UNIVERZITET METROPOLITAN BEOGRAD

KI102 - OSNOVE PROGRAMIRANJA

Osnove programskih jezika

Lekcija 02

PRIRUČNIK ZA STUDENTE

KI102 - OSNOVE PROGRAMIRANJA

Lekcija 02

OSNOVE PROGRAMSKIH JEZIKA

UVOD

U okviru ove lekcije naučićemo sledeće:

U okviru ove lekcije ćemo se upoznati sa:

- Pregledom programskih jezika
- Sinktaksom i semantikom programskih jezika
- Paradigmama programskih jezika
- Programskim jezicima niskog nivoa (low-level)
- Programskim jezicima visokog nivoa (high-level)
- Konceptima tradicionalnog programiranja

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Poglavlje 1

Pregled progamskih jezika

MAŠINSKI JEZIK

Mašinski jezik čini skup ugrađenih jednostavnih mašinskih instrukcija u binarnom obliku, razumljiv računaru, ali ne i ljudima.

Mašinski jezik čini skup ugrađenih jednostavnih mašinskih instrukcija. Ove instrukcije su u obliku binarnog koda. Računar razume samo ove mašinske instrukcije u obliku binarnog koda. Na primer, ako zahtevate od računara da vam sabere dva broja, morate napisati program u binarnom kodu, koji bi mogao da izgleda ovako:

1101101010011010

Proizvođači računara, odnosno, procesora, definišu mašinski jezik svog računara, odn. procesora. Ovo znači da binarni kod napisan za jedan računar, može da ne bude upotrebljiv kod drugog računara.

Da objasnimo ovo malo detaljnije. Program na mašinskom jeziku se sastoji od niza instrukcija koje su opisane sa po nekoliko bajtova koje govore procesoru koju operaciju treba izvršiti i sa kojim operandima.

Procesor čita instrukcije iz memorije i izvršava ih sekvencijalno.

Mašinska instrukcija se sastoji od operacionog koda (opcode) i operanada koji određuju nad kojim se podacima obavlja ta operacija (registri, memorijske lokacije). Broj operanada zavisi od operacionog koda.

Glavna memorija na određenom mestu (adresi) sadrži podatke u binarnom obliku, tj. vidu bajtova (slika 1).

Adresa	Sadržaj	Dešifriran sadržaj
1400	01000011	C
1401	01110010	r
1402	01100101	e
1403	01110111	w
1404	00000011	3

Slika 1.1 Prikaz sadržaja glavne memorije računara

U sledećem primeru imamo 2 operanda, jer instrukcija radi sa 2 operanda.

00001001	0000000001000000	0000000001000001
Opcode	Adresa 1	Adresa 2

Slika 1.2 Primer sabiranja dva broja datih na dve memorijske lokacije

Ovaj primer ilustruje sabiranje dva broja na adresama 128 i 129 na mašinskom jeziku Fon Nojmanove mašine. Operacioni kod (opcode) je 8-bitan, a adrese memorijskih lokacija 16-bitne.

Mana ovakvog načina zadavanja naredbi računaru je što je teško razumljiv ljudima, programiranje je komplikovano, a mogućnost pronalaženja greške veoma teška.

ASEMBLERSKI JEZIK

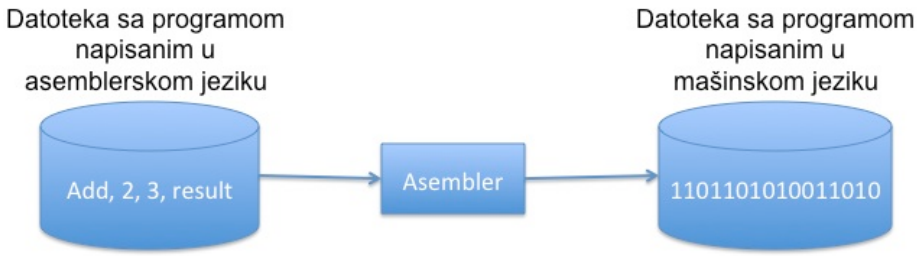
Asemblerski jezik koristi kratke reči (mneumonik) za nazive mašinskih instrukcija. Asembler prevodi asemblerski jezik u mašinske instrukcije

Primena mašinskog jezika je vrlo naporna. Programi pisani u mašinskom jeziku teško se čitaju i menjaju. Zato je stvoren asemblerski jezik još u ranim fazama računarstva. On koristi kratku reč (mneumonik) koja je karakteristična za naredbu mašinskog jezika. Na primer, mnenumotik koji se koristi za sabiranje je add a za oduzimanje koristi se mnenumotik sub. Sabiranje brojeva 2 i 3 se vrši na sledeći način:

add, 2, 3, result

Asemblerski jezika je razvijen da bi se olakšao posao programera. Umesto da koriste mašinski jezik računara, lakše programiraju korišćenjem asemblerskog jezika. Međutim, računar ne razume asembler jezik. Zbog toga, korišćenjem odgovarajućeg prevodioca – asemblera - asemblerski jezik se prevodi u mašinski jezik (slika 3).

Pisanje programa u asemblerskom jeziku je lakše nego pisanje programa u mašinskom jeziku. Međutim, ni u njemu nije lako pisati programe. Jedna instrukcija u asemblerskom jeziku odgovara jednoj mašinskoj instrukciji.



Slika 1.3 Asembler prevodi program pisan u asemblerskom jeziku u mašinski jezik

Da bi pisali program u asemblerskom jeziku, morate da znate kako CPU radi.

Asemblerski jezik se smatra jezikom niskog nivoa, jer je po prirodi blizak mašinskom jeziku i što je zavistan od mašine, tj. od računara.

PROGRAMSKI JEZICI TREĆE GENERACIJE

Programi treće generacije ne zavise od računara na kome se izvršavaju

U cilju da se izbegnu asembler jezici niskog nivoa, koji su zavisni od računara za koji su razvijeni, došlo je do razvoja treće generacije programskih jezika koji ne zavise od računara. Najpoznatiji programski jezici te treće generacije programskih jezika su FORTRAN (FORmula TRANslator), koji je razvijen za izradu matematičkih i inženjerskih programa, i COBOL (COmmon Business Oriented Language) koji je razvila američka mornarica za poslovne aplikacije.

Programski jezici treće generacije definišu skup primitiva (gradivnih elemenata jezika) višeg nivoa (složenijih) jer jedna takva primitiva zamenjuje više primitiva nižeg nivoa (jednostavnije). Zato je svaka primitiva višeg nivoa tako kreirana da može da zameni redosled (sekvencu) primitiva niskog nivoa koja je podržana mašinskim jezikom. Naš prethodni primer koji je bio napisan jezikom prve i druge generacije, sada se u programskom jeziku treće generacije može napisati na sledeći način:

assign TotalCost the value Price + ShippingCharge

Ova programska instrukcija višeg nivoa ne zavisi od računara koji treba da je izvrši. Program koji se naziva prevodilac (translator) prevodi program koji sadrži primitive višeg nivoa u program predstavljen mašinskim jezikom. Ovi programski prevodioci se često zovu i kompajlerima (compilers).

Pored kompajlera, postoje i programski prevodioci koji se nazivaju interpreterima (interpreters). Razlika je u tome što interpreteri, odmah po prevođenju instrukcije višeg nivoa u instrukcije izražene u mašinskom jeziku izvršavaju ove instrukcije, pre nego što počne da prevodi sledeću instrukciju višeg nivoa. Za razliku od kompajlera, koji prevedu program visokog nivoa u program pisan u mašinskom jeziku i koji se memoriše za kasniju i ponovljenu upotrebu, interpreter odmah izvršava svaku instrukciju programa višeg nivoa.

Primenom odgovarajućeg kompajlera, tj. kompajlera koji je razvijen za određen računar, aplikacioni program pisan u jeziku treće generacije, je mogao da se izvršava na bilo kom računaru, koji ima kompajler za programski jezik u kome je pisan aplikacioni program. Međutim, ta "nezavisnost" od računara nije uvek bila 100%. Kako se neke operacije na računarima različito izvode, a da bi oni brže radili posle primene prevedenog programa, dolazilo je i do prilagođavanja programskih jezika specifičnostima pojedinih mašina, pa je došlo i do ugrožavanja njihove nezavisnosti od računara. Da bi se smanjila upotreba ovih dijalekata programskih jezika američki institut za standarde (American National Standards Institute) i međunarodna organizacija za standardizaciju ISO (the International Organization for Standardization) standardizovali su važnije programske jezike.

NEDOSTACI TREĆE GENERACIJE RAČUNARA

Zbog specifičnosti razvijenih kompajlera za programe treće generacije, došlo je do ograničene njihove primene na različitim računarima.

Proizvođači kompajlera, da bi omogućili da oni brže prevode, ali, još značajnije, da prevedeni programi brže rade (optimizacija prevedenog koda), dodavali su pojedine programske module (proširenja jezika, engl. langauage extensions) u svojim kompajlerima koji su specifično razvijeni za pojedine računare, te je to ograničilo primenu programa pisanih u standardizovanim programskim jezicima samo na one računare za koje su oni razvili te

brže kompajlere. Ako programer u svoj aplikativni program ubaci i instrukcije koje koriste ta proširenja programskog jezika, onda tako napisan program neće moći da koriste kompajleri ostalih proizvođača računara, te će on moći da se primenjuje samo na kompajlerima proizvođača koji je uveo ta proširenja programskog jezika.
Naučnici i inženjeri koji su se bavili razvojem programskih jezika su počeli da razmišljaju o razvoju programskih jezika koji omogućavaju primenu apstraktnih koncepata, koji su potpuno nezavisni od kocepata koji su prilagođeni pojedinim računarima. Oni su istraživali mogućnost da sami računari (tj. njihovi operativni sistemi) pronalaze najpogodnije algoritme za realizaciju postavljenog koncepta, nego da samo izvršavaju programe koji primenjuju unapred definisane algoritme. Ovo je proizvelo razvoj novih generacija programskih jezika, koje je često teško generacijski klasifikovati.

PROGRAMSKI JEZICI NISKOG I VISOKOG NIVOA (VIDEO)

Objašnjava se evolucija od programskih jezika niskog nivoa do programskih jezika visokog nivoa

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

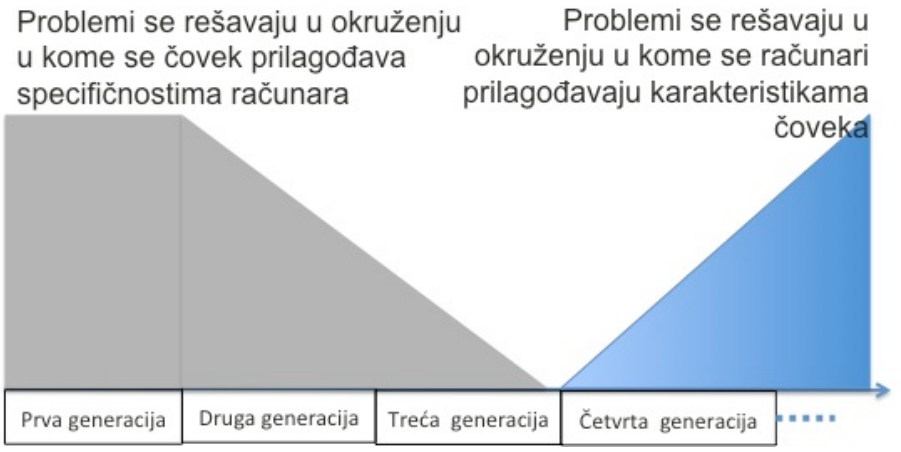
Poglavlje 2

Paradigme programskih jezika

PARADIGME PROGRAMIRANJA I NJIHOVA EVOLUCIJA

Paradigma programiranja je specifičan proces programiranja

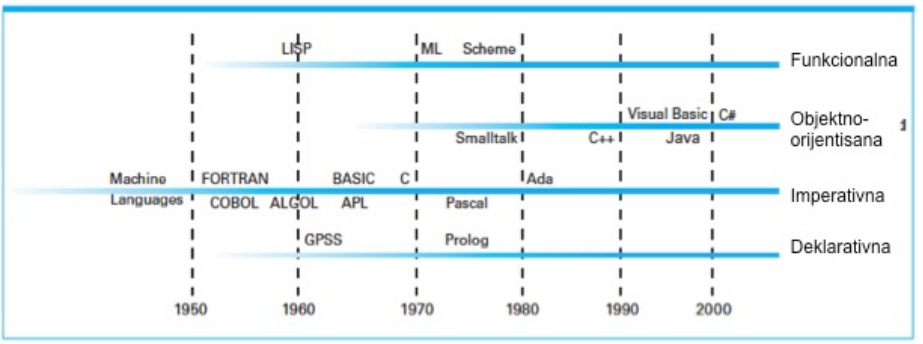
Programski jezici se mogu klasifikovati po generaciji kojoj pripadaju, a koja se može prikazati na vremenskoj osi (slika 1). Programski jezici više generacije dozvoljavaju veći stepen njihove nezavisnosti od specifičnosti računara na kojima se izvršavaju njihovi programi i postaju, u većem stepenu, orijentisani ka problemu koji rešavaju.



Slika 2.1 Generacije računara

Razvoj programskih jezika nije sledio linearni trend razvoja, već su različiti jezici imali svoje specifične putanje razvoja, podržavajući različite procese programiranja, koji se nazivaju programskim paradigmima. Slika 2 prikazuje ove različite putanje razvoja, zavisno od primenjene paradigme programiranja. Slika prikazuje četiri različite putanje razvoja, koje odgovaraju četiri paradigmi razvoja softvera:

- Funkcionalna paradigma
- Objektno-orijentisana paradigma
- Imperativna (ili proceduralna) paradigma, i
- Deklarativna paradigma.



Slika 2.2 Evolucija paradigmi programiranja

Kako ove paradigme programiranja utiču na način razvoja softvera, te obuhvataju sve faze razvoja jednog procesa, one su u stvari paradigme razvoja softvera, a ne samo paradigme programiranja.

IMPERATIVNA I DEKLARATIVNA PARADIGMA PROGRAMIRANJA

Imperativna paradigma rešava problem primenom odgovarajućeg algoritma i potrebnih podataka.

Imperativna (proceduralna) paradigma

Imperativna paradigma, poznata i kao proceduralna paradigma, predstavlja tradicionalni način razvoja softvera, tj, predstavlja tradicionalni proces programiranja. Imperativna (proceduralna) paradigma definiše proces programiranja po kome se komande definišu sekvencijalno, tj. po redosledu izvršavanja, i one manipulišu podacima radi dobijanja željenog rezultata. Proces programiranja se odvija u skladu sa odabranim algoritmom za rešavanje konkretnog problema. Na primer, ako želite da napravite program za rešavanje kvadratne jednačine, onda treba da koristite algoritam koji predstavlja poznati matematički iskaz koji daje dva moguća rešenja kvadratne jednačine. Alogoritam predstavlja proces koji čine poređane komande koje jedna za drugom izvršavaju neophodne operacije nad podacima, kako bi se dobio željeni rezultat.

Deklarativna paradigma

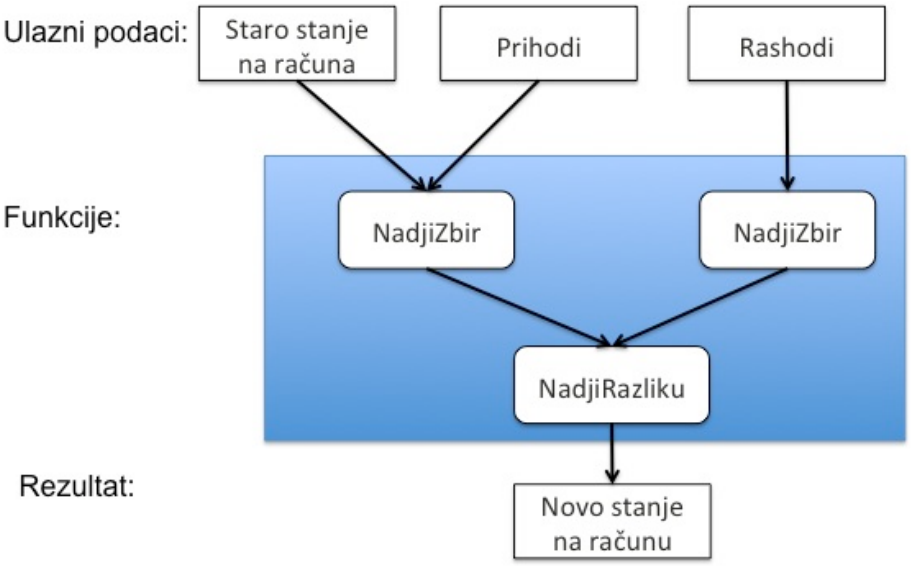
Deklarativna paradigma zahteva od programera da opiše problem koji ima, a ne algoritam za njegovo rešavanje. Sistemi koji koriste deklarativnu paradigmu ustvari primenjaju neki opšti, unapred odabran i postavljen algoritam rešavanja problema. U slučaju primene deklarativne paradigme, posao programera se svodi na precizno opsivanje problema koji se treba rešiti, a ne opisivanje algoritma za njegovo rešavanje. Zbog toga što ne postoje tako opšti procesi rešavanja problema koji se mogu primeniti na sve probleme, primena deklarativne paradigme je ograničena samo na specifične probleme određene kategorije, za koje postoji opšti postupak rešavanja. Na primer, na ovaj način se razvijaju programi za simulaciju različitih sistema (transporta, zaključivanja, političkih, ekonomskih i dr.). Njima se testiraju postavljene hipoteze ili utvrđuju predviđanja mogućih rešenja. Najčešće se računaju karakteristične

vrednosti parametara problema za različite vremenske trenutke. Poseban podstrek primene deklarativne paradigme je prouzrokovala primena formalne logike i matematike, kojim je definisan opšti postupak rešavanja neke klase problema ili svih problema. U tom pravcu se razvila i primena tzv. logičkog programiranja (logic programming).

FUNKCIONALNA PARADIGMA PROGRAMIRANJA

Funkcionalna paradigma dovodi do rezultata (funkcije) na osnovu unosa određenih podataka. Rezultat se dobija sekvencijalnom primenom niza jednostavnih funkcija, tj. programskih jedinica.

Po funkcionalnoj paradigmi, proces rešavanja problema se svodi na unos potrebnih podataka i dobijanje proizvedenih rezultata. Entitet koji proizvodi rezultat kada mu se unesu određeni podaci se u matematici zove funkcija, te je tako ova pradigma i dobila naziv (funkcionalna paradigma). Po ovoj paradigmi, program čine manje, prethodno definisane programske jedinice (pripremljene za određene funkcije), tako da izlazni rezultat iz jedne programske jedinice, je ulazni podatak u drugoj programskoj jedinici. Pri tome, obezbeđuje se da početni unos podataka, daje na kraju željeni razultat. Ustvari, primena funkcionalne paradigme se svodi na primenu niza povezanih jednostavnih funkcija koje zajedno, daju ukupno, krajnji rezultat koji odgovara postavljenoj složenoj funkciji. Na slici 3 je dat primer rešavanja problema uravnotežavanja vaših prihoda i rashoda.



Slika 2.3 Primer primene funkcionalne paradigme

OBJEKTNO-ORIJENTISANA PARADIGMA PROGRAMIRANJA

Objektno-orijentisana paradigma posmatra softverski sistem kao skup objekata, sposobnih da sprovode definisane akcije, a njihovom interkacijom, objekti rešavaju ukupni problem.

Objektno-orijentisana paradigma primenjuje proces programiranja koji se najčešće naziva objektno-orijetisanim programiranjem. Softverski sistem, po ovoj paradigmi, čini skup (kolekcija) jedinica, koje se nazivaju objektima. Svaki od objekata je uvek spreman da izvede akcije nad sobom, ili da zahteva akcije koje bi izvršili drugi objekti. Međusobnom interakcijom, ovi objekti rešavaju ukupni problem programske aplikacije. Na primer, garfički korisnički iterfejsi programskih sistema (na primer, MS Word) imaju niz ikonica. Svaka ikonica predstavlja neki softverski objekat. Kada kliknete neku ikonicu, ona reaguje nekom akcijom, jer ste aktivirali određeni skup procedura (koje se nazivaju metodima). Ovi metodi opisuju reakciju objekta na razne događaje na koje objekat reaguje. Ceo objektno-orijentisani sistem čini niz objekata, koji neki eksterni događaj (na primer, klik neke ikonice korisničkog interfejsa) pokreće talas događaja, tako što aktivirani metodi nekih objekata, proizvode događaje koji aktiviraju metode drugih objekata, i tako redom, ceo sistem proizvodi željenu reakciju usklađenu interakcijom pojedinih njegovih objekata koji razmenjuju poruke koje aktiviraju pojedine metode koji su sadržani u objektima. Jedan objekat čine podaci i metodi koji manipilišu tim podacima. Svaki metod koristi odgovarajući algoritam za tu manipulaciju podacima. Drugi objekti koji žele da manipulišu podacima drugog objekta, koriste metode tog objekta za njihovu manipulaciju, umesto da imaju svoje metode za manipulaciju podacima drugih objekata.

Na ovaj način, programer ima manje posla, jer se jedan metod koristi u mnogim i različitim slučajevima. Jedna klasa definiše i opisuje svojstva (podatke i metode) objekata koje klasa predstavlja. Klasa u svakom trenutku može da stvara objekte za koje ti objekti koje stvara jedna klasa imaju iste karakteristike, iako predstavljaju različite entitete. Najčešće imaju različite vrednosti podataka. Na primer klasa Student kreira objekte koji predstavljaju konkretne studente, sa konkretnim imenima i prezimenima, kao njihovim specifičnim podacima. Ako se u podacima nalazi i datum rođenja studenta, onda se može definisati metod koji u trenutku svog aktiviranja, može da izračuna godine, mesece i dane života studenta.
Programi razvijeni u skladu sa objektno-orijentisanom paradigmom (na primer, C++), vrlo često koriste i deo jezika koji primenjuje imperativnu paradigmu (na primer C). Na taj način, metodi koji se koriste u objektno-orijentisanim sistemima su jedinice koje primenjuju imperativnu paradigmu u tim programskim jedinicama. Objektno-orijentisani programski jezici Java i C# su derivati programskog jezika C++. Zbog toga, i oni od njega nasleđuju to imperativno jezgro, jer i njihovi metodi primenjuju imperativnu (proceduralnu) paradigmu.

Poglavlje 3

Programski jezici višeg nivoa

OPŠTA SVOJSTVA PROGRAMSKIH JEZIKA

Programski jezik mora da zadovolji zahteve univerzalnosti, primenljivosti i efikasnosti

Svaki programski jezik je sredstvo rada, i kao takav - treba da se precizno projektuje. Neke programske jezike su napravili pojedinci (C++), neke male grupe ljudi (C, Java), a neke su pravile velike grupe (ADA). Da bi programski jezik bio zaista programski jezik, on mora da zadovolji određene zahteve.

Programski jezik mora biti univerzalan

To znači da svaki problem mora imati rešenje koje se može isprogramirati u jeziku; naravno, ako je problem uopšte rešiv na računaru. Ovo možda izgleda kao strog zahtev, ali ga čak i mali programski jezici mogu ispuniti. Svaki jezik koji može definisati rekurzivne funkcije, univerzalan je. Sa druge strane, jezik koji nema ni rekurziju ni iteraciju, ne može se smatrati univerzalnim. Programski jezik može imati oblast za koju je namenjen. Na primer, programski jezik čiji su jedini tipovi podataka brojevi i nizovi, može se koristiti za rešavanje numeričkih problema, ali verovatno neće biti pogodan za primenu u veštačkoj inteligenciji.

Mora da postoji mogućnost implementacije programskog jezika na računaru

To znači da mora postojati mogućnost realizacije svakog dobro napisanog programa. Matematička notacija se ne može u potpunosti primeniti, pošto je pomoću te notacije moguće formulisati probleme koje nije moguće rešiti na računaru. Jezici kojima ljudi govore se takođe ne mogu primeniti, pošto su neprecizni i dvosmisleni.

Programski jezik u praksi treba da ima prihvatljivo efikasnu implementaciju, tj. primenu u računaru

Postoji mnogo različitih mišljenja o tome šta je prihvatljiva efikasnost. Programeri koji koriste FORTRAN, C ili PASCAL mogu očekivati da njihovi programi budu skoro isto tako efikasni kao i programi pisani u asembleru. Programeri koji koriste PROLOG moraju prihvatiti manju efikasnost, ali se to nadoknađuje upotrebom jezika koji je pogodan za određenu oblast.

PROGRAMSKA INSTRUKCIJA

Jedna naredba napisana u jeziku višeg nivoa odgovara skupu jednostavnih mašinskih instrukcija

S obzirom na to da je pisanje i održavanje programa na mašinskom jeziku komplikovano pogotovo ako se radi o većim i komplikovanim programima, programi se pišu u jezicima višeg nivoa kao što su Java, C/C++, Perl, itd.

Jedna instrukcija u jeziku višeg nivoa obavlja zadatak koji je dosta komplikovaniji u odnosu na mašinsku instrukciju. Jedna naredba napisana u jeziku višeg nivoa odgovara skupu jednostavnih mašinskih instrukcija. Primer:

```
double pi = 3.14156;
```

Ova jednostavna instrukcija napisana u Javi ili C/C++ jeziku prevodi se na skup mašinskih instrukcija koje:

- rezervišu potrebnu memoriju
- smeštaju broj na odgovarajuću memorijsku lokaciju
- beleže tu lokaciju za kasniju upotrebu

Svi ovi detalji su sakriveni od programera, a ovakav način pisanja je jednostavniji i kraći. Program napisan na jeziku visokog nivoa sastoji se od sekvenci naredbi smeštenih u tekstualnom fajlu. Tako napisan program se zove izvorni kod (engl. source code).

Primer:

```
public class MojPrviJavaProgram {
  public static void main(String[] args) {
    System.out.println("Zdravo, svete!");
  }
}
```

Iz primera se vidi da je program napisan na jeziku višeg nivoa razumljiviji čoveku i da se sastoji od iskaza formiranih od reči jezika kojim ljudi govore. Ovako napisani izvorni kod se smešta u tekstualni fajl sa imenom ***MojPrviJavaProgram.java***. Pošto se izvorni kod sastoji od regularnog teksta, pisanje i editovanje koda je moguće raditi iz jednostavnog programa za editovanje teksta.

Procesor računara ne može da izvršava ovakav program pošto procesor razume samo mašinske instrukcije. Da bi ovako napisani kod mogao da se izvršava na računaru, potrebno je izvorni kod pretvoriti u ono što procesor razume, mašinski kod. Poseban program koji se zove prevodilac (engl.compiler) zadužen je za prevođenje koda. Kao ulaz, prevodilac uzima tekstualne datoteke sa izvornim kodom i od njih pravi nove datoteke koje sadrže instrukcije u binarnom obliku. Kada se program prevede, on postaje razumljiv računaru. Tako dobijeni izvršni kod se učitava u memoriju računara, odakle procesor čita instrukcije i izvršava ih.

SINTAKSA, SEMANTIKA I PRAGMATIKA PROGRAMSKOG JEZIKA

Sintaksa programskog jezika se odnosi na formu programa, a semantika programskog jezika se odnosi na značenje.

Svaki programski jezik ima svoju sintaksu i semantiku. Jezici koje ljudi koriste u svakodnevnom životu, takođe imaju sintaksu i semantiku. Kod programskih jezika postoji i pragmatika, koja je jedinstvena za njih.

Sintaksa programskog jezika se odnosi na formu programa. Ona prezentuje način na koji se moraju urediti izrazi, komande, deklaracije i ostale programske konstrukcije - da bi se dobio ispravan program. Sintaksa određuje formu po kojoj programeri pišu programe, drugi programeri čitaju i računar obrađuje.

Semantika programskog jezika se odnosi na značenje programa. Ona ukazuje na to kako se od ispravnog programa može očekivati da se ponaša. Semantika određuje način na koji programer komponuje programe, kako ih drugi programeri razumeju i kako ih računar interpretira.

Pragmatika programskog jezika se odnosi na način na koji se jezik namerava da koristi u praksi. Pragmatika utiče na to kako programeri očekuju da se programi projektuju i implementiraju u praksi.

Sintaksa je važna, ali su semantika i pragmatika još važnije. Ovo se može jasnije sagledati na primeru jednog eksperta koji razmišlja o rešenju datog problema.

Programer prvo dekomponuje problem i identifikuje odgovarajuće programske jedinice (procedure, pakete, apstraktne tipove ili klase).

Posle toga, on razmatra odgovarajuću primenu svake od programskih jedinica, određuje tipove podataka i promenljivih koji postoje u jeziku, definiše kontrolne strukture, izuzetke itd.

Tek na kraju programer kodira program, tj. piše program u sintaksi određenog programskog jezika. Samo u ovoj završnoj fazi sintaksa programskog jezika postaje bitna. Vrlo je važno da seprimena sintakse određenog programa izvrši što kasnije u procesu razvoja nekog programa, jer svi nedostaci nekog programa nisu posledica sintakse (sem u slučaju elementarnih grešaka u njenoj primeni, što se lako otklanja), već u pogrešno postavljenoj semantici.

Zato je projektovanje nekog programskog sistema mnogo značajnije nego čisto programiranje tj. kodiranje, kada se primenjuje sintaksa određenog programskog jezika da bi se semantika programa pretvorila u formu (sintaksu) koji prevodilac programa (interpreter ili kompajler) razume i koji prevodi u izvršni, mašinski program.

ISTORIJSKI PREGLED RAZVOJA PROGRAMSKIH JEZIKA

Postoji više generacija programskih jezika višeg nivoa

Piter Vegner je dao jednu klasifikaciju programskih jezika višeg nivoa, koju ćemo i mi ovde koristiti. Kao kriterijum su korišćene karakteristike jezika koje su se po prvi put pojavile. Tako je **prva generacija programskih jezika** nastala u periodu od 1954. do 1958. godine. To su bili jezici:

FORTRAN I, ALGOL 58, Flowmatic, i IPL 5

i svi su podržavali matematičke izraze.

Jezici druge generacije su se koristili tokom perioda od 1959. do 1961. godine. To su bili jezici:

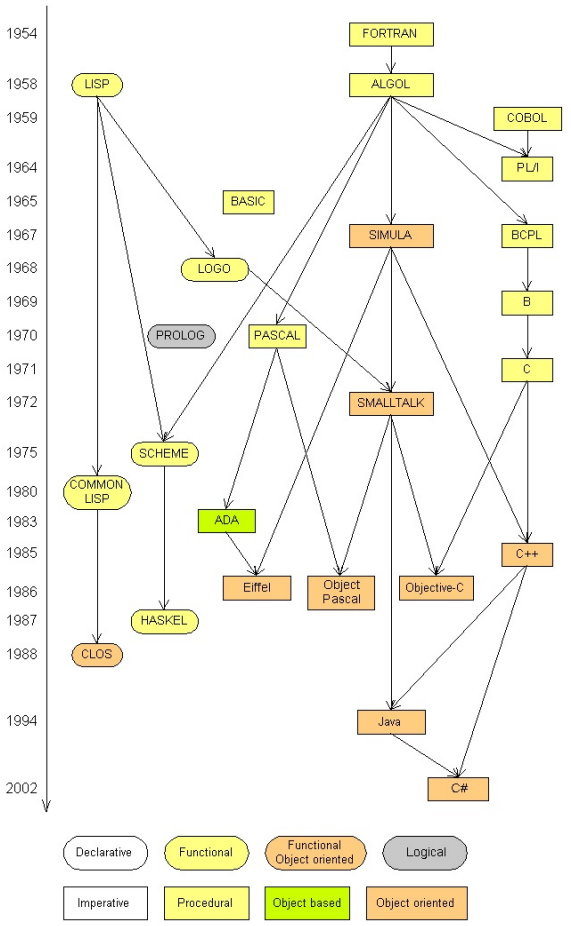
- FORTRAN II - podržavao potprograme, odvojeno kompajliranje
- ALGOL 60 - blokovska struktura, tipovi podataka
- COBOL - opis podataka, rukovanje datotekama
- Lisp - obrada listi, pokazivači

Jezici treće generacije su se koristili tokom perioda od 1962. do 1970. godine. To su bili jezici:

- PL/1 FORTRAN + ALGOL + COBOL
- ALGOL 68
- Pascal
- Simula

Period od 1970. do 1980. beleži zastoj u razvoju programskih jezika. Nastalo je mnogo novih jezika, ali je malo njih zaista zaživelo. Tako su tada nastali, ali i opstali jezici kao što je Smalltalk (naslednik jezika Simula), Ada (naslednik jezika ALGOL 68 i Pascal) ili C++ (izveden iz jezika C i Simula).

Jezici prve generacije su se uglavnom koristili u naučnim i inženjerskim oblastima, tako da su uglavnom i rešavani matematički problemi. Jezici kao što je na primer bio FORTRAN I su omogućavali programerima da pišu matematičke formule oslobađajući ih potrebe da poznaju mašinski jezik.



Slika 3.1 Prikaz istorijskog razvoja programskih jezika

ŠTA JE JAVA?

Java nije samo objektno-orijentisani programski jezik. Java je i tehnologija za razvoj objektno-orijentisanih programa

Programski jezik Java razvio je tim stručnjaka firme Sun Microsystems početkom 90-tih godina. Kreiran je s ciljem da omogući programerima da jednom napisan program mogu da izvrše na bilo kojoj mašini, tj. računaru.

Java je jednostavan, objektno-orijentisan, distribuiran, interpretatorski, robusan, bezbedan, arhitektonsko neutralni, prenosiv, visoko performansni, više nitni i dinamički programski jezik.

Java je u početku kreiran za razvoj veb alikacija. Danas je to jezik koji se koristi za samostalne aplikacije koje se mogu izvršavati na različitim platformama (računarima i operativnim sistemima) na serverima, sitnim računarima, i mobilnim uređajima. Danas se Java često koristi za razvoj aplikacija instaliranih na veb serverima. Ove aplikacije obrađuju podatke, vrše obračune, i generišu veb stranice.

Tzv. Java standard definiše specifikaciju Java programskog jezika i Java API (Application Programming Interface).

Specifikacija Java programskog jezika daje tehničku definiciju sintakse i semantike Java jezika i može se naći na: http://docs.oracle.com/javase/specs/

Interfejs programske aplikacije (API), poznat i kao biblioteka, sadrži unapred definisane klase i interfejse koji se mogu koristiti pri razvoju programa pisanih u Javi. Ova biblioteka je u stalnom razvoji čime se proširuju i poboljšavaju mogućnosti razvoja softverskih sistema u Javi. Java API se može naći na: http://download.java.net/jdk8/docs/api/

Java se nudi u tri posebna izdanja (edicije):

Java Standard Edition (Java SE) za razvoj klijentskog dela aplikacije. Razvijen program može da se izvršava samostalno ili kao aplet u okviru veb pregledača (Web browser)..

Java Enterprise Edition (Java EE) za razvoj serverskog dela aplikacija, kao što su Java servlets, JavaServer Pages (JSP), i JavaServer Faces (JSF)

Java Micro Edition (Java ME) za razvoj aplikacija za mobilne uređaje, kao što su mobilni telefoni.

Oracle povremeno objavljuje nova izdanja Jave. Poslednje izdanje je **Java 8**. Oracle objavljuje nove verzije Jave u obliku **Java Development Toolkit. (JDK).** Sada je aktuelan **JDK 8.**

Poglavlje 4

Koncepti tradicionalnog programiranja

TIPOVI INSTRUKCIJA U PROGRAMSKOM JEZIKU

Programski jezici zahtevaju da mi koristimo izvesne konstrolne strukture kako bi izrazili naš algoritam korišćenjem koda

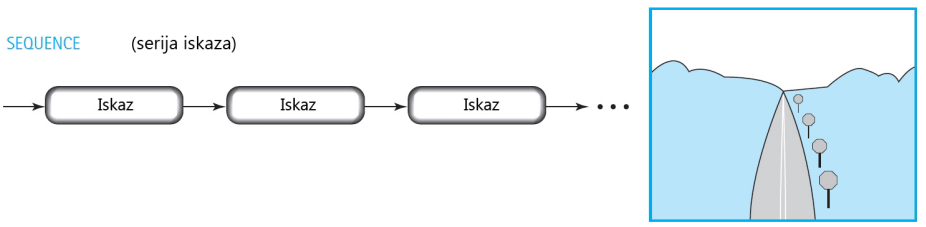
Instrukcije programskog jezika odražavaju operacije koje računar može da izvrši:

- Računar može da prenese podatke sa jednog mesta na drugo.
- Računar može da dobije podatke sa ulaznog uređaja (na primer, tastatura i miš) i da zapiše podatke na izlazni uređaj (ekran, na primer).
- Računar može da skladišti podatke ili da ih učita iz svoje memorije (RAM) ili sekundarnog skladišta (Hard disk, CD-ROM, itd).
- Računar može da izvrši poređenje podataka za operacije jednakosti i nejednakosti, i da donese odluku u zavisnosti od rešenja.
- Računar može da izvrši aritmetičke operacije (sabiranje i oduzimanje, na primer) veoma brzo.
- Računar može da izvrši grananja za različite tipove instrukcija.

Programski jezici zahtevaju da mi koristimo izvesne kontrolne strukture (control structures) kako bi izrazili naš algoritam korišćenjem izvornog koda (source code).

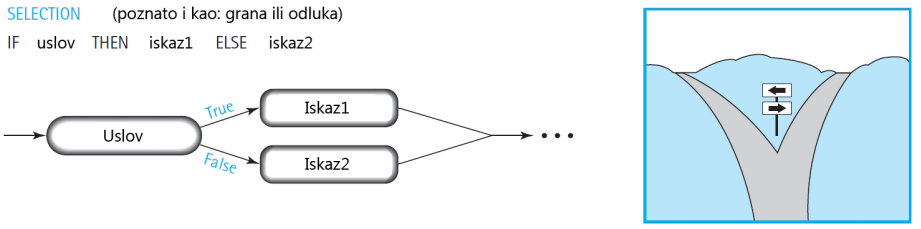
Postoje četiri različita načina struktuiranja iskaza (instrukcija) u najvećem broju programskih jezika: sekvenca, izbor (selekcija), petlja i potprogram.

Sekvenca (sequence) je serija iskaza koji se izvršavaju jedan za drugim, Slika-1.



Slika 4.1 Sekvenca

Selekcija (selection), uslovna kontrola struktura, izvršava različite iskaze u zavisnosti od određenog uslova, Slika-2.

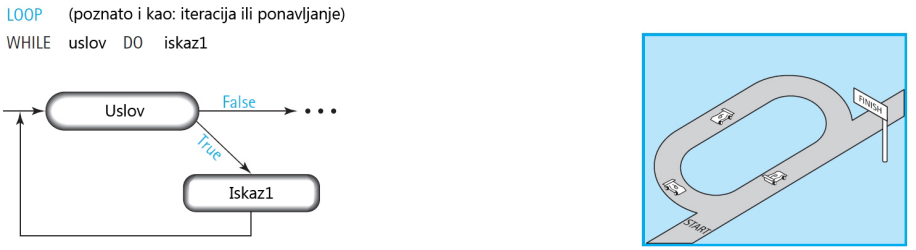


Slika 4.2 Selekcija

PETLJE I POTPROGRAMI

Petlja (loop), ponavlja skup iskaza sve dok se ne zadovolji neki uslov. Potprogram je imenovana sekvenca instrukcija napisana odvojeno od glavnog programa

Petlja (loop), ponavljajuća kontrolna struktura, ponavlja skup iskaza sve dok se ne zadovolji neki uslov, Slika-3.

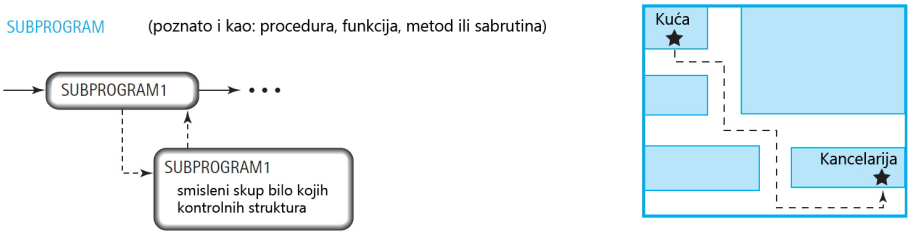


Slika 4.3 Petlja u programskom jeziku

Primer. Pretpostavimo da vozite automobil. Putujući nekim putem je isto kao da pratite sekvencu instrukcija. Kada naiđete na račvanje na putu, morate da odučite kojim ćete putem ići i onda jednostavno krenete jednim ili drugim putem. Ovo je upravo ono što računar radi kada se susretne sa kontrolnom strukturom selekcije (ponekad se naziva grana ili odluka) u programu.

Ponekad morate da idete oko bloka više puta pre nego što nađete mesto za parkiranje. Računar radi sličnu stvar kada se susretne sa petljom (loop) u programu.

Potprogram (subprogram) nam dozvoljava da struktuiramo naš kod tako što ga iscepkamo na manje jedinice koda, Slika-4.



Slika 4.4 Podprogram

Potprogram (subprogram) je imenovana sekvenca instrukcija napisana odvojeno od glavnog programa. Kada program izvršava instrukciju koja se odnosi na neko ime ili podprogram, ustvari se izvršava kod. Kada se završi rad potprograma, nastavlja se izvršavanje glavnog programa od prve sledeće instrukcije.

Primer. Pretpostavimo da svakoga dana idete do svoje kancelarije, na posao. Uputstvo za odlazak od kuće do posla formira proceduru koja se zove *"Go to the office"*. Stoga ima smisla, da vam neko da sledeće uputstvo *"Go to the office, then go four blocks west"*, bez navođenja svakog pojedinačnog koraka koje treba da izvršite da bi ste otišli do posla. Podprogrami nam dozvoljavaju da napišemo delove našeg programa odvojeno, a da ih onda spojimo u konačnu formu. Oni mogu u velikoj meri da uproste zadatak pisanja velikih programa.

TRI KATEGORIJE ISKAZA U PROGRAMU

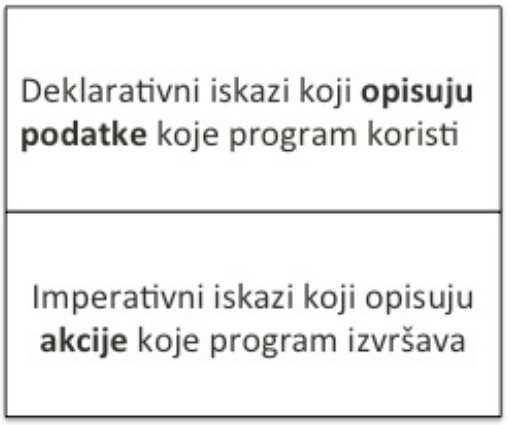
Program predstavlja skup iskaza koji mogu biti: deklarativni iskazi, imperativni iskazi i komentari

Ovde ćemo prikazati primenu koncepta tradicionalnog programiranja (imperativnog i objektno-orijentisanog) na primeru sa različitim programskim jezicima:
C – imperativni (proceduralni) programski jezik treće generacije
C++ - objektno-orijentisani jezik koji nastao proširenjem C jezika
Java i C# - objektno-orijentisani jezici nastali kao derivati C++ jezika
FORTRAN i Ada – imperativni jezici treće generacije, znači sa novim proširenjima imaju mnoge karakteristike objektno-orijentisanih jezika
Ovde ćemo primeniti imperativnu paradigmu, jer je primenjuju i objektno-orijentisani jezici, jer njihovi metodi predstavljaju kratke imperativne (proceduralne) programe.

Program sadrži skup iskaza koji spadaju u tri kategorije:

1. ***Deklarativni iskazi*** – definišu prilagođenu terminologiju koja se upotrebljava u programu, ko što su nazivi podataka koji se koriste u programu.
2. **Imperativni iskazi** – opisuju korake primenjenih algoritama
3. **Komentari** – poboljšavaju čitljivost programa objašnjavajući svojstva programa u čoveku razumljivijoj formi.

Imperativni program (ili metod u objektno-orujentisanom programu) ima strukturu prikazanu na Slici-5. Počinje sa nizom deklarativnih iskaza koji opisuju podatke koji će biti korišćeni u programu. Zatim dolaze imperativni iskazi koji opisuju algoritam po kom će se program izvršavati. Komentari se mogu, po potrebi, koristiti svuda u programu da bi se povećala njegova razumljivost.



Slika 4.5 Struktura tipičnog imperativnog programa ili programske jedinice

PROMENLJIVE I TIPOVI PODATAKA

Promenljiva je naziv memorijske lokacije sa podatkom promenljive vrednosti. Tip podataka određuje način na koji se promenljiva smešta u memoriju i akcije koje se mogu izvesti nad njom

Programski jezici visokog nivoa koriste imena za označavanje memorijskih lokacija (umesto numeričkih adresa). One su "promenljive" jer podaci na tim lokacijama mogu promeniti svoju vrednost u toku izvršenja programa. One se moraju definisati na početku programa deklarativnim iskazima da bi se označio njihov tip popodataka koji će biti smešten u memorijskoj lokaciji koja je namenjena nekoj promenljivoj. Taj tip se naziva – tipom podataka (data type) i on određuje način na koji će se promenljiva usladištiti (memorisati) i operacije koje se mogu izvršiti nad njom. Primeri:

- integer – tip koji označava samo cele brojeve, sa tradicionalnim aritmentičkim operacijama i sa upoređivanjima njihovih vrednosti
- float – označava brojeve koji ne moraju biti celi brojevi i koji se skladište sa decimalnom tačkom, a sa operacijama koje su slične operacijama sa podatcima tipa integer.

Na početku programa, programer daje deklarativne iskaze kojim određuje tipove svih podataka i promenljivih koje program kasnije koristi. Na taj način, daje instrukciju kompajleru o prostoru koji treba da obezbedi na memorijskim lokacijama u glavnoj memoriji računara na kojima će se nalaziti vrednosti ovih podataka u toku izvršenja programa.

Jedan deklarativni iskaz se može koristiti i za više podataka i promenljivih, kao na primer:

```
int Height, Width;
```

U najvećem broj programskih jezika, moguće je u deklarativnom iskazu i dodeliti promenljivoj određenu, početnu vrednost. Na primer:

```
int WeightLimit = 100;
```

Postoje i drugi tipovi podataka:

- char – označava podatak koji se sastoji od simbola (znakova), a operacije nad njima najčešće vrši upoređivanje simbola (npr. slova) ili određivanje da li se jedan simbol javlja pre drugog po alfabetskom redosledu, ili se proverava da li se jedan niz simbola (string) javlja untar i dr.

Iskaz:

```
char Letter, Digit;
```

se može koristiti u jezicima C, C++, C# i Java radi deklarisanja da su promenljive Letter i Digit tipa char.

LOGIČKE PROMENLJIVE

Logičke promenljive se označavaju sa tipom Boolean i predstavljaju podatak koji može imati samo jednu od dve moguće vrednosti: true (istinito) ili false (pogrešno).

U svim jezicima postoji i sledeći tip podataka:

- Boolean– označava podatke koji mogu imati samo jednu od dve moguće vrednosti: true (istinito) ili false (pogrešno). Operacije nad ovim tipom najčešće proveravaju da li je trenutna vrednost promenljive true ili false.

Na primer, u sledećem iskazu se proverava da li je prethodno deklarisana promenljiva tipa Boolean ima vrednost true.

```
if (LimitExceeded) then (...) else (...)
```

Tipovi podataka koje smo do sada naveli (integer, float, char, Boolean) se definišu u svim programskim jezicima, nazivaju se **primitivnim tipovima podataka**. Pored njih, mogu se javiti tipovi podataka koji se još ne koriste šire, kao što su: slike(image), zvučni zapis (audio), video zapis (video) i hipertekst (hypertext). U objektno-orijentisanim jezicima programer može da sam definiše svoje tipove podataka.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

STRUKTURA PODATAKA

Struktura podataka je konceptualno oblikovanje podataka ili uređivanje podataka

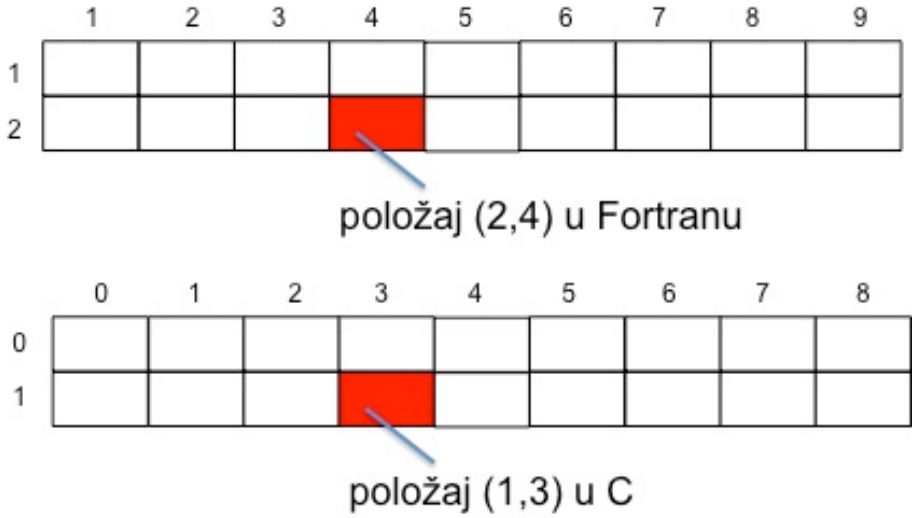
Pored dodeljivanja nekoj promenljivoj određeni tip podataka, mogu se promenljive povezati sa strukturom podataka. Struktura podataka je konceptualno oblikovanje podataka ili uređivanje

podataka. Na primer, neki tekst može da se predstavi kao jedan dugačak niz (string) znakovnih simbola (broj, slovo, posebni znaci...).

Vrlo česta struktura podataka je array (niz). Array je blok elemenata istog tipa, kao što je jednodimenziona lista, ili dvodimenziona tabela, sa svojim redovima i kolonama, ili neka višedimenziona tabela. Programski jezici omogućavaju programeru da na početku programa deklariše ime niza (array), kao i dužinu (broj elemenata) svake dimenzije višedimenzionog niza. Na slici 6 je predstavljena struktura podataka u vidu jednog dvodimenzionog niza podataka koja se deklariše sledećim iskazom:

```
int Scores[2][4];
```

Kada je neka struktura podataka deklarisana na početku programa, onda se ona može pozivati svojim imenom, a svaki njen element (određenog primitivnog tipa) se može preuzimati ili ažurirati, pri čemu do određenog elementa dolazimo ako znamo njegov indeks (redni broj) duž svake od dve dimenzije. Menjajući ove indekse, može se pristupiti bilo kom elementu, ako je potrebno, preko odgovarajuće vrednosti indeksa svake od dimenzije strukture.



Slika 4.6 Dvodimenzioni niz sa dva reda i devet kolona

HETEROGENE STRUKTURE

Heterogoni nizovi sadrže različite tipove primitivnih podataka.

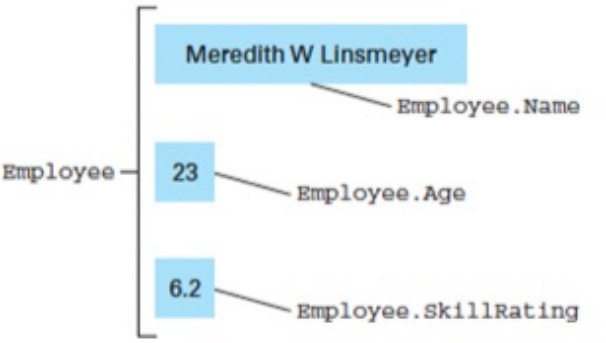
Pored nizova čiji su svi elementi istog primitivnog tipa, postoje nizovi koji sadrže *različite tipove primitivnni podataka*, te se ovi tipovi podataka nazivaju agregatni tipovi , ili se koriste i druga imena: structure, ili slog (record) ili heterogeni niz, tj. heterogenious array. U ovom slučaju, elementi bloka mogu biti različitog tipa. Na primer, u jeziku C struktura podataka pod nazivom Employee se može iskazati na sledeći način:

```
struct {
    char Name[25];
    int Age;
```

```
    float SkillRating;
} Employee
```

Elementi svake od tri dimenzije niza su različitog tipa. Kada je jedna struktura deklarisana na početku programa, programer može kasnije da poziva ili da pristupa određenom jednodimenzionom nizu (redu) višedimenzione strukture, na taj način što će u programu navesti ime strukture (tj. **Employee**) staviti tačku, a iza nje upisati ime polja (field) strukture, tj. jednodimenzionog niza, kao na primer: **Employee**.**Name** ili **Emplyee.Age** ili **Employee.SkillRating** .

Na slici 7 je prikazan jedan slučaj primene strukture **Employee.**



Slika 4.7 Primer strukture Employee

KONSTANTE

Konstanta je podatak koji ne menja vrednost u toku izvršenja programa.

Ako neki podatak treba da ima nepromenjenu, konstantnu vrednost za vreme izvršenja programa, onda se on naziva konstantnom. Konstanta se u programskim jezicima C++ i Java deklariše na sledeći način:

C++:

```
const int AirportAlt = 645;
```

Java:

```
final int AirportAlt = 645;
```

Ovo znači da se identifikator AirportAlt povezuje sa vrednošću 645, tj. definiše se konstanta sa imenom AirportAlt sa vrednošću 645. Bilo gde u progamu, kada se poziva konstanta AirportAlt, automastki će biti korišćena njena vrednost 645. Vrednost konstante se ne može menjati za vreme izvršenja programa.

Posle deklarisanja promenljivih i konstanti, programer počinje da opisuje algoritme koje će koristiti. Za tu svrhu se koriste **imperativni iskazi.** Najosnovniji imperativni iskaz je iskaz dodele. Na ovaj način se određeoj promenljivoj, tj. njenoj lokaciji, biti dodeljena neka

vrednost. Oznaka za operaciju dodeljivanja je znak jednakosti =. S njegove leve strane je ime promenljive, a s desne, izraz koji koji se izvršava i njegov rezultat se dodeljuje promenljivoj s leve strane znaka jednakosti. Evo primera u kome se zbir vrednosti promenljivih X i Y dodeljuje promenljivoj Z.

Bilo koji aritmetički iskaz može da bude izkaz sa desne strane znaka dodeljivanja (=). Dobijeni rezultat posle izvršenja se dodeljuje promenljivoj s leve strane znaka operacije dodele.

ISKAZI UPRAVLJANJA

Iskaz upravljanja definiše redosled izvršnih operacija u programu.

Iskaz upravljanja definiše redosled izvršnih operacija u programu. Oni izvršavaju algoritam (postupak) rešavanja problema. U svim jezicima se koriste iskazi upravljanja poznati kao if-then-else and while iskazi. Njihov pseudo kod, tj. kod koji objašnjava njihov rad, ali nije namenjen računaru već čoveku jer je u obliku koji je lako razumljiv, izgleda ovako:

If-then-else:
if (uslov)
then (iskazA)
else (iskazB)

while:
while (uslov) do
(telo petlje)

U programskim jezicima C, C++, C# i Java , ovi iskazi bi bili napisani na sledeći način:

```
if (uslov) iskazA
    else iskazB;
```

i

```
while (uslov)
    {telo petlje};
```

switch:

Pored navedena dva iskaza upravljanja, često se koriste i iskaz swich i case. On omogućuje da se izabere jedna od ponuđenih nekoliko sekvenci programskih iskaza, zavisno od vrednosti dodeljenoj određenoj promenljivoj. Naprimer, iskaz:

```
switch (promenljiva) {
    case 'A': iskazA; break;
    case 'B': iskazB; break;
    case 'C': iskazC; break;
    default: iskazD
}
```

napisan u C, C++, C# i Javi određuje izvršenje jednog od ponuđenih iskaza (iskazA, ili iskazB, ili iskazC), zavisno da li promenljiva ima vrednost 'A', 'B', ili 'C'. Za neku drugu vrednost, program izvršava iskaz iskazD.

PRIMERI UPOTREBE ISKAZA UPRAVLJANJA

Uz pomoć narednih primera utvrditi znanje o iskazima upravljanja

Napomena. U toku pisanja nekih od narednih algoritama koristićemo sledeće simbole za odgovarajuće operacije:

- '+' za sabiranje
- '-' za oduzimanje
- '*' za množenje
- '/' za deljenje
- '←' za dodelu (tj. =) . Na primer, A ← X*3 znači da će A nakon operacije imati vrednost X*3.

Primer 1. Napisati algoritam kojiodlučuje da li je student položio ispit

```
READ bodovi
IF(bodovi>=51)
     THEN student je položio
```

Primer 2. Kontrola pristupa sistemu na osnovu unetog korisničkog imena i lozinke:

```
IF (postoji korisnik sa unetim korisničkim imenom i lozinkom) THEN
    Odobri pristup
ELSE
    Zabrani pristup
```

Ako je ispunjen if uslov, odobrava se pristup, ako if uslov nije ispunjen, pristup se zabranjuje. Ispunjen uslov znači da je njegova vrednost ***true***.

Primer 3: Napisati algoritam koji izračunava konačnu ocenu studenta i na osnovu toga izveštava da li je položio ispit ili je pao. Ukupna ocena se računa kao srednja vrednost tri ocene. Pseudokod:

```
Input a set of 4 marks
Calculate their average by summing
     and dividing by 4
if average is below 50
     Print "FAIL"
else
     Print "PASS"
```

Detaljan algoritam:

```
Step 1: Input M1,M2,M3,M4
Step 2: GRADE <- (M1+M2+M3+M4)/4
```

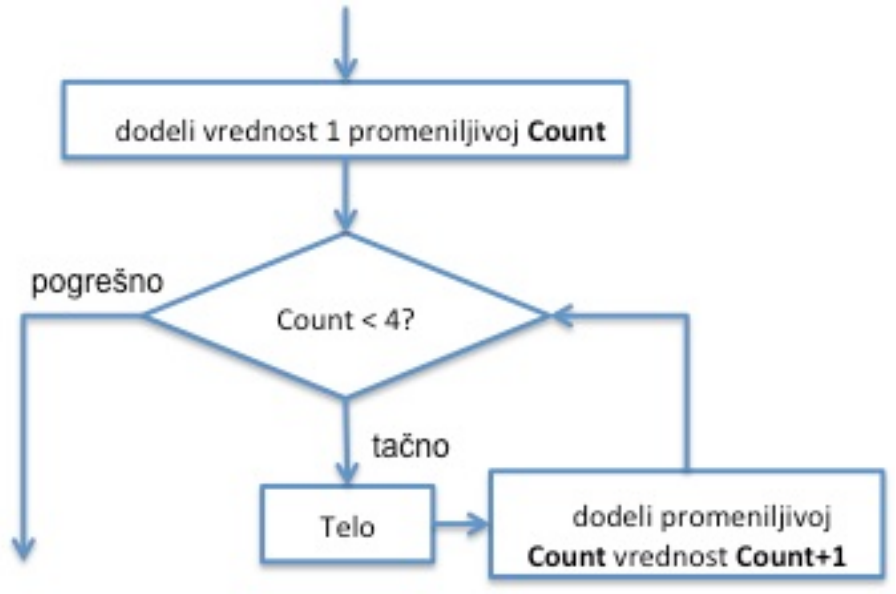
```
Step 3: if (GRADE < 50) then
            Print “FAIL”
        else
            Print “PASS”
        endif
```

UPRAVLJAČKA STRUKTURA FOR

Iskaz upravljanja for definiše programsku petlju, tj. sekvencu programskih iskaza, koja će se izvršavati onoliko puta koliko je definisano u for iskazu

Pored navedena tri iskaza upravljanja, u programskim jezicima C, C++, C# i Java se koristi i iskaz upravljanja for. Ovaj iskaz definiše programsku petlju, tj. sekvencu programskih iskaza, koja će se izvršavati onoliko puta koliko je definisano u for iskazu, kao što je pokazano u primeru na slici 8. U ovom primeru, telo petlje se izvršava tri puta, za slučaj kada promenljiva Count ima vrednost 1, 2 i 3. Kada dobije vrednost 4 ili neku drugu, izvršenje programa zaobilazi navedno telo programske petlje.

Primenom navedena četiri iskaza upravljanja, ili njihovih derivata u nekim drugim programskim jezicima, računarski programi izvršvaju svoje predviđene operacije. Ovo znači da bilo koji problem možete rešiti upotrebom ovih iskaza upravljanja.




```
for (int Count = 1; Count < 4; Count++)
    body ;
```

Slika 4.8 Primer upotrebe upravljačke strukture for

KOMENTARI

Komentari su iskazi koji ne utiču na izvršenje programa jer su namenjeni samo čoveku jer mu daju objašnjenja pojedinih delova programa.

Važno je da programer napiše svoj program tako da ga i drugi programeri mogu lako da razumeju, jer će drugi programeri raditi u održavanju nekog programa ili u daljem njegovom razvoju. Isti program može da se napiše tako da drugi programeri mogu s mukom da shvate kako program radi, a može i da se napiše tako da svima bude odmah jasno kako on radi. Vrlo je važno da programer upravo tako napiše svoj program, tako da svi mogu lako da razumeju njegovu ideju i način njenog izvršenja u programu. Zbog toga, svi programski jezici omogućavaju iskaze koje nazivamo komentarima (coments), jer njihov sadržaj ne utiče na izvršenje programa, jer sadrže samo tekst koji je namenjen čoveku. Taj tekst objašnjava određene delove programa.
U jezicima C++, C# i Java se koriste dva načina za unos komentara u program. U prvom, komentar se unosi u više redova, a označava se početak i kraj redova sa komentarima. Evo primera:

```
/*  Ovde se unosi tekst komentara u jednom ili više redova    */
```

Drugi način omogućava unos teksta komentara samo u jednom redu:

```
//  Ovde se unosi tekst kratkog komentara u samo jednom redu
```

Iskazi sa komentarima se mogu koristiti bilo gde u programu. Dobra praksa je da se na početku programa, ili programske jedinica, u više redova preko komentara da objašnjenje šta program radi, šta su mu promenljive i čemu one služe, i koji algoritam program koristi i sa kojom svrhom. Kasnije u programu, poželjno je, gde treba, ubaciti komentar u jednom ili par redova, u kojima se objašnjava šta ili zbog čega se koriste naredni programski iskazi (ili instrukcije ili naredbe).

POKAZNI PRIMERI - ISKAZI UPRAVLJANJA

U nastavku navodimo primere upotrebe iskaza upravljanja

Primer 1. Napisati algoritam koji učitava dva broja, određuje veći među njima i i štampa poruku o najvećoj vrednosti.

```
Korak 1: Input VALUE1, VALUE2
Korak 2: if (VALUE1 > VALUE2) then
             MAX <- VALUE1
         else
             MAX <- VALUE2
         endif
Korak 3: Print “The largest value is”, MAX
```

Algoritam alternativno može biti napisan i na sledeći način

```
Korak 1: Start
Korak 2: Read/input A and B
Korak 3: If A greater than B then C=A
Korak 4: if B greater than A then C=B
Korak 5: Print C
Korak 6: End
```

Primer 2. Napisati algoritam koji učitava tri broja i štampa vrednost najvećeg broja.

```
Step 1: InputN1, N2, N3
Step 2: if (N1>N2) then
            if (N1>N3) then
                MAX <- N1[N1>N2, N1>N3]
            else
                MAX <- N3[N3>N1>N2]
            endif
        else
            if (N2>N3) then
                MAX <- N2[N2>N1, N2>N3]
            else
                MAX <- N3[N3>N2>N1]
            endif
        endif
Step 3: Print “The largest number is”, MAX
```

Algoritam alternativno može biti napisan i na sledeći način

```
Korak 1: Start
Korak 2: Read/input A,B and C
Korak 3: If (A>=B) and (A>=C) then Max=A
Korak 4: If (B>=A) and (B>=C) then Max=B
Korak 5: (C>=A) and (C>=B) then Max=C
Korak 6: Print Max
Korak 7: End
```

POKAZNI PRIMERI - PETLJE

U nastavku navodimo primere upotrebe petlji

Primer 1. Napisati algoritam koji računa prosečnu vrednost prvih 10 brojeva.

```
total=0
average=0
FOR 1 to 10
  Read number
  total=total+number
END FOR
average=total/10
```

Primer 2. Učitavaj i sabiraj brojeve sve dok je njihov ukupan zbir manji od zadate vrednosti S.

```
WHILE(total<S)DO
  Read number
  total=total+number
END WHILE
```

Koja je razlika između primera 1 i 2?

Razlika je u tome što u prvom primeru znamo unapred tačan broj ponavljanja, a to je 10 puta. U drugom primeru to ne znamo. U drugom primeru, broj ponavljanja zavisi od unetih brojeva.

Primer 3: Algoritam koji određuje srednju vrednost brojeva od 0 do 99

```
Step 1. Start
Step 2. I ← 0
Step 3. Write I in standard output
Step 4. I ← I+2
Step 5. If (I <=98) then go to Step 3
Step 6. End
```

Primer 4: Projektovati algoritam koji sa ulaza dobija prirodni broj n, i štampa sve parne brojeve koji su manji ili jednaki N:

```
Step 1. Start
Step 2. Read n
Step 3. I ← 1
Step 4. Write I
Step 5. I ← I + 2
Step 6. If ( I <= n) then go to Step 4
Step 7. End
```

Primer 5: Napisati pseudokod za programski segment koji iznova i iznova traži od vas da unesete broj u opsegu od 1 do 100 sve dok ne unesete korektan broj (broj je nekorektan ako je manji od 1 ili veći od 100).

```
REPEAT
    DISPLAY “Enter a number between 1 and 100”
    ACCEPT number
UNTIL number < 1 OR number > 100
```

ZADACI ZA SAMOSTALNI RAD

Na osnovu materijala prethodno opisanih primera uraditi samostalno sledeće zadatke:

Zadatak 1. Napisati algoritam koji određuje zbir brojeva 2, 4, 6, 8, …, n

Zadatak 2. Napisati algoritam koji učitava 100 celih brojeva i štampa njihov zbir.

Zadatak 3. Napisati algoritam koji učitava 4 broja i štampa najveći.

Zadatak 4. Napisati algoritam koju učitava sa ulaza 100 brojeva i štampa najveći među njima.

Zadatak 5. Napisati algoritam koji određuje srednju vrednost N unetih brojeva.

Zadatak 6. Napisati algoritam koji redom učitavana brojeve sa ulaza, određuje i štampa kvadratni koren (funkcija SQRT) . Prekid unosa novih brojeva se vrši kada se unese broj 0.

ZAKLJUČAK

Na osnovu svega obrađenog možemo zaključiti sledeće:

Televizor je za nas "crna" i nepoznata kutija. Jedino o čemu razmišljamo je kako ćemo da ga upalimo, da se uvalimo u fotelju i da ga gledamo. Znamo da je TV sredstvo komunikacije koje se koristi da bi poboljšalo naše živote. Tako i računari postaju uobičajena stvar kao i televizori, jednostavno: sastavni deo naših života. I slično kao kod televizora, računari se baziraju na složenim principima ali su dizajnirani tako da su laki za korišćenje.

Računari nisu baš inteligentni; mora im se precizno reći šta da urade. Stoga su prave računarske greške veoma retke (obično usled otkaza delova ili električnih kvarova). Pošto mi govorimo računarima šta treba da urade, najveći deo grešaka koje prave računari su ustvari ljudske greške.

Računari se danas veoma puno koriste u nauci, inženjerstvu, poslovanju, državnoj upravi, medicini, proizvodnji dobara, i u umetnosti. Učenje programiranja u Javi vam može omogućiti da ovaj moćan alat koristite još efikasnije.