije da nas usluži.

3. Disjunkcija $\vee$
   Operator disjunkcije, u oznaci $\vee$, koji se čita kao veznik `ili`, prima dve iskazne formule i dodeljuje instinitnu vrednost novodobijenoj formuli na sledeći način:

p	q	$p \vee q$
$\top$	$\top$	$\top$

p	q	$p \vee q$
$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$

Nasuprot konjukciji, disjunkcija će biti ispunjena sve dok je jedan od oba disjunkta ispunjen. Ovakvo ponašanje **odstupa** od načina kako disjunkcija funkcioniše u svakodnevnom govoru. Pogledajmo naredni primer:
Osoba A nudi izbor "Želiš li negaziranu ili gaziranu vodu" osobi B koja vrši odabir. Osoba C zatim pita osobu B "Da li si izabrao negaziranu ili gaziranu vodu".
Potpuno prirodno, osoba C očekuje odgovor "gaziranu" ili odgovor "negaziranu" a nikako "i gaziranu i negaziranu". Dakle, u govornom jeziku na disjunktivna pitanja očekuje se isključujući odgovor, ali to nije slučaj u formalnoj logici! Formalno gledano, osoba B je u potpunom pravu da izabere obe opcije, jer osoba A očekuje bilo koji potvrdan odgovor od osobe B. Iako na prvi pogled ovaj uslov ispunjavanja u slučaju kada su oba disjunkta tačna se možda čini suvišnim ili nepotrebnom, uverijmo se da to nije slučaj sa primorm šalterskog radnika.
Ako na šalteru piše da je dovoljno priložiti ličnu kartu ili vozačku dozvolu, to znači da je dovoljno da priložimo bilo koji od ta dva dokumenta kako bi nas šalterski službenik uslužio, ali će nas svakako uslužiti i ako priložimo oba - sasvim prirodno. Da smo zahtevali ekskluzivnu disjkunkciju ovde, šalterski službenik ne bi smeo da nas uslužio u slučaju da priložimo oba dokumenta, što je sasvim absurdno.
No, čak i u programiranju nekada želimo da iskažemo i slučaj ekskluživne disjunkcije[23], koji se u formalnoj logici označava

simbolom $\veebar$ i čita kao `ekskluzivno ili` i koji se ponaša na sledeći način:

p	q	$p \veebar q$
$\top$	$\top$	$\bot$
$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$

4. Implikacija $\Rightarrow$
   Operator implikacije, u oznaci $\Rightarrow$, koji ce čita kao `ako ... onda ...`, je skraćeni zapis formule $\neg\vee$, i kao takav se ponaša na sledeći način:

p	q	$p \Rightarrow q$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$

Obratimo dobru pažnju na ovu tabelu jer njeno ponašanje može da deluje kontraintuitivno! Možemo videti da je cela implikacija netačna ako i samo ako je implikator p tačan a implikator q netačan! Zašto je ovo ovako? Hajde da razmotrimo to iz par uglova.

- Dolazak do zaključka
  Ako posmatramo na implikaciju kao da od pretpostavke p dolazimo nekim putem do zaključka q (što i jeste glavna

interpretacija implikacije), onda je jasno da ako od tačne pretpostavke stignemo do tačnog rešenja, da je ceo naš postupak ispravan. Npr: 2 je veće od 1, a 1 je veće od 0, dakle 2 je veće od 0.
Ukoliko od netačne pretpostavke stignemo do netačnog rešenja, opet je naš postupak validan; ništa nismo narušili. Npr rečenica "Sve patke su kopitari, dakle svaka guska je sisar" je tačna jer smo mi iz neke netačne premise stigli do netačnog zaključka.
Jasno je da ako od tačne premise stignemo do netačnog zaključka da smo negde napravili grešku. Npr ako kažemo da je zbir unutrašnjih uglova svako trougla 180 stepeni te da svaki ugao u troglu mora biti po 60 stepeni, vidimo da smo od tačne premise stigli do netačnog zaključka, jer smo zanemarili činjenicu da stranice trougla ne moraju biti sve iste dužine.
Najveći problem pretstavlja slučaj kada je p netačno, a q tačno. To znači da smo od netačne premise stigli do tačnog zaključka. Ali ni to ne bi trebalo da pretstavlja problem. Recimo da postavimo pitanje osobama koje nisu upoznati sa linearnom algebrom da li je tačno da svaki normalni linearni operator čuva normu vektora. Statistički gledano , oko pola ljudi bi odgovorilo tačno na to pitanje - da u opštem slučaju to ne mora da važi - prostim nagadjanjem. Iako do tačnog odogova nisu došli na ispravan način - pronalaženjem kontraprimera, tj. normalnog operatora koji ne čuva normu - ipak su odgovorili tačno, te se zato i evaluira ovaj slučaj kao tačan.

- Na praktičnom primeru
  Neka nam za ovu srvhu slovo p označava "Pada kiša", a oslov q "Teniski meč je otkazan". Tada rečenica "Ako pada kiša teniski meč je otkazan" bi stvarno trebalo da bude istinita za slučaj kada pada kiša i kada je teniski meč otkazan, trivijalno.

Ukoliko ne pada kiša a teniski meč nije otkazan, znamo da nešto nije u redu, pošto se po pravilu teniski meč tada otkazuje (ne računajuci mečeve u zatvorenim halama). Ako pak ne pada kiša, teniski meč idalje može biti otkazan iz nekog drugog razloga; teren nije spreman, igrač se razboleo ... I naravno, ako ne pada kisa onda teniski meč nije otkazan.

- Formalno gledano
  Već smo rekli da je $p \Rightarrow q$ samo kraći zapis za $\neg p \vee q$. Uverimo se da za iste vrednosti implikanata dobijamo iste vrednosti implikacije i u drugom zapisu:

p	$\neg p$	q	$\neg p \vee q$
$\top$	$\bot$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\top$

Implikant p se naziva *dovoljan uslov*, dok se implikant q naziva *potreban uslov*. To proizilazi iz činjenice da ako imamo implikaciju $p \Rightarrow q$ dovoljno je da znamo da je p tačan da bi zaključili da je i q tačan. Dok je potrebno da znamo da je q tačan da bi zaključili i da je p tačno (jer $\bot \Rightarrow \top$ je tačno).

Iako se nećemo baviti formalnim dokazivanjem na ovom kursu, vrlo je važno da vidimo dva pravila prirodne dedukcije za implikaciju, kako bi smo otklonili svaku mogućnost u pogrešnom zaključivanju u daljem toku kursa!
Ta dva pravila su *modus ponens* i *modus tollens* i oni izlgedaju:

$$\frac{p \Rightarrow q, \;\; p}{q}(MP) \qquad \frac{p \Rightarrow q, \;\; \neg q}{\neg p}(MT)$$

Prvo pravilo nam kaže kada imamo implikaciju, ako je dovoljan uslov ispunjen onda znamo da je ispunjen i potreban dok nam drugi uslov kaže kada imamo implikaciju, ako potreban uslov nije ispunjen onda nije ni dovoljan. Ovo možemo najlakše ilustrovati na praktičnom primeru:

- p = Marija je došla žurku
- $q =$ Ana je došla na žurku
- $p \Rightarrow q =$ Ako dodje Marija nažurku, sigurno će doći i Ana.
  Lako možemo da vidimo da ako znamo da je Marija došla na žurku onda znamo da je tu negde i Ana. Slično, ako znamo da Ana nije tu, nema svrhe tražiti ni Mariju.
  Ono što definitnvo ne možemo reći je da ako Marija nije tu neće ni Ana biti! Ana je samo rekla da ako Marija dolazi, ona sigurno dolazi, ali nije rekla ništa o tome da li će doći ako Marija nije tu. Možda joj se ipak ide i bez Marije. Dakle pravilo:

$$\frac{p \Rightarrow q, \quad \cancel{\neg p}}{\cancel{\neg q}}$$

  nikako ne smemo koristi! U tom slučaju jednostavno ne možemo reći nista od potrebnom uslovu. Da ovo pravilo ne važi, možemo lako videti i iz tabele; ukoliko p nije tačno, ceo iskaz $p \Rightarrow q$ je uvek tačan nezavisno od istinitnosne vrednosti od q.
  Potpuno analogno tome, ako na žurci zateknemo Anu, to ne znači da je tu negde i Marija! Kao što smo upravo rekli, možda se Ani dolazilo i bez nje! Dakle pravilo:

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

takodje nikako ne smemo koristiti! Isto, iz tabele možemo videti da kada znamo da je cela implikacija tačna i da je potreban uslov tačan, ne možemo sa sigurnošću ništa reći o istinitosti dovoljnog uslova, jer u oba slučaja je implikacija tačna.

5. Ekvivalencija $\Leftrightarrow$
   Operator ekvivalencije, u oznaci $\Leftrightarrow$, koji se čita `... ako i samo ako ...` , je skraćeni zapis formule $\Rightarrow \wedge \Leftarrow$ [24] i ponaša se kao:

p	q	$p \Leftrightarrow q$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\top$

Dakle, cela ekvivalencija je zadovoljena samo kada se isitinitnosne vrednosti obe formule poklapaju. Kod ovog operatora, oba iskazna slova su ujedno i *potreban* i *dovoljan uslov*. Iako ima ogromnu ulogu u matematici, ovaj operator se retko koristi u računarstvu, jer nam efektivno ne govori ništa novo.[25]

Definicija 0.6 (Tautologija)**.**

Tautologija je formula koja je tačna za sve istinitnosne vrednosti svih svojih iskaznih slova. Drugim rečima, formula koja je uvek tačna.

Zadatak 0.1.

Pokazati da su nardne formule tautologije:

- $p \Rightarrow p$
- $p \vee \neg p$
- $p \Leftrightarrow p$

Pre nego što nastavimo dalje obratimo još malo pažnju na to kako negacija funkcioniše.
Ukoliko želimo da negiramo neku formulu u celosti, to možemo da uradimo tako što negiramo svaku podformulu ponaosob i svaku konjukciju pretvorimo u disjunkciju a svaku disjunkciju u konjukciju. U to da negacija 'obrće' ova dva operatora možemo da se uverimo tabličnim putem:

p	q	$p \wedge q$	$\neg(p \wedge q)$	$\neg p$	$\neg q$	$\neg p \vee \neg q$
$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$	$\top$
$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$

p	q	$p \vee q$	$\neg(p \vee q)$	$\neg p$	$\neg q$	$\neg p \wedge \neg q$
$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\bot$
$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$

Pored ovog važnog pravila, važna pravila koje ćemo mi koristiti su pravila eliminacije i uvodjenja duple negacije. Prosto rečeno, kad god naeltimo na neku formulu oblika $\neg\neg p$, koja suštinski govori da "Nije tačno da ne posedujem ličnu kartu", možemo se osloboditi te

duple negacije i reći samo p, odnosno "Posedujem ličnu kartu". Slično tome, ukoliko imamo formulu oblika p, uvek joj možemo dopisati dva uzastopna znaka negacije ispred nje i transformisati je u $\neg\neg p$, mada ovo drugo pravilo se redje koristi, ako i uopšte.
Bez ulaženja u previše detalja, pravilo duple negacije nam ogućava i pravilo isključenja trećeg, koje kaže da uvek možemo da pretpostavimo da važi formula oblika $p \vee \neg p$, koja u našem primeru kaže "Posedujem ličnu kartu ili ne posedujem ličnu kartu".[26]

Zadatak 0.2.

Popuniti tablice za naredne izraze:

- $p \Rightarrow (q \Rightarrow p)$
- $(p \wedge q) \vee r$
- $\neg(\neg p \vee \neg q) \vee (\neg p \wedge q)$

*Matematičke osnove - predikatska logika

Do sada smo videli kako možemo da poistovetimo cele iskaze sa iskaznim slovima. Tako na primer, rečenice poput "Svaka čovek je smrtna" i "Aristotel je čovek" možemo da poistovetimo sa iskaznim slovima p i q redom. Sasvim je jasno da ako je svaki čovek smrtan i ako je Aristotel čovek, tada i za Aristotela mora da važi da je smrtan - nešto u šta smo se uverili i eksperimentalnim putem, jer otac logike više nije sa nama. No ovakav zaključak ne možemo da izvedemo koristeći se samo iskaznom logikom. Po pravilima iskazne logike, rečenica "Aristotel je smrtan" je potpuno nova rečenica koja zahteva potpuno novo iskazno slovo, npr. r, koje ne možemo nikako da zaključimo koristeći samo iskazna slova p i q. U

te svrhe uvodimo predikatsku logiku, odnosno logiku I reda, koja nam omogućava da radimo sa `promenjivima` i `predikatima`.

Definicija 0.7 (Predikat)**.**

Predikat predstavlja bilo koji iskaz koji govori o medjusobnoj relaciji promenjivih.

Definicija 0.8 (Predikatski jezik)**.**

Predikatski jezik čine sledeći simboli:

1. Logički simboli:
   - Veznici: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$
   - Znak jednakosti: $=$
   - Kvantifikatori: $\forall, \exists$
   - Skup promenjivih: $V = \{x, y, z, \ldots\}$
   - Pomoćni simboli zagrada: $()$
   - Konstante: $\top, \bot$
2. Predikatski simboli:

- Skup predikata: $\mathcal{P} = \{P, Q, R, S, T, \ldots\}$

Vrlo slično iskaznoj logici i u predikatskoj logici baratamo sa istim veznicima koji se ponašaju na isti način uz pridati znak jednakosti koji nema formulano interpretaciju i interpretira se uvek na isti način, intuitivno. Pored toga iskazna slova su sada zamenjena sa dva nova skupa i uvedena su dva nova simbola, koje nazivamo kvantifikatorima. Pre nego što predjemo na objašnjavanje kako oni

funkcionišu, radi kompletnosti pokažimo kako se formiraju predikatske formule:

Definicija 0.9 (Arnost)**.**

Dužina predikatskog simbola, koja se još naziva arnost predikatskog simbola, je tačan broj promenjivih koji taj predikat uzima u obzir. Takav broj je fiksan i ne može se menjati.

Primer 0.1.

- $P(x, y, z)$ - predikat P je arnosti 3 jer uzima u obzir 3 promenjive. Predikatu P možemo dodeliti i neke nove 3 promenjive, ali ih uvek mora biti tačno 3!
- $\leq (4, 5)$ - predikat manje jedanko je arnosti 2.
- $\text{BitiVisok}(x)$ - predikat BitiVisok je arnosti 1 i on uzima u obzir samo jednu promenjivu. Suštinski, on nam kaže da sta god je x, x mora biti nešto sto je visoko (npr. neka osoba x).
- $\text{Voli}(x, y)$ - predikat Voli je arnosti 2 i on može da označava da promenjiva x voli promenjivu y.

Predikate arnosti 1 nazivamo unarnim predikatima, predikate arnosti 2 nazivamo binarnim a arnosti 3 ternarnim predikatima.

Definicija 0.9 (Predikatska formula)**.**

Predikatske formule se grade na sledeći način:

1. Konstante $\top$ i $\bot$ su predikatske formule.
2. Ako je P neki predikat od arnosti n i $t_1, t_2, \ldots, t_n$ nekih n promenjivih, tada je $P(t_1, t_2, \ldots, t_n)$ predikatska formula.

3. Ako su φ i ϕ neke predikatse formule, tada su i $\neg\varphi, \varphi \wedge \phi, \varphi \vee \phi, \varphi \Rightarrow \phi, \varphi \Leftrightarrow \phi, \varphi = \phi$ predikatske formule.
4. Ako je φ predikatska formula, tada su i $\forall\varphi, \exists\varphi$ takodje predikatske formule.
5. Svaka predikatska formula je dobijena konačnom primenom pravila 1. - 4.

Vidimo da kod predikatske logike važe malo drugačija pravila oko notacije: promenjive su iskazane malim slovima engleske abecede uglavnom počevši od slova x, predikati su iskazani velikim slovima engleske abecede uglavnom počinjavši od P, dok su formule iskazane malim slovima grčkog alfabeta. Naravno, kao što se može videti u primerima gore, predikate možemo po potrebi i punim imenima da zovemo, jedino je važno da nemamo nikakvih razmaka, jer se razmak uvek tretira kao početak novog simbola, konstante itd. Takodje nekada, kao na primer sa simbolom $\leq$ umesto da pisemo $\leq (4, 5)$, tj. umesto da koristimo prefiksnu notaciju, koristimo infiksnu notaciju $4 \leq 5$.

Glavnu razliku izmedju predikatske logike i iskazne logike čine kvantifikatori $\forall$ i $\exists$ koji se čitaju `svaki` odnosno `za svaki/sve` i `postoji` redom. Definišimo ih i opišimo kako oni funkcionišu:

Definicija 0.10 (Kvantifikatori)**.**

Kvantifikatori su specijalni simboli predikatske logike koji se odnose na promenjive u zadatoj predikatskoj formuli i razlikujemo univerzalni kvantifikator u oznaci $\forall$ i egzistencionalni kvantifikator u oznaci $\exists$, koji su duali jedan drugome, tj. jedan se može dobiti negiranjem drugog kvantifikatora i negiranjem formule na koju se taj kvantifikator odnosi:

- $\forall\varphi \equiv \neg\exists\neg\varphi$
- $\exists\varphi \equiv \neg\forall\neg\varphi$

Primer 0.2.

- $\forall x \forall y P(x, y)$ - "za svako x i svako y vazi P od x y"
- $\forall x \exists y \text{Voli(y, x)}$ - "za svako x postoji y tako da y voli x"
- $\forall x \forall y \forall z (x \leq y \land y \leq z \Rightarrow x \leq z)$ - "za svako x, y i z, ako je x manje od y i y manje od z onda je i x manje od z" (tranzitivnost)

Formalno gledano, kvantifikatori se odnose samo na jednu promenjivu koju prati formula, tako na primer, najpravilniji oblik zapisa prve tačke iz gornjeg primera bi bilo $\forall x (\forall y (P(x, y)))$, no mi ćemo 'izvlačiti' sve kvantifkatore ispred zagrada i uparivati one koji stoje uz isti kvanitifakor uzastupno da dobijemo skraćeni zapis: $\forall x, y P(x, y)$.

1. Univerzalni kvantifikator $\forall$
   Univerzalni kvantifikator kojeg obeležavamo obrnutim velikim slovom A (eng. 'all') označava da svako pojavljivanje vezane promenjive za taj kvantifikator u zadatoj formuli može da se zameni bilo kojim drugim slovom alfabeta, odnosno jezika. Tako na primer za zadati skup promenjivih i jednočlani skup predikata arnosti 1 važi:

   $V = \{x, y, \text{Pera}, \text{stolica}\}$
   $\mathcal{P} = \{P\}$
   $\text{Ako važi } \forall t P(t) \text{ tada možemo da napišemo sledeće formule}$
   $P(x), \quad P(y), \quad P(\text{Pera}), \quad P(\text{stolica})$

dakle, svako pojavljivanje simbola t možemo da zamenimo sa *bilo kojom* promenjivom iz našeg skupa. S druge strane to ne znači da možemo da kažemo nešto poput $P(\text{Žika})$, jer Žika nije u skupu naših promenjivh. Drugačije rečeno, predikat P se evaluira kao tačan za **sve** promenjive iz skupa. Takodje, ukoliko u formuli imamo nevezanu promenjivu, npr nešto poput $\forall t Q(t, a)$, tada a nazivamo *nevezanom promenjivom* i ona se tretira kao konstanta i njeno pojavljivanje **ne smemo** zamenjivati drugim slovima. Iskažimo par rečenica koristeći predikate i univerzalni kvantifikator
posmatrajući skup svih ljudi kao naš skup promenjivih i konstanti:

- "Svako je sebi drugar" -> $\forall x(\text{Drugar}(x, x))$
- "Ako je jedna osoba drugar drugoj tada je i ta druga osoba drugar prvoj" -> $\forall x, y(\text{Drugar}(x, y) \land \text{Drugar}(y, x))$
- "Svako je drugar sa Petrom" -> $\forall x(\text{Drugar}(x, \text{Petar}))$
- "Sve što skače htelo bi da skače" -> $\forall x(\text{Skače}(x) \Rightarrow \text{HteloBiDaSkače}(x))$
- "Svaka osoba je smrtna" -> $\forall x(\text{Osoba}(x) \land \text{Smrtan}(x))$

2. Egzistencionalni kvantifikator $\exists$
Egzistencionalni kvantifikator kojeg obeležavamo velikim slovom E iz perspektive ogledala u ravni slova (eng. 'exists') označava da postoji (barem jedno ali možda i više ili čak sve!) promenjiva iz skupa svih promenjivih koju kada zamenimu u propratnu formulu dobijamo formulu koja se evaluaria kao tačna za zadatu promenjivu. Tako na primer za zadati skup promenjivih i jednočlani skup predikata arnosti 1 važi:

$V = \{\text{Nikola}, \text{Marko}, \text{Milica}, \text{Ana}\}$
$\mathcal{P} = \{\text{MuskiPol}\}$
Ako važi $\exists x(\text{MuskiPol}(x))$ tada znamo da barem jedna promenjiva iz
zadovoljava ovu formulu. U našem slučaju to su dve:
$\text{MuskiPol}(\text{Nikola}), \quad \text{MuskiPol}(\text{Marko})$

dakle, svako pojavljivanje simbola x možemo da zamenimo nekom odredjenom, ali barem jednom promenjivom iz našeg skupa. Kao što vidimo, egzistencijalni kvantifikator je više restriktujući od univerzalnog. Ista pravila za nevezane promenjive koje su važile za univerzalni, važe i za egzistencijalni kvantifikator. Par primera sa egzistencijalnim kvantifikatorm su:

- "Petar ima drugara" -> $\exists x.\,(\text{Drugar}(x, \text{Petar}))$
- "Ne postoji jednocifren broj koji nije jedan a koji deli broj 31" -> $\neg\exists x(x \in (1, 10) \land x|31)$
- "Postoje dve osobe koje su drugari" -> $\exists x, y(\text{Drugar}(\text{x}, \text{y}))$

Naravno ova dva kvantifikatora mogu i da se kombinuju, tako na primer, ako predikat $\text{Drugar}(x, y)$ interpretiramo kao "x je drugar y-nu", formula:

$$\forall x \exists y(\text{Drugar}(y, x))$$

kaže da svaka osoba ima (barem jednog) drugara. Ovde je vrlo važno da obratimo pažnju na redosled kvantifikatora!!! Formula

$$\exists y \forall x(\text{Drugar}(y, x))$$

kaže da postoji y koji je drugar svakome! Naravno, bitan je i redosled u samom predikatu:

$$\exists y \forall x(\text{Drugar}(x, y))$$

Ova formula kaže da postoji osoba kojoj je svako drugar. Znajući ovo, kako bi smo iskazali da postoji jedan broj, tako da kada se

sabere sa bilo kojim drugim brojem, taj drugi broj ostaje nepromenjen? (ovde se očigledno govori o broju 0)
S druge strane, dva ista uzastopna kvantifikatora mogu da komutiraju medjusobno, pa su formule:

$$\forall x \forall y (P(x, y))$$
$$\forall y \forall x (P(x, y))$$

identične i označavaju istu stvar. Isto i za egzistencijalni kvantifikator. Ove komutacije mogu da se dešavaju i u formulama sa više različitih kvantifikatora, dokle god se komutiranje vrše izmedju dva uzastopna ista kvantifikatora!

Već smo rekli da su univerzalni i egzistencijalni kvantifikatori duali jedni drugima. Pogledajmo još kako negacija utiče na njih. Ako bi smo želeli da iskažemo negaciju rečenice "Svi nose šešir", tj. zapisano matematički $\forall x. (S(x))$, to bi smo izveli tako što bi primenili prostu negaciju na ceo izraz: $\neg \forall x. (S(x))$ i dobili rečenicu "Nije tačno da svi nose šešir". Iako je ovo sasvim zadovoljavajuće sa matematičke strane, često u govoru, pa čak i u govoru o matematici, želimo da izbegnemo ovako čudne konstrukcije rečenica i da se osoblodimo ovog "Nije tačno ..." dela. To moramo učiniti vrlo oprezno i paziti da ne upadnemo u klopku i da kažemo da je ta rečenica ista kao i "Niko ne nosi šešir" ("Ne postoji osoba koja nosi šešir")! **OVO JE POGREŠAN ZAKLJUČAK I NIJE NUŽNO TAČAN!!!!!** Ovo što smo upravo iskazali je **suprotnost** a ne **negacija** našeg prvobitnog izraza. Naime, ako se prisetimo u kakvoj su vezi unvierzalni i egzistencijalni kvantifikator, svhatamo da negacija menja kvantifikator **uz negiranje propratne formule!** Dakle, ispravno je reći $\exists x. \neg (S(x))$, odnosno "Postoji osoba koja ne nosi šešir"!!! Ovim smo iskazali da postoji barem jedna osoba koja ne nosi šešir i nismo rekli ništa o broju osoba koje ne nose šešir. Možda stvarno niko ne nosi šešir, ali mi to jednostavno ne možemo

da zaključimo! Uverimo se na praktičnom primeru da ovakvo ponašanje je valjano:

Uzmimo rečenicu "Svako živo biće je revolviralo oko sunca barem jednom". Ovo očigledno nije tačno; novorodjenci rodjeni u istoj godini kada čitalac čita ovu skriptu nisu revolvirali celim putem oko sunca. Dakle, po zakonu isključenja trećeg, pošto ta rečenica nije tačna, njena negacija mora biti tačna. Da kažemo "Nijedno živo biće nije revolviralo oko sunca barem jednom" takodje nije tačno, jer osoba koja čita ovu skriptu je sigurno to uradila barem jednom, a sa velikom sigurnošću mogu da kažem da je to uradila i dvocifren broj puta, dakle ta rečenica ne može biti negacija prvoj, jer ni ona nije tačna. S druge strane, rečenica "Neko živo biće nije revolviralo oko sunca barem jednom" odnosno "Postoji živo bice koje nije revolviralo oko sunca barem jednom" jeste tačna rečenica. U to možemo da se uverimo tako što pronadjemo bilo koju pčelu jer njen životni vek ne prelazi 200 dana, te nijedna pčela nije u mogućnosti da revolvira oko sunca barem jednom i upravo je ovakva rečenica negacija početne.

U retkim slučajvima, negacija rečenice može da se poklapa i sa njenom suprotnošću. Reći da "Svaka osoba zna odgovor na $P \stackrel{?}{=} NP$[27]" pitanje je očigledno netačno. Njena negacija bi glasila "Postoji osoba koja ne zna odgovor na $P \stackrel{?}{=} NP$" što jeste tačno - važi za sve koji čitaju ovu knjigu[28], ali je takodje tačno i "Niko ne zna odgovor na $P \stackrel{?}{=} NP$", ali u opšte slučaju, negaciju nikako ne možemo da poistovetimo sa suprotnošću!

Zadatak 0.3.

Zadate rečenice pretvoriti u govorni/matematički oblik:

- Za svaka dva realna broja postoji broj koji je izmedju njih
- Svi koji misle, postoje.
- Svako ko daje ljubav dobija mnogo
- Za svaki broj postoji broj koji kada se sabere sa njime daje broj 0
- Svaka osoba ima mamu i ima tatu
- Svako voli sebe
- Postoji osoba koja ne voli ni jednu drugu osobu osim sebe
- $\forall x.\,(\text{Mačka}(x) \Rightarrow \text{Sisar}(\text{x}))$
- $\exists t.\,\forall x.\,\exists y.\,((x \neq y \wedge x \neq t \wedge \neg(x|t) \wedge y|t) \Rightarrow (y = t \vee y = 1))$
- $(\forall \varepsilon > 0)(\exists n_0 \in \mathbb{N})(\forall n \in \mathbb{N})(n \geq n_0 \Rightarrow |a_n - a| < \varepsilon)$

---

1. Reč algoritam dolazi od latinizovanog imena aprapskog matematičara Abu Džafar Muhamed Ibn Musa Al Horezmija koji je u IX veku dao veliki doprinos matematici svojim delom Hisab al džabr val mukabala (od koje potiče i reč algebra). Početkom XII veka, jedna Al Horezmijeva knjiga je prevedena na latinksi pod naslovom Algoritmi de numero indorum (Al Horezmi o indijskoj veštini računanja). ↩
2. David Hilbert (1862 - 1943) ↩
3. 1903 - 1995 ↩
4. 1912 - 1954 ↩
5. Prava definicija bi koristila množenje i rekurziju:
   $a^b \overset{def}{=} (a^0 = 1)|(a^{b+1} = a \cdot a^b)$ ↩
6. Odnosno kraće kao $k = \sum_{j=0}^{r} c_{i_j} \cdot n^j$ ↩
7. Iako smo za sistem sa osnovom (10) koristili grčki jezik, deka = 10 na grčkom, za sistem sa osnovom (2) koristimo latinski, bi = 2, a ne grčki, di = 2 ↩
8. Ostatak prilikom deljenja nekog broja n brojem p mogu biti samo brojevi $0, \ldots, p-1$, tako prilikom deljenja nekog broja brojem 2 ostatak mogu biti samo brojevi 0 i 1 ↩

9. Ovo je jako uprošćen model koji ne odgovara u potpunosti realnoj slici ↩
10. Iako Java nije popularan jezik za pisanje igrica ipak ima odredjen broj netrivijalnih igrica poput Minecraft-a koje su u potpunosti napisane u Javi ↩
11. Precizna definicija Jave preko njene paradigme nije jasno priznata. Prilikom navodenja paradigmi kojima jedan programski jezik pripada, često je dovoljno navesti samo dve-tri najvažnije, pa bi tako Javu okarakterisali kao imperativan, objektno orijentisan jezik visokog nivoa. ↩
12. NASM - Netwide Assembler ↩
13. Dennis Ritchie (1941 - 2011) ↩
14. James Gosling (1955 -) ↩
15. Jednom napisan, izvršava se svugde ↩
16. Zapravo ni Java nije pravi objektno-orijentisan programski jezik. Takvo zvanje se može dati Ruby programskom jeziku u kome nemaju primitivne promenjive, već je svaka promenjiva objektnog tipa. ↩
17. Formalno se ispostavlja da je lakše sve uvoditi bez tog znaka, ali za naše potrebe je i ovakva definicija sasvim dobra. ↩
18. Ovaj izraz potiče iz knjige "Kvaka 22" - Džosef Heler. ↩
19. Kada kažemo da je nešto kontinualni, podrazumevamo da za bilo koje dve 'tačke' toga možemo naći neku tačku izmedju njih. Realni brojevi su kontinualni jer za bilo koja dva broja, npr. 0.001 i 0.002 možemo naći broj izmedju njih, npr. broj 0.0015, dok kod diskretnih stvari to ne možemo. Tako je skup celih brojeva diskretan jer izmedju brojeva 2 i 3 ne postoji ni jedan broj (tog skupa). ↩
20. Institute of Electrical and Electronics Engineers ↩
21. Iskazna slova se uglavnom označavaju počevši od slova $p, q, r, s, t, \ldots$ (p - proposition - iskaz) ↩
22. Skraćeno, akko ↩
23. Zapravo, operator `xor` koji označava eksluzivnu disjunciju - eXclusive or - je operator koji se najčešće koristi u programiranju. Razlog toga se poteže iz raznih interesantnih svojstava koje taj operator ima. ↩
24. U ovo se uverite sami, na isti način kao i za impliaciju. ↩
25. U matematici je jako koristan upravo iz ovog razloga. Ukoliko znamo da su neka dva objekta ekvivalentna a sa jednim je mnogo lakše raditi nego

sa drugim, tada znamo da štagod novo zaključili za taj lakši objekat, iste stvari će važiti za objekat sa kojim je teže raditi. ↩

26. Dokaz da ovako nešto uvek možemo da iskažemo je sledeci: Dodatno pretpostavimo da nije tačno da važi pravilo isključenja trećeg. Uz to, ponovo dodatno pretpostavimo da važi neko iskazno slovo. Tada po zakonima disjunkcije važi i disjunkcija tog iskaznog slova sa svojom negacijom, što je u direktnoj kontradikciji sa početnom pretpostavkom. Tada možemo da zaključimo da ipak ne važi to dodatno pretpostavljeno iskazno slovo, ali tada važi njegova disjunkcija sa tim iskaznim slovom, što je ponovo u kontradikciji sa početnom pretpostavko, što znači da ne-ne važi zakon isključenja trećeg. Tada uz pravilo eliminacije duple negacije dolazimo do željenog zaključka. ↩
27. Jedno od milenijumskih otvorenih pitanja vezanih za računarstvo. Ovde se suštinski pita da li za svaki problem čija zadata rešenja mogu da se 'brzo' provere mogu da se isto tako 'brzo' i pronadju. ↩
28. Barem u toku pisanja ove skripte, 2025. godine ↩