S v e u č i l i š t e   u   Z a g r e b u

F a k u l t e t   s t r o j a r s t v a   i   b r o d o g r a d n j e

Katedra za osnove konstruiranja

N. Pavković, D. Marjanović, N. Bojčetić

PROGRAMIRANJE I ALGORITMI

Skripta, prvi dio

Zagreb, 2005.

Sadržaj

PREDGOVOR 4
DIGITALNO RAČUNALO 1
Virtualna arhitektura računala 1
Hardver, softver i višerazinska računala 4
Dijelovi računala - način rada 4
Prikaz načina rada jednostavnog procesora 5
PROGRAMSKA PODRŠKA 6
Sistemska programska podrška 6
Operacijski sustavi 6
Operacijski sustav MS Windows 7
Operacijski sustav UNIX 7
Programski jezici (prevodioci) 8
Razvojne okoline 8
Aplikacijska programska podrška 8
Datoteke 9
UVOD U ALGORITME I PROCES RAZVOJA PROGRAMSKE PODRŠKE 9
Pojam algoritma 9
Značajke algoritma 10
Prikaz algoritma 11
Simboli dijagrama toka 12
Faze razvoja programa 13
Analiza problema 14
Postavljanje modela 14
Izrada algoritma 14
Izrada dijagrama toka 14
Kodiranje programa 15
Prevođenje programa 15
Testiranje programa 16
Dokumetiranje programa 16
Eksploatacija programa 17
OSNOVNI KONCEPTI PROGRAMSKIH JEZIKA 17
Sintaksa i semantika programskog jezika 17
Formalne metode prikaza sintakse 17
Sintaksa, semantika i prevođenje programa 18
Tipovi i strukture podataka u programskim jezicima 19
Osnovni (primitivni tipovi) podataka 19
Tipovi podataka u Visual Basic-u: 20
Varijable 20
Određivanje imena varijable 20
Adresa varijable 21
Tip varijable 21
Vrijednost varijable 21
Doseg varijable 21
Trajanje varijable 22
Deklaracija varijable 23
Pokazivači (kazala, "pointeri") 23
Polja 24
Deklariranje polja 25
Izrazi (eng. expressions) 26
Zagrade 26
Aritmetički izrazi 27
Redoslijed izvođenja operatora 27
Konverzija tipova podataka u izrazima 27
Relacijski izrazi 28
Logički (Boolean) izrazi 28
Mješoviti izrazi 29
Naredba za dodjeljivanje (eng. assignment statement) 29
Kontrolne strukture na razini naredbi 30
Bezuvjetni skok 31

Uvjetna grananja (naredbe selekcije) 31
Odlučivanje na temelju vrijednosti numeričkog izraza 31
Odlučivanje (uvjetno grananje) – “Ako ..... Onda” 32
Uvjetno grananje: – “Ako ..... Onda ..... Inače” 32
Uvjetno grananje sa višestrukim ispitivanjem uvjeta 33
Ugnježđenje kontrolnih struktura - uvjetna grananja jedno unutar drugog 34
Višegransko usmjeravanje 35
Naredbe za ponavljanje izvođenja sekvenci programa (petlje) 36
Petlje kontrolirane eksplicitnim brojačem 37
Petlja "For Each .... Next" 38
Petlje kontrolirane logičkim uvjetima 38
Petlja sa ispitivanjem logičkog uvjeta na početku 38
Petlja sa ispitivanjem logičkog uvjeta na kraju 39
Petlja sa ispitivanjem logičkog uvjeta koja se izvodi dok je uvjet neistinit 39
Naredbe za izlaz iz petlje 40
Ugnježđenje petlji - više petlji jedna unutar druge 40

PREDGOVOR

Ove podloge namijenjene su studentima Fakulteta strojarstva i brodogradnje kao osnovna literatura za praćenje predavanja i vježbi iz kolegija PROGRAMIRANJE I ALGORITMI i kolegija PRIMJENA RAČUNALA B. Pretpostavka uspješnog savladavanja kolegija je redovito praćenje nastave i samostalan rad na računalu. Vježbe se održavaju na računalima u PC učionicama CADLab -a.
Pisanje podloga za kolegij koji obrađuje materiju koja je tako podložna brzim promjenama kao računalstvo, nezahvalan je posao. Sjena zastarijevanja nadvila se nad rukopis onog trenutka kada je započela priprema teksta. Ipak vjerujemo da će podloge omogućiti studentima lakše savladavanje prvih koraka računalstva na Fakultetu strojarstva i brodogradnje Sveučilišta u Zagrebu. U CADLab -u instalirana računalna i programska oprema, razvojni alati, CAD/CAE aplikacije, pristup Internetu, te podrška asistenata i demonstratora omogućuju studentima FSB-a usvajanje znanja potrebnih svakom inženjeru. Stoga očekujemo da će studenti koristiti računala u svakodnevnom radu i izvan kolegija Katedre za osnove konstruiranja.

Namjena je gradiva izloženog u ovim skriptama dati općeniti pregled tehnika programiranja i kreiranja algoritama neovisno o pojedinom programskom jeziku i operativnom sustavu. Zadnjih nekoliko godina na vježbama se uči programski jezik Visual Basic, pa je stoga većina primjera u ovim podlogama prilagođena tome, a objašnjena je i sintaksa osnovnih naredbi Visual Basic-a. Gdje se smatralo pogodnim, dani su i primjeri iz drugih programskih jezika ili pregledi i usporedbe različitih postupaka i sintaksi u više različitih jezika. Važno je napomenuti da učenje programiranja ne znači samo učenje naredbi (sintakse) pojedinog jezika, nego prvenstveno znači naučiti osnove kreiranja algoritama te poznavati zajedničke teoretske osnove svih programskih jezika, odnosno poznavati osnove računalne znanosti. Predavanja iz ovog kolegija upravo stoga nastoje dati pregled osnova računalne znanosti neophodnih za učenje i razumijevanje procesa razvoja složenih programskih sustava.

U satnicom dopuštenom okviru namjena je ovog kolegija budućem inženjeru dati osnovna znanja o načinu rada računala i razvoju jednostavnih programskih aplikacija za svakodnevnu uporabu u inženjerskoj praksi. Dani pregled metodologija razvoja programskih susutava trebao bi inženjera osposobiti za aktivnog suradnika u izradi složenih aplikacija i sustava. Znanje stečeno na ovom kolegiju može biti polazište za dublje proučavanje svijeta računalnog programiranja – područja ljudske djelatnosti koje se u današnje vrijeme vjerojatno najbrže razvija i mijenja u usporedbi s drugim djelatnostima.

Autori

U Zagrebu, listopad 2005.

DIGITALNO RAČUNALO

Zakonitosti koje opisuju svijet oko nas spoznajemo kroz podatke i informacije i neprestano dograđujemo povećavajući tako količinu znanja.
Veliki broj međuzavisnosti u našoj okolini, velika brzina u kojoj se nižu događaji i promjene oko nas, te sve veći značaj naše interakcije s okolinom, zahtijevaju izgradnju pomagala čovjeku koje će mu povećati: sposobnosti sagledavanja, sposobnosti primanja i obrade podataka i informacija, sposobnosti zaključivanja i sposobnosti odlučivanja.
Računalo kao temeljno informacijsko pomagalo čovjeku, prema gore navedenim potrebama, razvijeno je tako da omogući primanje, uskladištenje, obradu i slanje podataka, informacija i znanja.

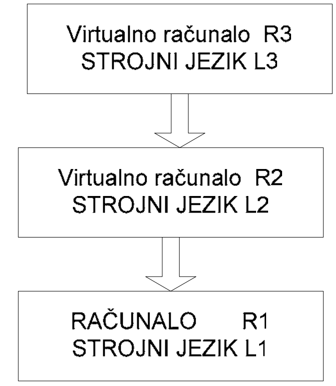
Digitalno računalo je stroj koji može rješavati probleme izvodeći dane mu instrukcije. Niz instrukcija koji opisuje kako se rješava pojedini zadatak naziva se PROGRAM.
Računalo se sastoji od elektroničkih sklopova koji mogu prepoznati i izvršiti organičeni skup instrukcija, u koji se prevode sve instrukcije programa koji se želi izvesti na računalu. Osnovne instrukcije vrlo su jednostavne. U biti nisu mnogo složenije od:
- zbrajanja dva broja;
- uspoređivanja broja s nulom;
- kopiranja podatka s jednog dijela memorije računala u drugi dio;
- skoka na slijedeću instrukciju programa.

Primitivne instrukcije tvore jezik pomoću kojih možemo upravljati, odnosno komunicirati s računalom. Ovakav jezik nazivamo STROJNI JEZIK.

Virtualna arhitektura računala

Pošto su instrukcije jednostavne, teško je i mukotrpno upotrebljavati ih, odnosno pisati programe u strojnom jeziku.
Problem se može rješiti na principjelno dva načina, ali oba zahtjevaju razvoj novog jezika, (skupa instrukcija) koji je bliži ljudima, odnosno lakši za upotrebu od ugrađenog strojnog jezika.
Novi skup instrukcija koji se razvija nazovimo L2, kao što ćemo označiti i strojni jezik oznakom L1. Ova se dva pristupa razlikuju u načinu na koji se programi pisani u jeziku L2 izvode na računalu koje u stvari može izvoditi samo instrukcije iz skupa L1.
Jedna metoda izvođenja programa napisanog u L2 jeziku je da se sve instrukcije prvo zamijene s ekvivalentnim nizom instrukcija u L1 jeziku. Prevodi se cijeli program iz jezika L2 u jezik L1. Računalo izvodi novi program napisan u L1 jer jedino to i može. Ovu metodu nazivamo PREVOĐENJEM. Program za prevođenje iz jezika L2 u L1 naziva se COMPILER.
Drugi pristup je da napišemo program u L1 jeziku kojemu će ulazni podaci biti instrukcije programa napisanog u L2. Program će čitati jednu po jednu instrukciju, svaku analizirati i odmah izvoditi ekvivalentni niz strojnih instrukcija jezika L1, za svaku instrukciju jezika L2. Ne generira se cijeli novi program u L1 jeziku. Metodu nazivamo inerpretacija, a program koji interpretira instrukcije INTERPRETER.

Prevođenje i interpretacija su slične metode i obje su u širokoj upotrebi. U obje metode instrukcije programa napisanog u L2 izvršavaju se izvođenjem ekvivalentnog niza instrukcija u jeziku L1. Razlika je što s kod prevođenja cijeli program L2 prevodi u L1. Time program napisan u L2 postaje nepotreban. Izvodi (izvršava) se samo program napisan u L1. Nasuprot tome, pri interpretiranju ne generira se novi program, već se svaka instrukcija programa L2 analizira, dekodira i odmah izvršava odgovarajući niz L1 instrukcija.

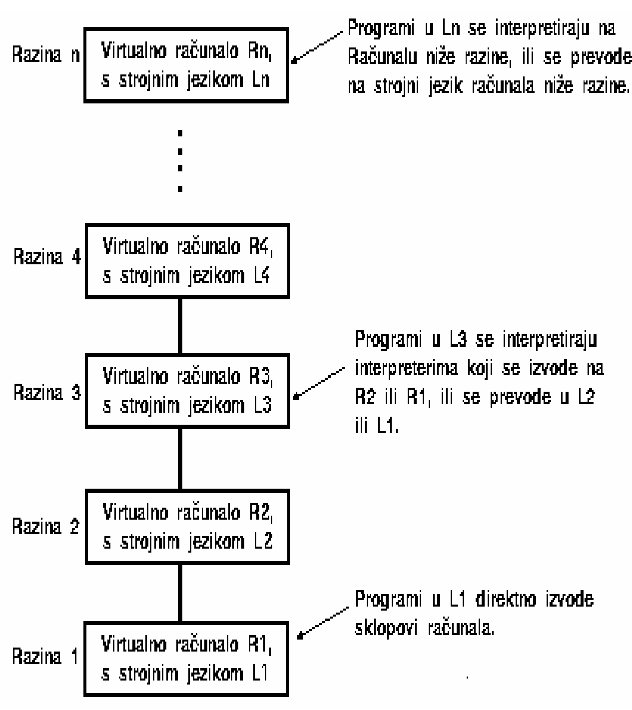


Slika 1: Virtualna arhitektura računala

Kako nebi uvijek razmišljali o prevođenju ili interpretaciji zamislimo da postoji računalo čiji je strojni jezik L2. Kada bi se takvo računalo moglo (dovoljno jeftino) realizirati nebi postojala potreba za L1. Međutim čak ako to i nije moguće možemo zamisliti takvo računalo, hipotetičko računalo ili virtualno računalo, čiji jezik je L2. Činjenica da se u stvarnom računalu instrukcije L2 prevode ili interpretiraju u L1 nije od značaja za ovo razmišljanje, ali nam omogućuje proučavanje strukture računala kroz različite razine. Naime, da bi prevođenje ili interpretacija bili praktično provedivi jezici L2 i L1 se ne smiju "previše" razlikovati. Ovo na prvi pogled nije u skladu s osnovnom idejom L2, jer ovakvo ograničenje svakako uzrokuje određenu nepraktičnost za razvoj aplikacija.

Rješenje problema koje se samo nameće je razvoj jezika L3 koji će biti više prilagođen ljudima, a manje stroju. Ljudi mogu programirati u L3 baš kao da je to strojni jezik hipotetskog računala. Takvi programi mogu se prevoditi u jezik L2 ili se interpretirati pomoću interpretera napisanog u L2.

Razvoj cijele serije novih jezika, "boljih" od njihovih predhodnika može ići bez ograničenja sve dok se ne postigne željeni cilj. Svaki jezik koristi svog prethodnika kao temelj i određuje jedno virtualno računalo. Ovo nam omogućuje proučavanje računala na nizu različitih razina (slika), ovisno o našim interesima. Jezik na najnižoj razini je najjednostavniji, a onaj na najvišoj razini je najsofisticiraniji.



Slika 2: Računalo razmatrano kroz niz razina

U ovakvom razmatranju važno je uočiti odnos između jezika i virtualnog računala. Svako računalo određeno je strojnim jezikom kojim se mogu opisati instrukcije koje računalo može izvršiti. Računalo definira jezik i obrnuto – jezik definira računalo, tj. računalo može izvršiti sve programe napisane u tom jeziku.

Računalo s *n* razina možemo razmatrati kao *n* računala s različitim strojnim jezicima, pri tome ne treba zaboraviti da se samo programi u strojnom (L1) jeziku mogu direktno izvršavati na elektroničnom sklopovlju računala. Međutim osoba koja piše programe za *n*-tu razinu ne mora biti upoznata s detaljima prevođenja (ili interpretiranja) na razinama nižim od one koja je neophodna za korištenje računala za potrebe određene vrste poslova. Većinu korisnika zanima samo razina na kojoj rade, a sve niže od te, nisu predmet njihovog interesa. Ovako opisana virtualna struktura računala može se primjeniti i na izvedena računala (koja su u komercijalnoj uporabi).

Svrstavanje razina računala može se prikazati na slijedeći način:

Razina sklopova - program kao niz instrukcija ne egzistira.
Mikroprogramska razina - ne postoje dva računala s identičnim mikroprogramskim razinama, (njihovi se centralni procesori razlikuju u broju i načinu izvođenja instrukcija) ali postoje zajedničke karakteristike kao i velike sličnosti. No npr. dvadesetak osnovnih instrukcija procesora dovoljno je da se kreira upotrebljivo računalo. Svako računalo može izvršavati barem jedan mikroprogram, koji implicitno definira jezik na 2. razini.
Strojna razina - je stvarna razina strojnog jezika. Na ovoj razini veće su sličnosti nego različitosti među jezicima odnosno arhitekturama različitih proizvođača računala/procesora. Ovakav je pristup subjektivan, pa ćemo zbog opće terminološke neusaglašenosti ovu razinu nazivati konvencionalnom strojnom razinom. Priručnici proizvođača računala, koji opisuju jezike ove razine (tipični naziv: Machine Language Reference Manual ...), de facto opisuju instrukcije virtualnog računala čije se instrukcije interpretiraju i izvršavaju na nižoj razini.
Razina operacijskog sustava (OS) - je hibridna razina: dio interpretira OS, a djelomično se parsira na nižu razinu. Ova razina nije nužno ekvivalentna sa korisničkim sučeljem operacijskog sustava.
Općenito ne postoje fiksna pravila kada se kod interpretira, a kada se prevodi (jer to ovisi od projektanata računala), ali može se reći da se kod na niskim razinama uglavnom interpretira, a na višim prevodi.
Assemblerska razina – na kojoj postoji simbolički oblik strojnih instrukcija, za razliku od numeričkog oblika na nizim razinama.
Razina programskih jezika (razvojnih okolina) – ovu razinu tvore simbolički jezici, podobni za rješavanje različitih problema (npr. BASIC, FORTRAN, PASCAL, C itd.). Programi pisani u takvim, visoko simboličkim jezicima prevode se pomoću programa-prevodioca (COMPILER-s) na niže razine, a ponekad se interpretiraju interpreterima (INTRPRETER-s).
Aplikacije predstavljaju skupine razvijenih programa prilagođenih zahtjevima određenog područja primjene. To praktično znači da su u te programe ugrađene velike količine informacija o pojedinom području primjene (aplikaciji). Ova razina je tek predmet istraživanja, pa možemo zamisliti računala namijenjena specijalno samo za npr. administraciju, obrazovanje, itd. Korisnik, koji se koristi tom razinom računala ne mora znati što se dešava na nižim razinama.

Slika 3: Razine arhitekture modernih računala

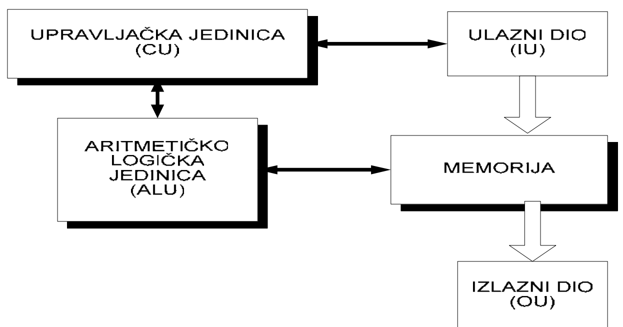
Ono što je za naše razmatranje značajno možemo sažeti na slijedeći način:
Računala su konstruirana i mogu se razmatrati kao serijski niz logičkih cjelina, a svaka se od njih osniva na prethodnoj. Svaka cjelina predstavlja drugu razinu apstrakcije. Na ovaj način možemo zanemariti složene detalje koji nisu značajni za naša razmatranja.
Skup informacija, instrukcija, podataka i svojstava pojedinog računala nazivamo arhitekturom računala.

Hardver, softver i višerazinska računala

Hardver (HW) se sastoji od sklopova računala – "fizičkih" ("opipljivih") objekata.
Softver (SW) se sastoji se od algoritama i njihove računalne reprezentacije, odnosno programa koji čine programsku podršku. Bit softvera je niz naredbi koje čine program, a ne fizički medij na kome su pohranjene.
Treba uočiti da se svaka operacija koja se rješava softverom može rješiti i hardverom (direktno ugraditi u hardver). Razlika između SW i HW bila je potpuno jasna samo na samim početcima razvoja računala, kada je HW mogao izvršiti nekoliko jednostavnih instrukcija (ADD, STORE, MOVE, JUMP), a sve ostalo je bio mukotrpno napisani SW. Danas ne postoje fiksna pravila što će se rješiti SW, a što HW. To ovisi o zahtjevima i ciljevima projektanta.
Dakle, možemo zaključiti da su **HW i SW logički ekvivalentni**.

Dijelovi računala - način rada

Gotova sva moderna računala temelje se na hardverskoj arhitekturi koju je razvio matematičar John Von Neumann. Originalnu Von Neuman-ova arhitekturu čini pet osnovnih dijelova:
aritmetičko-logička jedinica, kontrolna (upravljačka) jedinica, memorija, ulazno-izlazni sklopovi i sabirnica koja omogućava tok podataka između svih dijelova.
Upravljačka jedinica obrađuje podatke, upravlja i nadzire protokom podataka između pojedinih dijelova sustava, te usklađuje pravilan rad cijelog sustava. Memorija (spremnik) pohranjuje podatke i programe, te ih po potrebi stavlja na raspolaganje ostalim dijelovima sustava. Ulazno-izlazni sklopovi omogućuju prijenos podataka između računala i okoline.



Slika 4: Originalna Von Neumannova arhitektura računala

Razvoj računala donio je razne izmjene i poboljšanja, ali i u današnjim računalima zadržana je osnovna ideja Von Neumannove arhitekture:

- programi i podaci se čuvaju u vanjskoj (sporoj) memoriji (tvrdi disk i sl.)
- kad se programi izvode, kopiraju se u memoriju s brzim pristupom, odakle se obrađuju u procesoru koji dohvaća i izvršava naredbe jednu po jednu

Prikaz načina rada jednostavnog procesora

Tradicionalna Von Neumann-ova arhitektura ima jedan tok instrukcija (jedan program) koji se izvršava u jednom procesoru, a jedna memorijska jedinica sadrži podatke. U računalnoj terminologiji ovakav proces se skraćeno naziva SISD (eng. - single instruction stream, single data stream). Za razliku od navedenog postoje i arhitekture sa više paralelnih tokova instrukcija i podataka.

Prepostavke primjera rada jednostavnog procesora s nekoliko registara:

- U radnom spremniku je smješten strojni oblik programa kao niz instrukcija pohranjenih u uzastopne memorijske lokacije.
- U radnom spremniku nalaze se i svi ulazni podaci koje će program dohvaćati tijekom izvođenja.
- Program će rezultate pohranjivati na određeno mjesto u spremniku.
- Unutar procesora sadržaji se mogu interpretirati:
    - kao instrukcije strojnog programa računala i
    - kao osnovni tipovi podataka.

Procesor je automatski izvoditelj programa koji obavlja instrukcije onim redom kojim su one smještene u spremniku tj. adresira instrukcije s pomoću programskog brojila. Ako se taj redoslijed izvođenja treba narušiti (izvesti neke "skokove") onda se unutar instrukcije mora prisilno promijeniti sadržaj programskog brojila.
Osnovna svojstva procesora su određena skupom registara i skupom instrukcija. Instrukcije mogu biti za premještanje sadržaja, za obavljanje aritmetičko-logičkih operacija, za programske skokove (bezuvjetni, uvjetni i skokovi u potprogram) i za upravljačka djelovanja.

Ako u programsko brojilo upišemo adresu prve instrukcije programa, procesor će automatski početi izvoditi taj program.
Struktura registara procesora:
programsko brojilo (prog. counter PC), adresni registar (AR), instrukcijski reg. (IR), akumulator (ACC), podatkovni reg. (DR), status reg. (SR).

Simbolički tok programa rada centralnog procesora:

1. DOHVATI instrukciju iz memorije, pohrani u IR.
2. Inkrementiraj PC (pomakni ga da pokazuje na slijedeću instrukciju).
3. DEKODIRAJ instrukciju (odredi tip intstrukcije).
4. Ukoliko instrukcija treba operande (podatke), odredi adresu (-e) gdje se nalaze.
5. Dohvati podatke (ako ih ima, i stavi u ACC).
6. IZVEDI instrukciju.
7. Spremi rezultat na odgovarajuće mjesto.
8. nastavi na 1. (slijedeća instrukcija)

Uobičajeno se ovakav slijed koraka naziva "FETCH-DECODE-EXECUTE" ciklus (dohvati – dekodiraj – izvedi). Ovaj ciklus je osnova rada svakog računala.

Da bi ostvarili potprogram služe nam dvije posebne instrukcije skoka:

- instrukcija za poziv potprograma
- instrukcija za povratak iz potprograma.

Pri pozivu potprograma treba prenijeti ulazne podatke, a potprogram pri završetku treba vratiti rezultate programu koji ga je pozvao.
Osnovni zadatak operacijskog sustava je da prije započinjanja nekog procesa u računalu mora stvoriti uvjete za njegovo izvođenje i da se to izvođenje može opisati modelom koji se zasniva na opisu jedostavnog računala.

PROGRAMSKA PODRŠKA

Programska podrška (software) dijeli se na dvije velike skupine: sistemsku i aplikacijsku.

Sistemska programska podrška

Namijenjena je pokretanju računala i zatim što djelotvornijem iskorištenju.Unosi se u računalo da bi ono uopće funkcioniralo, te da bi se mogućnosti računala što efikasnije iskoristile. Sistemska programska podrška može se podijeliti na slijedeće osnovne vrste:

- Operacijski sustavi
- Pomoćni programi operacijskog sustava
- Interpreteri i prevodioci- razvojne okoline

Operacijski sustavi

Operacijski sustav je skup programa koji omogućuje korištenje računala. Operacijski sustav (OS) djeluje kao okolina unutar koje se izvode postojeći programi i razvijaju novi.

Sve do nedavno gotovo svaki proizvođač računala razvijao je svoj operacijski sustav prilagođen arhitekturi i namjeni cjelokupnog računalnog sustava i njegove složenosti. Bez obzira na raznolikost sučelja operacijski sustavi su u osnovi međusobno vrlo slični.

Osnovni zadaci operacijskog sustava su:

- komunikacija s korisnikom,
- raspoređivanje procesorskog vremena,
- upravljanje ulazno / izlaznim funkcijama (kontrola repova redova?),
- raspoređivanje rada ostalih uređaja računala:
  - dodjela radne memorije,
  - kontrola i pozivanje programa s vanjskih uređaja,
  - nadzor i organizacija struktura podataka,
  - ispitivanje pogrešaka,
  - održavanje aktivnosti sustava.

Obzirom na način rada i broj korisnika operacijske sustave možemo podijeliti na **jednokorisničke** operacijske sustave i **višekorisničke** operacijske sustave.

Jednokorisnički operacijski sustavi obrađuju zahtjeve koji dolaze od jednog korisnika. To su jednostavni OS koji nemaju ugrađene mehanizme zaštite korisnika ili sustava od neautoriziranog korištenja.
Višekorisnički OS moraju obrađivati zahtjeve koji dolaze iz različitih izvora i upravljati uređajima računala na temelju tih zahtjeva. Npr. nekoliko korisnika može tražiti pristup nekom uređaju u isto vrijeme. OS mora odrediti redoslijed izvršavanja zahtjeva i omogućiti njihovo izvođenje.
Daljnji razvoj omogućio je kreiranje OS kod kojih mnogo programa može istovremeno raditi, odnosno postavljati zahtjeve OS. To su višezadaćni (eng. multitasking) OS. Iako se kod ovih OS u jednom trenutku izvodi samo jedan program u CP-u (centralni procesor), paralelno izvođenje više programa omogućeno je zbog različite brzine različitih uređaja, te zbog toga što se U/I operacije na različitim perifernim uređajima mogu odvijati paralelno.

Pitanje redoslijeda izvršavanja različitih programa, odnosno problem dodjele procesorskog vremena rješava se na dva načina:

- Na temelju sistemskih prekida, najčešće realiziranih na temelju nekih događaja u tijeku izvođenja programa, npr. U/I zahtjev (OS upravljan događajima, (eng. ("event driven")). Događajem se smatra svaka promjena stanja u izvođenju programa, pri tome nije važno tko ili što je uzrokovalo promjenu. Događaj može uzrokovati zahtjev korisnika, korisnički program ili operacijski sustav.
- Dodjelom određenog djelića vremena CP svakom programu u sustavu. Svaki program registriran u računalu dobiva određeni dio vremena CP. Ovakvi se OS nazivaju timesharing. OS. OS nadzire kružnu izmjenu programa u izvođenju, uz optimalizaciju korištenje mogućnosti računala.

Moguće je dodjeljivati procesorsko vrijeme i kombinacijom navedena dva načina.

Kod višekorisničkih sustava nužan je mehanizam zaštite korisnika i podataka od neautoriziranog korištenja, kao i zaštite OS od korisnika. Redoslijed izvršavanja programa može se regulirati i dodjelom različitih prioriteta različitim korisnicima, za to je u pravilu zadužen sistem inženjer.

Prevladavajući operativni sustavi koji su u širokoj uporabi na računalima opće namjene mogu se razvrstati u dvije osnovne skupine: tzv. "unixoidi" (odn. unix-u slični sustavi) i skupina varijanti sustava Microsoft Windows. Tzv. "mainframe" računala i drugi složeniji računalni sustavi upotrebljavaju drugačije operativne sustave, većinom dosta različite od Unix-a i MS Windows-a.

Operacijski sustav MS Windows

Po broju instalacija OS Microsoft Windows u različitim verzijama danas je sigurno "najpopularniji" operativni sustav. Pretpostavka je da se studenti snalaze u okolišu MS Windows-a u dovoljnoj mjeri za izvođenje nastave na vježbama, stoga smatramo da nije potrebno detaljnije obrazlagati ovaj OS.

Operacijski sustav UNIX

UNIX je višekorisnički, višeprogramski OS. Prva verzija razvijena je 1969 u "Bell laboratories". Od tada ovaj se OS (u različitim verzijama i pod raznim nazivima) primjenjuje na velikom broju hardwerskih platformi. Raličiti "derivati" UNIX-a nazivaju se obično "unixoidi", a na PC računalima danas se najviše koristi LINUX. Naglo širenje UNIX-a vezano je uz porast popularnosti samostalnih radnih stanica. Osnovni razlozi za popularnost UNIX-a su:

- Portabilnost - UNIX je u cijelosti realiziran u programskom jeziku C, što omogućuje instalaciju na velikom broju računala.
- Izvorni kod je poznat i dostupan - proizvođačima procesora je povoljnije preuzeti gotov i poznati OS od razvijanja vlastitog.
- UNIX zadovoljava sve zahtjeve koji se postavljaju pred bilo koji OS pružajući korisniku mogućnosti koje on može zahtijevati.

U osnovi UNIX se sastoji od dva dijela:

- jezgre OS koja se zove KERNEL, koja omogućuje rad različitih uređaja računala, nadgleda komunikaciju, te dodjelu memorije i redoslijed izvršavanja programa, i
- komunikacijske ljuske (SHELL) koja služi kao komandni interpreter, s kojom korisnik interaktivno komunicira.

Pod kontrolom ljuske korisnik koristi postojeće uslužne programe, komunicira s OS i razvija nove aplikacije. Iako UNIX izvorno nije namijenjen krajnjim korisnicima računala, već programerima upravljačkih sustava, informatička zajednica brzo ga je prihvatila. Razvoj UNIX-a divergirao je u različitim smjerovima, pa se ubrzo pojavio problem kompatibilnosti programskih proizvoda razvijanih pod različitim verzijama OS. Problem relativno složene komunikacije na radnim stanicama se rješavao razvojem standardnih elemenata korisničkiih sučelja s prozorima,

pri kojima se koriste grafički simboli - ikone, za pozivanje programa kojima se izvode standardne funkcije. Ovakva sučelja (danas opće prihvaćena) omogućuje jednostavniji rad i brže uhodavanje korisnika.
Neke od inačica operacijskog sustava UNIX: ULTRIX (1984.), AIX (IBM 1990.), Solaris – SUN OS - (Sun Microsystems 1991.), Linux (Linus Torvald 1991.)

Programski jezici (prevodioci)

Programski prevodioci nekad su bili praktički jedina ili prevladavajuća vrsta sistemske programske podrške – sve do intenzivnijeg razvoja računalne grafike i grafičkih programskih sučelja koja su omogućila upotrebu računala i za druge svrhe osim numeričkih proračuna i baza podataka. Kao kratak povijesni pregled, ovdje ćemo samo nabrojiti neke od važnijih (poznatijih) programskih jezika, odnosno ujedno i programskih prevodioca:

FORTRAN (FORmula TRANslator, IBM 1954.)
Algol 58, 60, 68 (ALGOrithmic Language 1958.)
COBOL (Common Business Oriented Language, CODASYL 1959.)
PL/1 (IBM sredina 1960-tih)
BASIC (1964.)
PASCAL (Niklaus Wirth) 1971.-
ADA (razvoj započeo 1975., dovršen 1995.)
C (Dennis M. Ritchie 1970.)
C++ (Bjarne Stroustrup)
Visual Basic, Delphi (Borland)
SmallTalk, Java (SUN)
PHP (Hypertext Preprocessor) – skriptni jezik

Razvojne okoline

Usporedno sa razvojem hardvera teče i razvoj sistemske i aplikativne programske podrške. Programski sustavi postaju izuzetno složeni, a isto tako i proces njihova razvoja. Pojam razvojne okoline obuhvaća zapravo skup programskih alata namijenjenih za razvoj složenih programskih sustava. U početku su razvojne okoline bile većinom građene oko jednog programskog jezika (npr. C, C++, Delphi, Visual Basic 5, Visual Basic 6) da bi danas imali i razvojne okoline sa "višejezičnom" platformom kakva je "Microsoft .Visual Studio .NET".
Microsoft Visual Studio .NET je kompletan skup razvojnih alata za izradu više različitih vrsta složenih mrežnih aplikacija i servisa kao i jednostavnijih "desktop" aplikacija. Ova razvojna okolina uključuje četiri programska jezika: Visual Basic .NET, Visual C++ .NET, Visual C# .NET i Visual J# .NET. Sva četiri jezika rabe zajedničku integriranu razvojnu okolinu koja omogućuje kreiranje složenih sustava čije su komponente napisane u različitim jezicima.
Microsoft opisuje .NET Framework kao "novu računalnu platformu kreiranu u cilju pojednostavljenja razvoja programskih aplikacija u visoko distribuiranom okruženju interneta".

Aplikacijska programska podrška

Obuhvaća programsku podršku pomoću koje primjenjujemo računalo u svim aspektima našeg djelovanja - omogućuje korištenje računala u svakodnevnim poslovima korisnika: razvoj novih programa i aplikacija, učenje, projektiranje, dopisivanje, upravljanje procesima, financije, uprava, zabava itd.

Primejri aplikativne programske podrške:

Programi za **obradu teksta**: MS Word, WordPerfect, TeX, HTML (Hiper Text Markup Language), ….
Programi za **obradu crteža i slika:** CorelDraw, Freehand, Corel Photo Paint, Paintbrush, Paint Shop Pro, MS Photo Editor, Photoshop

Programi za **prijelom i uređivanje publikacija:** Adobe Illustrator, ...
Programi za kreiranje i upravljanje **bazama podataka**: Clarion, dBase, Ingres, Informix, MS Access, Oracle, Paradox,…..
Tablični kalkulatori: Lotus 123, MS Excel
Antivirus programi: Norton Anti virus, Sophos, ….
Internet alati: Internet explorer, Netscape, MS FrontPage,….
Programi za **modeliranje i konstruiranje proizvoda**: AutoCAD, CATIA, IDEAS, PRO/Engineer, Solidworks,…..
Programi za rješavanje **matematičkih zadaća**: MATHCAD, Mathematica, Matlab, ….

Datoteke

Datokeka (eng. file) u računalnom sustavu je niz "bitova" spremljenih kao jedna cjelina, tipično u okviru datotečnog sustava (eng. file system) na tvrdom disku ili nekom drugom mediju za pohranu. Sadržaj datoteke može biti program napisan u nekom od programskih jezika (tekstualni zapis), ili program u strojnom jeziku (binarnom kodu), tj. izvršni oblik programa, podaci potrebni za izvođenje programa, podaci potrebni za razne evidencije, tekstovi, crteži itd. Premda datoteku prikazujemo kao jedan jedinstveni niz, najčešće se ona sprema u nekoliko fragmenata na različita mjesta na tvrdom disku (to pogotovo vrijedi za velike datoteke).
Jedna od osnovnih zadaća operativnog sustava je organiziranje pohrane i manipulacije s datotekama unutar datotečnog sustava. Način pristupa sadržaju datoteka prije svega određen je internim mehanizmima upravljačkog sustava, ali i programima kojim se datoteke kreiraju.
Datoteke kreiramo korištenjem programske podrške (softverom). Uobičajeno je da pojedina vrsta, odnosno proizvođač programa organiziraju zapis u datoteci po određenim pravilima koje nazivamo "format" datoteke (eng. file format).
Nekada je bilo uobičajeno da se datoteke definiraju kao niz slogova (eng. records), no to je danas više iziuzetak nego pravilo. Neki operativni sustavi (uglavnom starijeg porijekla) omogućuju da se sadržaj datoteke segmentira u tzv. slogove fiksne ili varijabilne dužine. MS Windowsi imaju samo jednu specijalnu klasu datoteka – tekstualne datoteke u kojima točno određeni niz znakova može separirati podatke u retke teksta (specijalna varijanta slogova varijabilne dužine). Unix ne manipulira sa slogovima datoteke na razini operativnog sustava, ali se to može raditi na razini aplikacija.

UVOD U ALGORITME I PROCES RAZVOJA PROGRAMSKE PODRŠKE

Pojam algoritma

Pojam algoritma osnovni je temelj računalnog programiranja, stoga je potrebno posvetiti posebnu pažnju analizi i razumijevanju algoritama.
U početku upotrebe pojma algoritma tako su se nazivala samo pravila računanja s brojevima, kasnije i pravila obavljanja ostalih zadataka u matematici. U XX stoljeću, pojavom računala, pojam se proširuje na računalstvo, a zatim i na druga područja.

Primjer Euklidovog algoritma za traženje najvećeg zajedničkog djelitelja dva cijela broja ***m*** i ***n***:

1. Odrediti ostatak od dijeljenja: podijeli ***m*** sa ***n***, i odredi ostatak od dijeljenja ***r***, $0 <= r < n$
2. Da li je ostatak jednak nuli? Ako jest, algoritam završava, ***n*** je traženi odgovor.
3. Ako ostatak nije jednak nuli, napravi zamjenu: postavi ***m*** = ***n***, ***n*** = ***r*** i vrati se na početni korak

Neke od varijacija definicije algoritma:

- Algoritam je precizan opis svih pravila i postupaka potrebnih za postizanje željenog rezultata.
- Algoritam je konačni skup pravila koja daju redoslijed operacija za rješavanje specifičnog problema.
- Algoritam je procedura (postupak) koji do posljednjeg detalja opisuje sve aktivnosti i njihov redoslijed, potreban da se obradom ulaznih podataka dođe do izlaznih podataka, odnosno rješenja.

Značajke algoritma

Osim što mora biti skup pravila algoritam mora posjedovati i slijedeća važna svojstva:

- **konačnost** – mora uvijek završiti nakon konačnog broja koraka
- **definiranost** – svaki korak algoritma mora biti precizno definiran – akcije koje treba poduzeti moraju biti u svakom slučaju rigorozno i nedvosmisleno (nedvojbeno) specificirane
- **ulaz** - algoritam može ali i ne mora imati ulazne veličine koje su dane inicijalno prije početka algoritma
- **izlaz** – algoritam ima jednu ili više izlaznih veličina koje su u specificiranom odnosu sa ulzanim veličinama
- **efikasnost** – se očekuje od svakog algoritma

Učinkovitost algoritma

Od algoritma se očekuje i da bude efikasan (učinkovit). To znači da sve operacije koje se obavljaju u algoritmu moraju biti dovoljno jednostavne tako da se mogu obaviti točno i u konačnom vremenu koristeći olovku i papir. Euklidov algoritam koristi samo operacije dijeljenja pozitvnih cijelih brojeva, provjere da li je broj jednak nuli, te dodjeljivanja vrijednosti jedne varijable drugoj. Navedene operacije su efikasne jer se cijeli brojevi mogu prikazati na papiru u konačnom obliku i jer postoji algoritam dijeljenja cijelih brojeva. Na primjer zbrajanje cijelih brojeva je učinkovito, ali zbrajanje realnih brojeva nije jer se može pojaviti broj s beskonačno mnogo znamenki. Algoritam koji bi izabirao potez igrača šaha tako da ispita sve moguće posljedice poteza, zahtijevao bi milijarde godina na najbržem zamislivom računalu.

Računalna procedura i računalni program

Postupak koji ima sva svojstva kao i algoritam, ali ne mora završiti u konačnom broju koraka jest **računalna procedura**. Primjeri za proceduru su operacijski sustav računala, uređivač teksta i sl. Vrijeme izvođenja procedure mora biti "razumno".

Računalni program je opis algoritma koji u nekom programskom jeziku jednoznačno određuje što računalo treba napraviti**.**

Programirati znači naučiti sintaksu nekog programskog (proceduralnog) jezika i steći osnovna intuitivna znanja glede algoritmizacije problema opisanog riječima.

Navesti ćemo ovdje mnogo puta citiranu izreku Niklausa Wirtha koji je koncipirao programski jezike Pascal i Modulu:

Algoritmi + strukture podataka = PROGRAMI

Programiranje na neki način možemo smatrati i vještinom i to vještinom koju nije lako naučiti. Potrebno je mnogo mukotrpnog rada (učenja) i stjecanja iskustva ("vježbanja") počevši od izrade jednostavnih prema kompleksnijim programima da bi se postalo "produktivnim" programerom. Nisu svi jednako talentirani u tom području, jer programiranje bi mogli okarakterizirati i kao "umjetnost razmišljanja i planiranja".

Temeljni problemi programiranja su:

- kako osmisliti algoritme
- kako strukturirati podatke
- kako formulirati algoritme
- kako verificirati korektnost algoritama
- kako analizirati algoritme
- kako provjeriti (testirati) program

Postupci izrade algoritama nisu jednoznačni te često zahtijevaju i veliku dozu kreativnosti. Ne postoje čvrsta pravila za definiranje algoritama. Od prvih programskih jezika nastoje se razviti metode i alati za formaliziranje algoritama. Napori su urodili različitim metodologijama prikaza problema i modela koji se koriste pri razvoju složenih programskih sustava.

Prikaz algoritma

Razvoj i prikaz algoritma preduvjet je izrade programa. Algoritmi se mogu prikazati na različite načine:

- opisno, rečenicama,
- grafički, dijagramom toka,
- u nekom jeziku koji je blizak čovjeku (pseudokod), ili
- nekim strogo formaliziranim programskim jezikom.

Međutim, program koji je zaista u računalu i po kojemu se izvodi neki konkretan postupak, uvijek je samo u binarnom (strojnom) obliku .

Stoga postoji potreba, da se na određeni uobičajeni način prikaže program tako da ima logičku ispravnost, ali i da je blizak i razumljiv čovjeku. Takav način prikaza je pseudokod koji nije ovisan niti o računalu niti o programskom jeziku, a značajan je stoga jer program napisan pseudokodom predstavlja zapravo model programa i najvažniji je rezultat rada u programskom inženjerstvu.

PRIMJER EUKLIDOVOG ALGORITMA U PASCALU I C-U:

```
program euclid(input.output);
var x.y: integer;
function gcd(u.v: integer): integer;
   var t: integer;
   begin
   repeat
      if u<v then
         begin t:=u; u:=v; v:=t end;
      u:=u-v
   until u=0;
   gcd:=v
   end;
begin
while not eof do
    begin
    readln(x.y);
    if (x>0) and (y>0) then writeln (x.y.gcd(x.y))
    end;
end.
```

```
# include <stdio.h >
main( )
{
   int x,y;
   while (scanf("%d  %d" ,&x , &y) != EOF)
      if ((x>0) && (y>0))
        printif (("%d  %d %d\n" , x , y , gcd (x , y));
}
int gcd (u , v)
int u , v;
{
  int t;
  do {
     if (u<v)
        {t = u; u = v; v = t; } ;
     u = u-v;
} while  (u ! = v);
return (u);
}
```

Simboli dijagrama toka

Prikaz algoritma dijagramom toka vjerojatno je ipak najpregledniji, pogotovo za programere početnike, stoga je uloga dijagrama toka nezaobilazna pri učenju programiranja. Dijagram toka sastoji se od simbola koji prikazuju različite vrste akcija: izvršavanje operacija, odlučivanje, učitavanje i ispisivanje podataka. Simboli dijagrama toka povezani su linijama koje prikazuju tijek odvijanja algoritma.

Slijedeća tablica prikazuje standardne simbole dijagrama toka:

Operacija (općenito). Prikazuje se jedna operacija ili više njih, ako je rezultat operacija promjene vrijednosti, oblika ili mjesta nekih informacija. Najčešće se ovdje zapravo upisuju naredbe za dodjeljivanje vrijednosti varijablama.

Aritmetička odluka (grananje). Instrukcija aritmetičke odluke izračunava aritmetički izraz na osnovu kojega se donosi odluka. Moguće su tri akcije ovisno o tome da li je vrijednost izraza manja, veća ili jednaka nuli. Simbol ima dakle jednu ulaznu putanju (granu) i tri izlazne.

Logička odluka (grananje). Prikazuje se operacija koja ispituje istinitost nekog izraza, nakon čega program odabire jedan put. Simbol ima jednu ulaznu putanju i dvije izlazne ("DA" i "NE", odn. istinitost i neistinitost logičkog izraza).

Ulaz podataka – obično se upisuje lista podataka (varijabli) čije se vrijednosti učitavaju

Izlaz podataka - obično se upisuje lista podataka (varijabli) čije se vrijednosti ispisuju na nekom izlaznom uređaju

Ulaz/Izlaz podataka

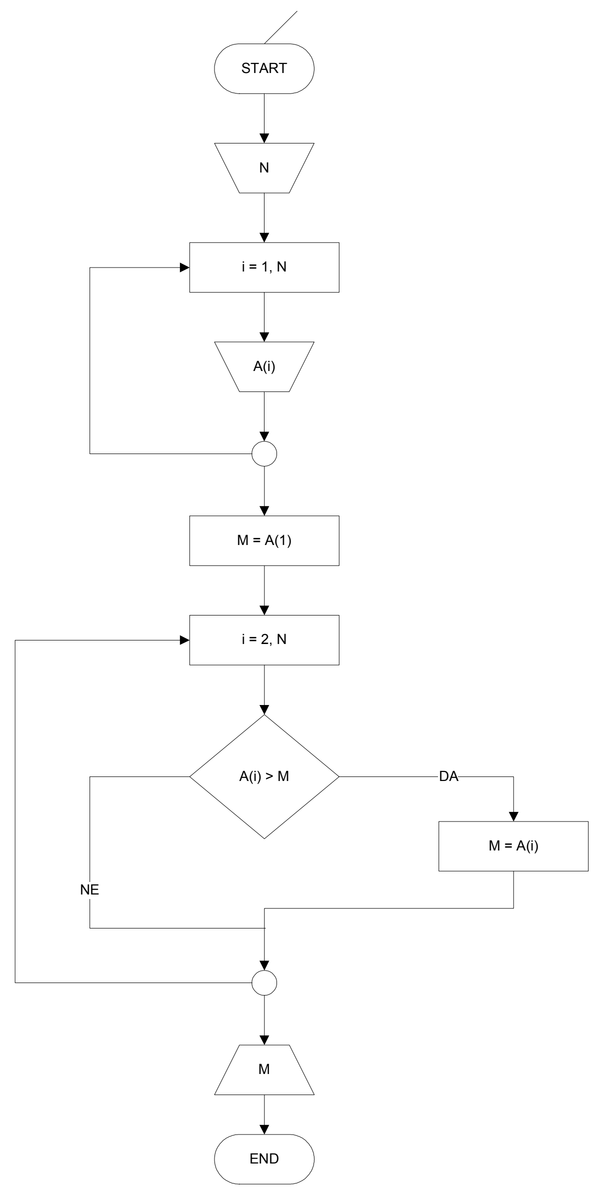
Crta toka odvijanja operacija programa prikazuje veze među simbolima u dijagramu. Radi jasnoće mogu se na crtama toka postaviti strelice usmjerene na simbol koje se kasnije izvodi. Prioritetni su smjerovi: a) odozgo prema dolje, b) slijeva udesno

Granično mjesto: početak (start), zaustavljanje (stop), prekid i slično.

Čvorište – mjesto spajanja više linija toka. U čvorište može ulaziti više linija, ali uvijek smije biti najviše samo jedna izlazna linija.

A

Povezivanje raznih dijelova u dijagramu toka. U kružić se upisuje neki simbol (brojka ili slovo). Izlaz iz nekog dijela dijagrama toka i ulaz u drugi dio, koji su međusobno povezani uz pomoć tog simbola, moraju imati istu oznaku. Više se izlaza može povezati s jednim jedinim ulazom.



Slika 5: Primjer dijagrama toka programa koji učitava niz brojeva te nalazi i ispisuje najveći član niza

Faze razvoja programa

Elektroničko računalo je stroj za obradu informacija, na osnovu programa koji je prethodno upisan u centralnu memoriju računala. Programiranje elektroničkog računala je složen proces koji zahtijeva egzaktno i precizno definiranje operacija kojima se obrađuje informacija, s ciljem da računalo izvršavanjem programa obavi željeni zadatak. Tijekom izrade računalnih programa, bez obzira koliko su oni jednostavni, potrebno je voditi računa o velikoj količini informacija i pravila neophodnih za ispravno funkcioniranje programa. Iz tog razloga proces izrade programa može se podijeliti na slijedeće faze:

1. Analiza problema
2. Postavljenje modela
3. Izrada algoritma
4. Izrada dijagrama toka
5. Kodiranje programa
6. Prevođenje programa
7. Testiranje programa
8. Dokumentiranje programa
9. Eksploatacija programa

Ove faze opisuju sve neophodne aktivnosti potrebne za izradu programa, ali u principu proces izrade programa je iterativan postupak, što znači da je vrlo često potrebno vratiti se na neki od prethodnih koraka i izvršiti potrebnu korekciju i modifikaciju.

Analiza problema

Elementarni uvjet koji mora biti ispunjen da bi se napravio računalni program je potpuno razumijevanje i poznavanje zadatka koji želimo isprogramirati. Odnosno, ukoliko nam nije jasno kako bi riješili zadatak bez računala (bez obzira koliko vremena nam treba za njegovo rješenje) definitivno ga je nemoguće riješiti pomoću računala, odnosno isprogramirati. Stoga je ANALIZA PROBLEMA prvi korak u izradi svakog programa. Ona se u principu sastoji od 3 osnovna segmenta:

- sagledavanje problema
- definiranje ulaznih informacija (podataka) i
- definiranje izlaznih informacija (podataka)

Prilikom sagledavanja problema bitno je uočiti sve relevantne činjenice i podatke koji opisuju problem, te uočiti koji su to ulazni podaci koje treba obraditi da bi došli do izlaznih podataka (rezultata). Evidentno je da pogotovo kod složenijih problema, u fazi analize nije uvijek moguće definirati sve ulazne i izlazne podatke. Stoga, u trenutku spoznaje da svi podaci nisu definirani, potrebno je ponoviti i nadopuniti analizu problema.

Postavljanje modela

Slijedeći korak u izradi programa je postavljanje modela, odnosno odabir metodologije pomoću koje se zadani problem može riješiti. U ovom izlaganju fokusiramo se na tipične inženjerske probleme čije rješavanje najčešće uključuje neki numerički proračun. Osnovna karakteristika ove faze izrade programa je potpuno razumijevanje zadanog zadatka te definiranje matematičkog modela, odnosno metodologije pomoću koje se zadatak može izvršiti. Model odnosno metoda za rješenje postavljenog problema proizlazi prije svega iz samog zadatka, te iz poznavanja:

- fizike problema
- tehničkih ili nekih drugih propisa i standarda
- matematičkih metoda i
- logike obrade ulaznih informacija s ciljem da se dobije izlazni rezultat.

U trenutku kada se postavi model, odnosno definira metodologija rješavanja problema, moraju biti definirani svi ulazni podaci, metodologija njihove obrade, te naravno koje su sve izlazne informacije.

Potrebno je naglasiti da nema recepta za izradu modela odnosno metodologije rješenja postavljenog zadatka. Kreiranje modela i metodologije obrade su umni i kreativni posao koje prije svega ovisi o poznavanju postavljenog problema. Ukoliko problem nije moguće riješiti pomoću postavljenog modela i metodologije "ručno" bez računala, model je neispravan i nije primjenjiv za programiranje na računalu.

Izrada algoritma

U ovoj fazi izrade programa rezultati analize problema i postavljenog modela se formiraju u formu procedure (recepta), koji opisuje sve neophodne operacije i njihov redoslijed nužan za izvođenje programa. Proces izrade algoritma uz malo iskustva je u biti rutinski posao, ali samo pod uvjetom da postoje ispravna analiza i model sa metodologijom rješavanja problema.

Izrada dijagrama toka

Dijagram toka programa je simboličko prikazivanje algoritma pomoću simbola dijagrama toka. Simboli dijagrama toka su standardizirani i u principu je moguće na osnovu algoritma predstavljenog simbolima dijagrama toka napisati program u bilo kojem programskom jeziku, odnosno dijagram toka je takva opća simbolička forma koja omogućava jednoznačno kodiranje

programa u nekom od programskih jezika. **Za programera početnika izrada dijagrama toka je važan korak kojeg nikako ne bi trebao zanemarivati.** Današnji programski jezici, tehnike programiranja i alati razvojnih okolina za iskusne programere nude dovoljno alternativa koje eliminiraju potrebu izrada dijagrama toka. Također, standardni simboli "klasičnog" dijagrama toka ne pokrivaju sve situacije koje nastupaju u današnjim tehnikama programiranja. U šezdesetim i sedamdesetim godinama proces razvoja programa bio je mnogo mukotrpniji nego danas, pa su dijagrami tokova bili od velike koristi pri "debugiranju" i održavanju programa.
Danas se razvijaju daleko složeniji programski sustavi kod kojih se dijagramski prikazi rade na višim razinama organizacije i strukture programskog koda, kao što su npr. UML dijagrami.

Kodiranje programa

Ova faza izrade programa je posljednja faza koja nije vezana za rad na računalu. Ona obuhvaća kodiranje programa opisanog dijagramom toka u neki od simboličkih programskih jezika.
Svaki simbolički jezik posjeduje svoj alfabet (niz znakova koji poznaje) i sintaksu (pravila po kojima se pišu instrukcije). Stoga je proces kodiranja programa u nekom od programskih jezika u biti prevođenje programa iz simbola dijagrama toka u programsku formu definiranu alfabetom, sintaksom i instrukcijama konkretnog programskog jezika.

Prevođenje programa

Važno je napomenuti da je općenito gledano za današnji stupanj razvoja računalnog programiranja teško dati "univerzalni" opis procesa prevođenja programa. Opis procesa koji slijedi stoga može dosta varirati u određenim detaljima za razne vrste programskih jezika, metode i tehnike programiranja, te integriranih okruženja i pomagala za razvoj programskih sustava. Upuštanje u opis svih specifičnosti pojednih slučajeva zahtijevalo bi previše prostora.

Prevođenje programa je prva faza izrade programa vezana za rad sa računalom. Sastoji se od dva, odnosno tri koraka ovisno da li se program napisan u nekom od simboličkih jezika izvršava pomoću INTERPRETER-a ili se prevodi u binarne strojne instrukcije pomoću COMPILER-a. Interpreteri su takvi računarski programi koji instrukcije simboličkog programskog jezika ne prevode u strojne binarne instrukcije već ih interpretiraju i odmah izvršavaju. Programi koji se izvršavaju pomoću interpretera u principu se lakše korigiraju i ispravljaju, ali zato se sporije izvršavaju. Programski jezici koji posjeduju prevodioce (compiler-e), omogućavaju prevođenje programa u strojni binarni kod koji je puno efikasniji i brži prilikom izvršavanja, ali kod njih je teže dijagnosticirati pogreške, a i sam postupak dobivanja izvršnog binarnog koda je nešto složeniji. U daljnjem tekstu će biti opisan postupak prevođenja simboličkih programskih jezika koji posjeduju prevodilac (compiler). Prvi korak kod prevođenja programa, i kod interpretera i kod compiler-a, je unošenje programa u računalo u izvornom obliku odnosno "source" kodu.
Nakon što je program spremljen u datoteku u izvornoj simboličkoj formi, evidentno je da takav program nije moguće izvršiti jer ne posjeduje strojne binarne instrukcije. Stoga je logičan i nužan slijedeći korak, a to je prevođenje programa u strojni binarni kod. Ovu operaciju obavljaju programi koji se nazivaju COMPILER-i, odnosno prevodioci. Posao compiler-a je višestruk i svaki compiler obavlja slijedeće funkcije:

- učitavanje instrukcije simboličkog programskog jezika,
- sintaktička analiza učitanih instrukcija i javljanje eventualnih pogrešaka,
- prevođenje sintaktički ispravnih instrukcija u strojni binarni kod, te
- spremanje prevedenog binarnog koda koji se obično naziva "object modul" u datoteku.

Na žalost, i ako se u object modulu nalaze prevedene strojne binarne instrukcije, object modul nije još spreman za izvođenje programa. Razlog za to se nalazi u činjenici da compiler u trenutku prevođenja ne zna kompletnu strukturu programa, pa nije u stanju definirati stvarne odnose memorijskih lokacija na kojima će se program nalaziti, stoga svaki program i pojedine rutine (potprogrami) u njemu, počne smještati od početka memorije. Rezultat toga je da se u object

modulu može naći više dijelova programa koji bi se istovremeno morali nalaziti u istim memorijskim lokacijama, što je nemoguće.
Treći korak u procesu prevođenja programa je povezivanje svih "object modula" u jedinstvenu cjelinu. To se obavlja pomoću programa koji se najčešće naziva "linker". Osnovne funkcije koje obavlja linker su:

- izrada memorijske mape programa,
- smještanje i povezivanje svih dijelova programa (rutina) na njihova stvarna mjesta u memoriji,
- ažuriranje memorijskih adresa iz "object modula" sa stvarnim adresama u skladu s memorijskom mapom, te
- spremanje ovako uređenog programa u datoteku koja se najčešće naziva izvršna verzija programa, ili popularno "exe verzija" programa.

Tek ovako pripremljen program može se izvršavati u memoriji računala jer se sastoji od strojnih binarno kodiranih instrukcija sa adresama stvarnih memorijskih lokacija na kojima se nalazi program i podaci.
U slučaju pronalaženja sintaktičke greške pri prevođenju programa, compiler nije u stanju generirati object modul, te se na osnovu poruke o grešci treba izvršiti korekcija, i novonastalu izvornu datoteku ponovo prevoditi. Druga vrsta grešaka su one koje otkriva linker prilikom povezivanja.

Testiranje programa

Ova faza izrade programa služi za provjeru odnosno verifikaciju programa - da li napravljeni program, kada se izvrši u stroju, obavlja postavljeni zadatak potpuno korektno. Nema generalnog postupka s kojim je uvijek moguće izvršiti apsolutnu verifikaciju svakog programa, pogotovo kod složenih programskih sistema, jer nije moguće pripremiti ulazne podatke testiranja i rješenja za njih za sve moguće kombinacije ulaznih podataka. Zbog toga se testiranje i verifikacija programa vrši za najkarakterističnije kombinacije ulaznih podataka za koje se znaju sva rješenja, ili ukoliko to nije moguće, za koje se znaju djelomična rješenja.

Ukoliko program ne zadovolji prilikom testiranja potrebno je otkriti što ne valja i ovisno o vrsti greške vratiti se na neku od prethodnih faza izrade programa. U ovoj fazi najčešće otkrivamo greške u algoritmima ili u modelu problema. Jedna vrsta tih grešaka se popularno nazivaju "bug-ovi". Grešku (posljedicu) je relativno lako uočiti prilikom testiranja, ali je redovito vrlo teško u složenijem programu pronaći uzrok greške, pogotovo početnicima i programerima s malo iskustva. U ovakvim situacijama dolaze do izražaja razlike u "kvaliteti" programiranja - organizaciji i načinu pisanja programskog koda. Razvijene su mnoge metode razvoja složenih programskih sustava koje nastoje eliminirati navedene kao i neke druge probleme održavanja i nadogradnje složenih programskih sustava (o tome detaljnije u poglavlju ####).
Nažalost većina današnjeg aplikativnog softvera je toliko složena da i pored vrlo opsežnih testiranja u toku razvoja ne budu otkrivene sve greške prije lansiranja na tržište. Redovita je pojava da greške otkrivaju tek korisnici tijekom eksploatacije programa i prijavljuju ih proizvođaču kroz unaprijed određene mehanizme komunikacije. Proizvođač tada izdaje tzv. "service pack"-ove kojima se zamjenjuju dijelovi programskog sustava u kojima su pronađene greške.

Dokumetiranje programa

Nakon što je program testiran i verificiran, potrebno ga je dokumentirati. Dokumentacija programa sastoji se iz dva osnovna dijela:

- tehnička dokumentacija (za programe sa uskom domenom primjene, najčešće za točno određene inženjerske probleme)
- korisnička dokumentacija

Tehnička dokumentacija se sastoji iz:
- opisa problema
- opisa modela i metode rješenja problema
- dijagrama toka i listinga programa (ako je tako ugovoreno)
- postupka instalacije programa
- postupka eksploatacije programa
- test primjera sa rezultatima

Korisnička dokumentacija je namijenjena korisnicima programa, i opisuje postupak korištenja programa, kako se zadaju ulazni podaci i kako se interpretiraju izlazni rezultati.

Eksploatacija programa

Bez obzira koliko to na prvi pogled izgledalo nelogično, eksploatacija programa je ujedno i jedna od faza njegovog razvoja. Praksa i iskustvo ukazuju da svi programi imaju svoj životni vijek, i da tokom svog "života" doživljavaju modifikacije i poboljšanja koje proizlaze iz iskustva stečenih tijekom njihove eksploatacije. Odnosno, drugim riječima rečeno, nakon određenog vremena potrebno je vratiti se u neku od prethodnih faza izrade programa, obično je to modeliranje ili izrada algoritma, i izvršiti njegovu korekciju i poboljšanje. Stoga je pri razvoju složenih programskih sustava uvijek (već od samog početka – postavljanja modela) potrebno voditi računa o svim aspektima održavanja, tj. daljnim modifikacijama i nadogradnji programskog sustava. Objektno orijentirano programiranje je metodologija čiji je razvoj pokrenut upravo zbog navedenih razloga, a biti će detaljno izložena u poglavlju ###. Danas je objektno orijentirano programiranje dominantna tehnologija u izradi složenih programskih sustava.

OSNOVNI KONCEPTI PROGRAMSKIH JEZIKA

Sintaksa i semantika programskog jezika

Proučavanje programskih jezika, slično kao i proučavanje prirodnih jezika može se podijeliti na *sintaksu* i *semantiku*.
Sintaksa programskog jezika je skup pravila i konvencija koji omogućuje formiranje korektnih programa sa stanovišta reprezentacije (prikaza). Sintaksa nema ništa sa značenjem niti sa "ponašanjem" programa u toku izvođenja, ali sintaksa je osnovni preduvjet za dobivanje (gradnju) značenja.

Elementi sintakse:
Sintaktičke jedinice najniže razine nazivaju se "leksemi" (eng. lexemes). Opis lexema može se dati u leksičkoj specifikaciji koja može biti odvojena od od sintaktičke specifikacije jezika. Računalni program se može promatrati i kao niz leksema i/ili kao niz znakova.
Leksemi programskog jezika uključuju identifikatore, konstante, operatore i specijalne riječi. Proces leksičke analize konvertira nizove znakova u jeziku u listu leksema. Leksemi s prosljeđuju tzv. "parser-u" na daljnju sintaktičku analizu. U računalnoj znanosti pojam "token" označava osnovni gramatički nedjeljivi element jezika - npr. kjučna riječ, operator, identifikator, itd. "Token" je kategorija lexema, u nekim slučajevima token i leksem su jedno te isto, npr. simbol aritmetičkog operatora.

Formalne metode prikaza sintakse

Osnovne formalne metode za opis sintakse programskih jezika su grafovi (dijagrami) sintakse i tzv. "context-free" gramatike (poznate i kao Backus-Naur forme (BNF).
Backus-Naur notacija (poznata kao BNF, tj. Backus-Naur forma) je formalna matematička metoda za opis jezika, koji su razvili John Backus i Peter Naur za opis sintakse programskog

jezika Algol 60. BNF se može promatrati i kao metajezik ili metasintaksa, odnosno formalni način za opis formalnih gramatika.

BNF specifikacija je skup "derivacijskih pravila" zapisanih kao:

```
<simbol> ::= <izraz sa simbolima>
```

Neka pravila sintakse ne mogu se zapisati sa BNF – npr. da sve varijable treba deklarirati prije nego su referencirane u programu

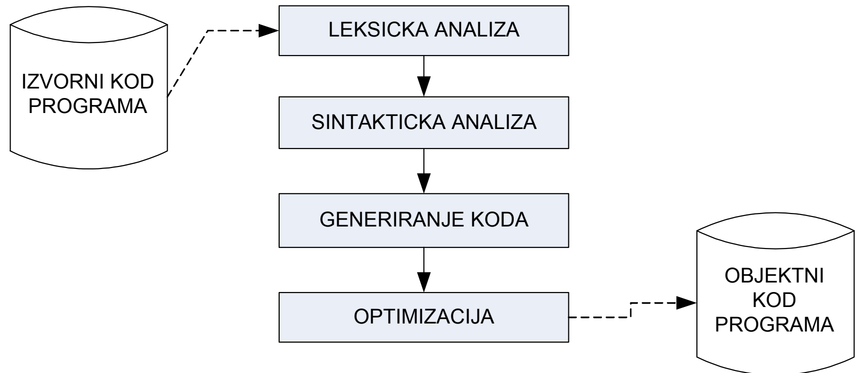
U računalnoj znanosti **semantika** programskog jezika je područje koje se bavi rigoroznim matematičkim proučavanjem *značenja* programskih jezika i računalnih modela.

Statička semantika programskog jezika bavi se "dozvoljenim" formama programa. U mnogim slučajevima statička semantička pravila određuju ograničenja na tipovima podataka. Naziv "statička" semantika je zbog toga što se provjera pravila odn. specifikacija može obaviti za vrijeme prevođenja ("kompajliranja") programa. Gramatika atributa (eng. attribute grammar) metoda je za formalni opis i sintakse i statičke semantike.

Dinamička semantika je specifikacija značenja, to je opis efekata izvođenja programa.
Opisati sintaksu programskog jezika relativno je jednostavno, za razliku od semantike. Npr. za dinamičku semantiku ne postoji općeprihvaćena metoda prikaza.
Za specifikaciju dinamičke semantike, uobičajene formalne tehnike su operacijske, aksiomatske i denotacijske metode. Sve navedene metode specifikacija semantike vrlo su kompleksne, stoga ih ovdje nećemo detaljnije obrazlagati.

Sintaksa, semantika i prevođenje programa

Kako se pitanja sintakse i semantike odražavaju na prevođenje ("kompajliranje") programa? Koncipiranje programa prevodioca (kompajlera) je vrlo kompleksan problem. Ovdje ćemo dati samo opću shemu osnovnih elemenata odnosno faza rada komplajlera u svrhu boljeg razumijevanja veza između sintakse, semantike i implementacije programskog jezika.



Slika 6: Osnovni elementi tipičnog programa prevodioca

Izvorni kod napisan u višem programskom jeziku prvo se podvrgava leksičkoj analizi čija je zadaća da prepozna osnovne "tokene" koji se pojavljuju u programu i da ih klasificira na konstante, identifikatore, rezervirane riječi, itd. Prva faza zapravo konvertira programski tekst u niz (listu) prepoznatih "tokena". Sintaktička analiza konvertira pretodnu listu u stablo "parsiranja", koristeći se internim prikazom gramatike jezika. Generiranje programskog koda zapravo je veza između sintakse i semantike (strojnog prikaza) jezika. U ovoj fazi stablo parsiranja konvertira se u ekvivalentni listu strojnih naredbi. Optimizacija nastoji poboljšati generirani kod u smislu smanjenja trajanja izvršavanja programa. "Objektni program" koji nastaje kao rezultat prevođenja može biti ili u strojnom jeziku ili u nekom "prelaznom obliku" do konačnog strojnog jezika koji se može izvesti na računalu.

"Parser" je algoritam odn. program za određivanje sintaktičke strukture rečenice ili niza simbola u nekom jeziiku. Kao ulazne podatke parser dobija niz "tokena" generiranih od "leksičkog analizatora". Kao izlaz parser može producirati tzv. stablo apstraktne sinatkse (eng. abstract sytax tree). U računalnoj znanosti "parsiranje" je proces analiziranja kontinuiranog niza znakova (pročitanog npr. sa tipkovnice ili iz datoteke) u cilju određivanja gramatičke strukture u odnosu na propisanu formalnu gramatiku. "Parser" je računalni program koji obavlja navedenu zadaću. Parsiranje transformira ulazni tekst u strukturu podataka, (obično struktura stabla), koja je pogodna za daljnje procesiranje.

Tipovi i strukture podataka u programskim jezicima

U računalnoj znanosti tip podatka je ime ili oznaka za skup vrijednosti i operacije koje se mogu obaviti na tom skupu vrijednosti. Programski jezici implicitno ili eksplicitno podržavaju jedan ili više tipova podatka. Tipovi podataka zapravo djeluju kao ograničenja u programima koja se statički ili dinamički provjeravaju. Osnovna ideja uvođenja tipova podataka je davanje značenja nečemu što je u konačnici zapravo samo niz bitova. Tipove obično povezujemo ili sa vrijednostima u memoriji ili sa objektima poput varijabli. Za računalo svaka vrijednost je jednostavno samo skup bitova, u hardveru nema razlikovanja između memorijske adrese, koda instrukcije, znakova, cijelih brojeva i decimalnih brojeva. Tip podatka govori nam kako treba tretirati taj niz bitova.

Programi se mogu promatrati i kao niz operacija koje se izvode na objektima - podacima. Tipovi objekata koje podržavaju pojedini programski jezici međusobno se razlikuju. Osnovni tipovi zajednički su većini programskih jezika koji se najčešće koriste, a razlike u definiranju (deklaracijama) istih tipova podataka i varijabli u različitim programskim jezicima nisu velike.
Temeljna svrha programa je obrada podataka pomoću računala, tako da je ishodišno pitanje programiranja opis i strukturiranje podataka.
Vanjski podaci, koji su definirani zadatkom, nisu jedini, jer niz podataka nastaje i za vrijeme obrade, postoje u memoriji i kasnije se nigdje ne vide. Zato možemo postaviti tri osnovne skupine podataka odnosno varijabli koje će postojati:

- ulazni podaci,
- izlazni podaci,
- unutarnji podaci programa.

Svaki podatak koji se obrađuje u nekom programu treba definirati i povezati tj. strukturirati u povezane cjeline (strukture podataka) kako bi jednostavno radili s njima.
Prema svojstvima pojedinih atributa koji opisuju entitete, podaci mogu biti tipa: cjelobrojni, realni broj, logička varijabla, znakovno polje ....
Sa stajališta struktura, podatke možemo povezati u: nizove (koji predstavljaju matematičke vektore i matrice), skupove, slogove, datoteke ....
Računalu je potrebno dati potpunu definiciju podataka kako bi se kontrolirale operacije nad njima: aritmetičke operacije dopuštene su nad brojevima, skupovne operacije nad elementima skupa itd.
Tip podataka (objekata) određuje i dozvoljene vrijednosti koje pojedini objekt može poprimiti, kao i skup dozvoljenih operacija. To ujedno ima i utjecaja na način pohrane podataka.

Osnovni (primitivni tipovi) podataka

Primitivni tipovi podataka, za razliku od kompozitnih, su tipovi podataka koje programski jezik nudi kao osnovne gradbene elemente. Ovisno o programskom jeziku i njegovoj implementaciji, primitivni tipovi mogu ili nemoraju imati "jedan prema jedan" korespondenciju sa objektima u računalnoj memoriji. Primitivni tipovi poznati su i kao "ugrađeni" tipovi (eng. built-in types) ili osnovni tipovi.

Tipični primitivni tipovi podataka uključuju: (navedene su oznake koje prevladavaju u većini jezika):

- ***niz znakova***: `character, char, string`
- ***cijeli broj***, sa nekoliko različitih područja vrijednosti : `integer, int, short, long`
- ***decimalni broj***, sa nekoliko različiti preciznosti: `float, real, single, double, double precision`
- ***logički*** (boolean), ima vrijednosti istinu ili laž (eng. **true** i **false**)
- ***referenca*** (eng. "pointer"), ili pokazivač (kazalo) – sadrži memorijsku adresu nekog drugog objekta

Ovdje je potrebno naglasiti razliku između preciznosti (eng. precision) i točnosti (eng. accuracy).
Preciznost broja određena je dužinom riječi procesora (32 ili 64 bita, najčešće 4 byta za prikaz integera odnosno realnog broja).
Preciznost se iskazuje brojem prvih važećih točnih znamenki, a točnost je bliskost stvarnoj (nepoznatoj) vrijednosti.
Za dovoljnu točnost potrebna je adekvatna preciznost, ali preciznost ne implicira automatski točnost jer su iskazane znamenke mogle nastati na temelju npr. pogrešnog mjerenja.

Neke od ključnih značajki programskih jezika:

- da li su deklarcije tipa podatka obavezne ili neobavezne
- da li su ograničenog dosega unutar programske jedinice (procedure, modula, bloka) ili ne?

Tipovi podataka u Visual Basic-u:

numerički: Integer, Long, Single, Double, Currency
tekstualni: String
logički: Boolean
"opći": Variant
datum: Date
"binarni": Byte

Varijable

Pojam varijable u programskom jeziku označava apstrakciju memorijske ćelije računala ili skupa memorijskih ćelija. Programeri često razmišljaju o varijablama kao imenima za memorijske lokacije (ćelije), premda pojam varijable obuhvaća i mnogo više. Programeru je mnogo prirodnije manipulirati sa simboličkim imenima memorijskih ćelija nego sa njihovim apsolutnim adresama u numeričkom obliku. Varijablu karakterizira slijedećih šest atributa:

- ime
- adresa
- tip
- vrijednost
- doseg (eng. scope)
- trajanje (eng. lifetime)

Određivanje imena varijable

Ime (naziv) je niz znakova upotrijebljen kao identifikator nekog entiteta u programu. Imena se u programima ne upotrebljvaju samo za varijable nego i za neke druge elemente (entitete) u

programu (potprograme, labele, formalne parametre, itd.). Pravila određivanja imena varijabli u različitim programskim jezicima uglavnom se razlikuju po slijedećim kriterijima:

- maksimalna duljina imena
- dozvoljeni znakovi u imenu varijable
- da li se razlikuju velika i mala slova ili se tretiraju kao isti znak ("case sensitive")
- da li su specijalne riječi rezervirane riječi ili kjučne riječi

Ključna riječ (keyword) – je riječ u programskom jeziku koja je specifična (specijalna) samo u određenom kontekstu.

Primjer ključne riječi u FORTRAN-u:

REAL APPLE	naredba deklaracije tipa varijable
REAL = 3.4	naredba dodjeljivanja vrijednosti varijabli REAL

Fortranski prevodioc prepoznaje razliku između imena i specijalnih riječi prema kontekstu.

Rezervirana riječ (reserved word) – je specijalna riječ programskog jezika koja se ne smije koristiti kao ime (identifikator).

To je bolja opcija nego ključne riječi (keywords) - u FORTRAN-u:

INTEGER REAL	deklaracija varijable imena "REAL" koja je tipa "integer"
REAL INTEGER	deklaracija varijable imena "INTEGER" koja je tipa "real"

Pravila imenovanja varijabli u Visual Basic-u:

- ime mora počinjati slovom abecede
- može sadržavati samo slova, brojeve i znak _
- ne smije sadržavati točku
- ime ne smije biti duže od 255 znakova
- ime mora biti jedinstveno u dijelu programa u kojem se nalazi varijabla.

Adresa varijable

Adresa varijable je adresa memorijske ćelije (lokacije) koju varijabla simbolizira (predstavlja). Takova asocijacija nije uvijek jednostavna kako se čini. U mnogim jezicima moguće je isto ime pridružiti (povezati sa) različtim adresama u različitim dijelovima programa. Za razumijevanje programskih jezika, od velike važnosti je i poznavanje trenutka u kojem se ime varijable povezuje sa memorijskom adresom.

Tip varijable

Tip varijable određuje područje vrijednosti koje varijabla može poprimiti kao i skup operacija koje se mogu obavljati na vrijednostima tog tipa. Na primjer u FORTTRAN-u cjelobrojni tip varijable koji koristi dva byte-a ima područje vrijednosti od -32768 do 32767. Na cjelobrojnom tipu podatka dozvoljene su operacije zbrajanja, oduzimanja, množenja i dijeljenja, te pozivi ugrađenih funkcija kao npr. apsolutna vrijednost.

Vrijednost varijable

Vrijednost varijable je sadržaj memorijske ćelije (ili više njih) koju varijabla simbolizira (predstavlja). Gledajući memoriju kao niz pojedinačno adresibilnih jedinica, u većini današnjih računala te jedinice su veličine "bajta" (byte) koji se sastoji od 8 bitova. Jedan "bajt" je premalena jedinica za zapis vrijednosti većine programskih varijabli. Stoga je običajeno govoriti o računalnoj memoriji u kontekstu "apstraktnih" ćelija, a ne "fizičkih" ćelija. Apstraktna ćelija sadrži odgovarajući broj "bajtova" za zapis vrijednosti određenog tipa varijable.

Doseg varijable

Doseg (eng. scope) je rang (područje) naredbi programa u kojima je varijabla "vidljiva". Varijabla je "vidljiva" u nekoj naredbi programa ukoliko može biti referencirana (poznata je njena adresa) u toj naredbi. Pravila dosega određuju kako je pojedino pojavljivanje imena

varijable u programu povezano sa varijablom. Posebno, pravila dosega određuju što se događa s referencama na varijable koje su deklarirane izvan procedure koja se trenutno izvodi.
Prema dosegu, varijble se obično dijele na:

- **globalne**
- **lokalne**

Globalne varijable su "vidljive", dostupne u svim procedurama, odnosno u jednom modulu koji sadrži više procedura. Lokalne varijable su deklarirane unutar procedure ili funkcije i dostupne su samo unutar te procedure ili funkcije. Način deklariranja globalnih varijabli dosta se razlikuje između programskih jezika.
Razlika lokalnih i globalnih varijabli može se vidjeti na primjeru dvije procedure u Visual Basic-u. Globalnim varijablama vrijednosti se dodjeljuju u trenutku pokretanja programa i one se nakon toga ne mijenjaju niti u jednoj proceduri. Loklanim varijablama vrijednosti se dodjeljuju u svakoj proceduri.

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim global1, global2 As Integer

        Private Sub Form1_Load()
        global1 = 50
        global2 = 80
    End Sub

    Private Sub prva_procedura_Click()
        Dim local1, local2 As Integer
        local1 = 100
        TextBox1.Text = CStr(global1)
        TextBox2.Text = CStr(global2)
        TextBox3.Text = CStr(local1)
        TextBox4.Text = CStr(local2)
    End Sub

    Private Sub druga_procedura_Click()
        Dim local1, local2 As Integer
        local2 = 200
        TextBox1.Text = CStr(global1)
        TextBox2.Text = CStr(global2)
        TextBox3.Text = CStr(local1)
        TextBox4.Text = CStr(local2)
    End Sub

End Class
```


Slika 7: Izgled forme nakon izvođenja prve procedure i nakon izvođenja druge procedure

Trajanje varijable

Trajanje varijable (eng. lifetime) je vremenski period u kojem varijabla zadržava svoju vrijednost dok se izvodi računalni program.
Doseg i trajanje varijable ponekad su povezani:

- Globalne varijable su postojane, traju koliko i izvodjenje cijelog programa i zadržavaju vrijednosti od jednog poziva potprograma (funkcije) do drugog.
- Lokalne varijable su deklarirane unutar funkcije (ili bloka) i traju smo dok se funkcija (blok) izvodi.
- Ako se želi sačuvati vrijednost lokalne varijable za slijedeći poziv procedure, treba ju deklarirati kao statičku – takvu mogućnost nemaju svi programski jezici.

Primjer deklariranja statičke lokalne varijable u Visual Basic-u:

```
Function Total(num)
    Static Suma
    Suma = Suma + num
    Total = Suma
End Function
```

Deklaracija varijable

Deklariranje varijabli i vrijeme kada se povezuju različiti atributi podataka važni su parametri snage, fleksibilnosti i efikasnosti nekog programskog jezika..
Općenito što je vrijeme povezivanja ranije (npr. za vrijeme prevođenja) moguće je generirati efikasniji kod. Nasuprot tome ukoliko je povezivanje kasnije fleksibilnost je veća.

Npr. izraz : y =y + 1
Naizgled trivijalno, ali treba povezati:

1. ime varijable i deklaraciju varijable,
2. deklaraciju i adresu,
3. adresu i vrijednost.

Pokazivači (kazala, "pointeri")

Ovaj tip podatka sadrži memorijsku adresu, odnosno referencu na drugi podatak. Pokazivači imaju dvije osnovne namjene:

- indirektno adresiranje (dosta se upotrebljava u programiranju na razini "assembler-a"
- kao metoda dinamičkog upravljanja alociranjem memorije

Jezici u kojima postoji tip podatka pokazivač (eng. pointer) obično uključuju i dvije osnovne operacije nad ovim tipom podatka:

- dodjeljivanje adrese neke varijable pokazivaču ("setiranje")
- dereferenciranje – dodjeljivanje ("uzimanje") vrijednosti varijable na koju pokazivač (pointer) pokazuje, odnosnu čiju adresu sadrži

Programski jezik C:

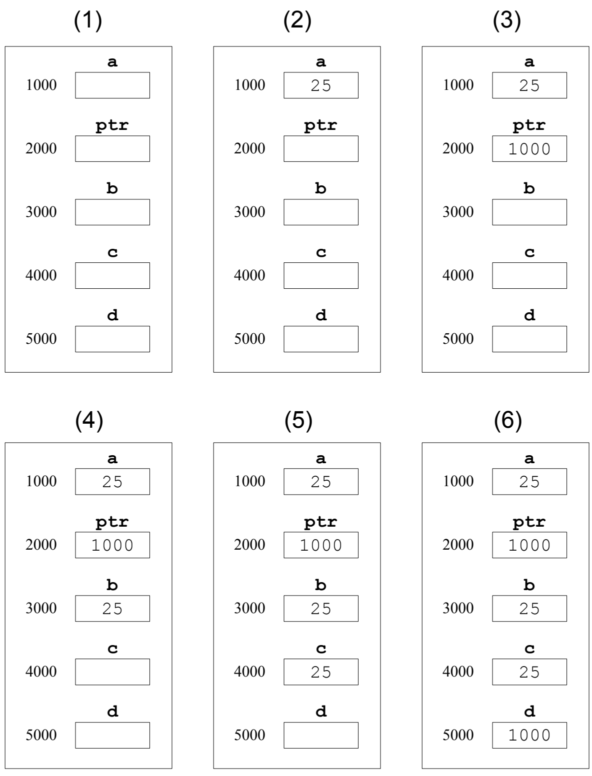
- & je operator dodjeljivanja adrese
- * je operator dereferenciranja

Programski jezik C ima i aritmetičke operacije nad pokazivačima (pointerima).

Za ilustraciju upotrebe pokazivača i operacija nad njima razmotrimo slijedeći primjer programa u C-u:

```
(1)  int a, *ptr, b, c, *d;
(2)  a = 25;
(3)  ptr = &a;
(4)  b = a;
(5)  c = *ptr;
(6)  d = ptr;
```

Na slijedećoj slici prikazana su stanje sadržaja pet memorijskih lokacija nakon izvođenja svake od naredbi programa. Naredbe programa označene su rednim brojevima u zagradama. Adrese memorijskih lokacija su: 1000, 2000, 3000, 4000 i 5000.



Slika 8: Sadržaji memorijskih lokacija u toku izvođenja primjera programa

Polja

Neki programski jezici omogućuju definiranje jednostavnijih struktura podataka, a gotovi svi jezici omogućuju definiranje strukture polja. Polje (eng. array) se koristi u situacijama kada treba manipulirati sa nizovima podataka (vektorima) ili sa tabličnim podacima odn. matricama. U takvim slučajevima daleko je pogodnije i razumljivije koristiti zajedničko ime za više memorijskih lokacija nego skup (niz) različitih imena.

Polje je:

- podatkovna struktura gdje isto ime dijeli više podataka
- sekvencijalni niz memorijskih lokacija kojima je pridruženo jedno zajedničko simboličko ime
- homogena agregacija podataka u kojima je jedan individualni element identificiran svojom pozicijom u odnosu na prvi element

Za polja vrijede i slijedeća pravila:

- Svi podaci u nekom polju moraju biti istog tipa
- Sva pravila imenovanja varijabli vrijede i za polja
- Elementi (članovi) polja se identificiraju *indeksom*
- Indeks određuje adresu elementa u polju
- U nekim jezicima je početna vrijednost indeksa 1 (npr. FORTRAN), a u nekima je početna vrijednost indeksa 0 (npr. C i Visual Basic)
- Indeks može biti nenegativni cijeli broj (konstanta, varijabla, cjelobrojni izraz)
- Polje može imati više indeksa, odnosno dimenzija
- Dimenziju polja određuje broj indeksa kojima se određuje pojedini podatak unutar polja.

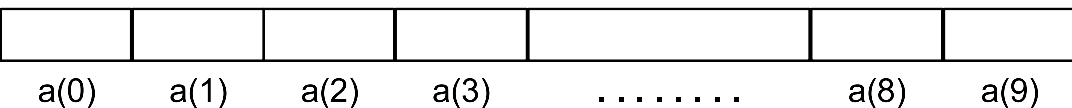
Primjeri određivanja indeksa:

x (0) x (9) x (n) x (MAX) x (n+1) x (k/m+5)

Primjer jednodimenzionalnog polja (vektora):

Visual Basic:

```
Dim a(10) As Integer
```



Jednodimenzionalno polje (vektor) koje ima 10 članova, zajedničko ime je "a", prvi član polja ima indeks "0", a zadnji član polja ima indeks "9".

Primjeri dvodimezionalnog polja (matrice):

(prvi indeks je redni broj retka, a drugi indeks je redni broj stupca!)

a(0,0)	a(0,1)	a(0,2)	....	a(0,n-2)	a(0,n-1)
a(1,0)	a(1,1)	a(1,2)	....	a(1,n-2)	a(1,n-1)
⋮	⋮	⋮	⋮	⋮	⋮
a(m-2,0)	a(m-2,1)	a(m-2,2)	....	a(m-2,n-2)	a(m-2,n-1)
a(m-1,0)	a(m-1,1)	a(m-1,2)	....	a(m-1,n-2)	a(m-1,n-1)

Slika 9: Dvodimenzionalno polje sa “m” redaka i “n” stupaca (poč. vrijednost indeksa: 0)

a(1,1)	a(1,2)	a(1,3)	....	a(1,n-1)	a(1,n)
a(2,1)	a(2,2)	a(2,3)	....	a(2,n-1)	a(2,n)
⋮	⋮	⋮	⋮	⋮	⋮
a(m-1,1)	a(m-1,2)	a(m-1,3)	....	a(m-1,n-1)	a(m-1,n)
a(m,1)	a(m,2)	a(m,3)	....	a(m,n-1)	a(m,n)

Slika 10: Dvodimenzionalno polje sa “m” redaka i “n” stupaca (poč. vrijednost indeksa: 1)

Deklariranje polja

U svim programskim jezicima potrebno je odrediti broj indeksa u polju (dimenzije polja) kao i ukupan (maksimalni) broj članova polja jer su to podaci neophodni za rezerviranje (adresiranje) memorijskih lokacija koje će polje koristiti pri izvođenju programa. Uobičajeno se takav postupak određivanja parametara polja naziva "deklariranje" polja.

Opći oblik naredbe za deklaraciju polja:

`tip_podatka ime_polja (veličina_polja)`

(redoslijed gore navedenih dijelova naredbe nije isti u svim jezicima, ali uvijek postoje sva tri navedena elementa naredbe)

Primjeri:

Visual Basic:

```
Dim a(10) As Integer
Dim b(50,50) As Single
```

C:

```
Float b[20][34];
```

FORTRAN:

```
REAL A(100,100)
```

Izrazi (eng. expressions)

Integralni dio svih imperativnih programskih jezika je koncept varijabli čija se vrijednost mijenja u toku izvršavanja programa. Vrijednosti varijabli mijenjaju se naredbom dodjeljivanja (eng. assignment statement). Može se reći da je naredba dodjeljivanja temeljna naredba za računala Von Neumann-ove arhitekture. Naredba dodjeljivanja u najjednostavnijem slučaju može kopirati sadržaj jedne memorijske ćelije u drugu ili memorijskoj ćeliji pridružiti konstantu, ali u većini slučajeva naredba dodjeljivanja uključuje aritmetičke ili mješovite izraze. Naredba dodjeljivanja općenito dakle sadrži izraz čiju vrijednost treba odrediti (izračunati), zatim operator dodjeljivanja i ciljnu lokaciju (varijablu) kojoj treba pridružiti vrijednost izraza.
Izraz u programskom jeziku je kombinacija vrijednosti, funkcija, token-a i procedura koji se interpretiraju prema pravilima redoslijeda i asocijacije čijom primjenom se izračunava i "vraća" vrijednost izraza. Pojam "token-a" objašnjen je u poglavlju o sintaksi prog. jezika.
Izrazi se sastoje se od jednog ili više operanada nad kojima djeluju operatori. Operandi mogu biti varijable i konstante. Operatori mogu biti unarni (imaju sam jedan operand) i binarni (imaju dva operanda):
Primjer:

-a*b¸ ovisno o situaciji – i + mogu biti i unarni i binarni: - a * b + c +x / y - z

Prema vrsti operanada i operatora izraze dijelimo na:

- aritmetičke
- relacijske
- logičke
- mješovite

Da bi razumjeli određivanje vrijednosti izraza (eng. expression evaluation) nužno je poznavati pravila redoslijeda izvođenja operatora i operanada.

Zagrade

Zagrade mijenjaju pravila redoslijeda izvođenja operatora. Dio izraza unutar zagrada ima viši prioritet izvršavanja od ostatka izraza izvan zagrada. U izrazu **`(A + B) * C`** prvo će se izvršiti zbrajanje, pa onda množenje. Ako izraz sadrži više ugnježđenih zagrada (jedne unutar drugih), zagrade se izvršavaju od unutarnjih prema vanjskim.

`( (A + B) / C ) + (D – E) / 2`

Izraz mora uvijek sadržavati jednak broj otvorenih i zatvorenih zagrada.

Aritmetički izrazi

Jedan od primarnih ciljeva prvih generacija programskih jezika bio je automatska evaluacija aritmetičkih izraza. Većina značajki aritmetičkih izraza u programskim jezicima naslijeđena je iz matematičkih konvencija. Aritmetički izrazi sastoje se od operatora, operanada, zagrada i poziva funkcija. Rezultat evaluacije (izračunavanja) aritmetičkog izraza uvijek je numerička vrijednost.

Redoslijed izvođenja operatora

Umjesto da se operatori izvode s lijeva na desno, u većini jezika određena su pravila redoslijeda izvođenja aritmetičkih operatora. U gotovo svim imperativnim jezicima najviši prioritet ima potenciranje, zatim množenje i dijeljenje, pa tek onda zbrajanje i oduzimanje. Ako više operatora ima jednaki prioritet, izvode se lijeva na desno.
Ako izraz sadrži dva uzastopna pojavljivanja operatora istog prioriteta, redoslijed izvođenja odeređuje se prema pravilima asocijativnosti – operator može imati ili lijevu ili desnu asocijativnost. Ako je lijeva, izvodi se prvi operator s lijeve strane, a ako je desna onda prvi operator s desne strane. Većina imperativnih jezika ima lijevu asocijativnost.

asocijativnost operatora	**Pascal**	**C**	**FORTRAN 77**
lijeva	svi operatori	*, /, %, binarni +, binarni -	***, /, +, -**
desna		**++, --, unarni +, unarni -**	**

Primjeri: **`A – B + C`** prvo se izvodi oduzimanje pa onda zbrajanje
`A ** B ** C` u FORTRAN-u se prvo izvodi desni operator potenciranja

Konverzija tipova podataka u izrazima

Većina programskih jezika dozvoljava da operatori u aritmetičkim izrazima imaju operande različitih tipova. Jezici koji dozvoljavaju takve aritmetičke izraze moraju definirati konvencije za implicitne konverzije tipova operanada zbog toga što računala obično nemaju operacije nad operandima različitih tipova. Takve implicitne (prisilne) konverzije inicirane od "compiler-a" u eng. računalnoj terminologiji obično se nazivaju "coercion in expressions".

Primjer aritmetičkog izraza sa varijablama različitih tipova (prog. jezik C):

```
int a; float b; double c;
c = a + b;
```

Implicitno se obično konvertira iz "nižeg" u "viši" tip , tj. viši tip bi trebao uključivati bar aproksimacije mogućih vrijednosti nižeg tipa: int - float - double

Sa razmatranog aspekta razlikujemo tri tipa programskih jezika:
- ne dozvoljavaju mješovite izraze (Ada, Modula-2), potrebna eksplicitna pretvorba npr. real(a) + b
- dozvoljavaju "razumne" kombinacije (npr. realni i cijeli brojevi, Pascal, FORTRAN)
- dozvoljavaju sve kombinacije (C, Visual Basic)

Ako programer želi eksplicitnu konverziju tipova podataka, gotovo svi jezici posjeduju ugrađene funkcije za tu namjenu.

Primjer nekih od funkcija za eksplicitnu konverziju tipova podataka u Visual Basic-u:

FUNKCIJA:	KONVERTIRA U:
CDbl	Double
CInt	Integer
CLng	Long
CSng	Single
CStr	String
CVar	Variant

Relacijski izrazi

Relacijski operatori uspoređuju vrijednosti dva operanda. Relacijski izraz ima dva operanda i jedan relacijski operator. Vrijednost relacijskog izraza je logička (istina ili laž), osim ako jezik ne poznaje logički tip podatka (npr. C).
Sintaksa relacijskih operatora u najčešće korištenim jezicima:

operacija	**Pascal**	**C**	**FORTRAN 77**	**Visual Basic**
jednakost	=	= =	**.EQ.**	=
nejednakost	<>	**!=**	**.NE.**	<>
veći od	>	>	**.GT.**	>
manji od	<	<	**.LT.**	<
veći od ili jednak	>=	>=	**.GE.**	>=
manji od ili jednak	<=	>=	.LE.	<=

Relacijski operatori uvijek imaju niži prioritet izvođenja od aritmetičkih.
U mješovitom izrazu `b + 3 > 2 * a` prvo će se izvesti aritmetički izrazi, pa tek nakon toga usporedba.

Logički (Boolean) izrazi

Sadrže logičke varijable i konstante, relacijske izraze i logičke operatore. Logičke varijable mogu poprimiti samo dvije vrijednosti – istinu ili laž (eng. true, false)

Logički operatori su:

- logička konjukcija AND
- logička disjunkcija OR
- negacija NOT

Ovisnost rezultata logičkih operacija o vrijednostima operanada obično se prikazuje tzv. tablicom "istinitosti" za logičke operatore (A i B su operandi – logičke varijable):

A	**B**	**A AND B**	**A OR B**
istina	istina	istina	**istina**
istina	laž	laž	**istina**
laž	istina	laž	**istina**
laž	laž	laž	laž

U većini jezika logički operatori imaju određen redoslijed izvođenja – najviši prioritet ima unarni NOT operator, pa zatim AND i nakon njega OR.

Programski jezik C nema logički tip podatka; numerička vrijednost 0 predstavlja laž (false), a sve ostale numeričke vrijednosti smatraju se istinom. Rezultat izvođenja logičkog izraza u C-u je integer sa vrijednošću 0 za laž ili vrijednošću 1 za istinu.

Mješoviti izrazi

Aritmetički izrazi mogu biti operandi relacijskih izraza, a relacijski izrazi mogu biti operandi logičkih izraza – sve zajedno tvoreći jedan izraz. Stoga moraju postojati i pravila redoslijeda izvođenja pojedinih vrsta operatora.
U većini jezika prvo se izvode aritmetički, zatim relacioni i na kraju logički operatori.

Primjer evaluacije mješovitog izraza:

Izraz: **`b/((a-b)*c) + a^2  >  a + (b-c)/3`**
Vrijednosti varijabli: **`a = 2, b = 3, c = 4`**
Redoslijed operacija:

```
a – b = -1
-1 * c = -4
a^2 = 4
b / (-4) = -0.75
-0.75 + 4 = 3.25
b – c = -1
-1 / 3 = -0.3333
2 + (-0.3333) = 1.6666
3.25 > 1.6666
```

konačna vrijednost izraza: **istina**

Naredba za dodjeljivanje (eng. assignment statement)

Kao što je već prije navedeno, naredba za dodjeljivanje (dodjeljivanje) jedan je od osnovnih elemenata imperativnih programskih jezika. Ona daje mehanizam za dinamičko mijenjanje vrijednosti varijabli. To znači da jedna te ista varijabla može sadržavati različite vrijednosti u tijeku izvršavanja računalnog programa. To je različiti koncept od nepoznanica (*x*, *y*, *z* ...) u algebri koje uvijek imaju istu vrijednost.

Opća sintaksa naredbe dodjeljivanja:

```
<varijabla>  <operator dodjeljivanja>  <izraz>
```

Navedena sintaksa vrijedi za osnovni (najjednostavniji) oblik naredbe koji se i najčešće koristi.

Primjeri:

```
A = B + C
suma = suma + clan_niza
K := K + 5
```

Naredba za dodjeljivanje izvodi se tako da se prvo izračuna vrijednost izraza s desne strane operatora dodjeljivanja. Zatim se ta izračunata vrijednost pridružuje (dodjeljuje) kao nova vrijednost varijabli s lijeve strane operatora dodjeljivanja. Pri tome nova vrijednost varijable zamjenjuje ("gazi, briše") staru (prethodnu) vrijednost.

U gotovo svim programskim jezicima operator pridružuvanja je znak jednakosti. Ta činjenica često uzrokuje zabune pri učenju programiranja, jer se događa da se naredba dodjeljivanja shvaća kao matematička jednadžba. Stoga ovdje moramo naglasiti da se naredba dodjeljivanja mora tumačiti isključivo kao naredba koja ima svoj redoslijed i korake izvršavanja, koji zapravo

nemaju ništa zajedničko sa pojmom jednadžbe osim simbola koji se koristi kao operator dodjeljivanja vrijednosti. Kao primjer navedimo sintaktički ispravnu naredbu: **`a = a + 1`** koja zapravo kaže: trenutnoj vrijednosti varijable "a" dodaj 1 i nakon toga zapiši to kao novu vrijednost varijable "a". Dakako "a = a + 1" nije ispravna jednadžba, odn. ne može biti jednadžba.

Kontrolne strukture na razini naredbi

Pored naredbe dodjeljivanja, potrebni su još neki mehanizmi (elementi jezika) koji će osigurati fleksibilnost i snagu izvođenja računalnog programa.
Naredbe programskog jezika normalno se izvode jedna za drugom - redoslijedom pisanja ako nije drukčije određeno. Većina programskih jezika posjeduje naredbe za kontrolu toka izvođenja (eng. control flow statements). Navedene naredbe omogućuju variranje redoslijeda izvođenja na slijedeće načine:

- Naredbe se izvršavaju samo pod određenim uvjetima
- Izvršavanje niza (grupe) naredbi se ponavlja više puta uzastopno
- Prelazi se na izvršavanje grupe naredbi izvan trenutnog programa u drugu programsku jedinicu (tzv. potprogram)

Kontrolnu strukturu čine kontrolne naredbe zajedno sa skupom drugih naredbi čije izvršavanje kontroliraju.

Kontrolne naredbe se obično dijele u tri temeljne skupine:

- **uvjetna grananja (selekcije),**
- **petlje (iteracije) i**
- **skokovi.**

Većina programskih jezika posjeduje inicijalnu ključnu ili rezerviranu riječ koja određuje vrstu kontrolne strukture (iznimka je Smaltalk). Jezike možemo podijeliti po tome da li kontrolna struktura ima ili nema "završnu" ključnu ili rezerviranu riječ koja označava kraj skupa naredbi koje čine kontrolnu strukturu.

- Jezici koji nemaju završnu rezerviranu ili ključnu riječ:
  Algol 60, C, C++, Java, Pascal, PL/1. Takvi jezici posjeduju druge načine grupiranja naredbi unutar kontrolne strukture:
    - Algol 60 i Pascal : **begin** ... **end**.
    - C, C++ i Java: vitičaste zagrade **{ ... }**.
    - PL/1: **DO** ... **END**.
- Jezici koji imaju završnu rezerviranu ili ključnu riječ:
  Ada, Algol 68, Modula-2, Fortran 77. Forme završne riječi su različite:
    - Visual Basic, Ada: završna riječ je end + razmak + inicijanlna riječ,
      npr: **if** ... **end if**, **loop** ... **end loop**.
    - Algol 68: inicijalna riječ napisana obrnuto, npr: **if** ... **fi**, **case** ... **esac**.
    - Fortran 77: **end** + inicijalna riječ, npr: **IF** ... **ENDIF**, **DO** ... **ENDDO**.
    - Modula-2: ima istu završnu riječ **end** za sve

Jezici koji posjeduju završnu ključnu ili rezrerviranu riječ imaju čitljiviji programski kod, pogotovo oni gdje je oblik završne riječi **end** + inicijalna riječ.

Bezuvjetni skok

Naredba bezuvjetnog skoka prenosi kontrolu izvođenja na specificirano mjesto (naredbu) u programu. Na taj način ova naredba daje veliku fleksibilnost, ali s druge strane prevelika fleksibilnost čini upotrebu ove naredbe opasnom. Bez određenih restrikcija u upotrebi programi mogu postati teško čitljivi ili gotovo potpuno nečitljivi, a kao posljedica toga vrlo nepouzdani i teški za održavanje. Stoga neki jezici niti ne sadrže ovakvu naredbu, a većina metoda i standarda programiranja preporuča upotrebu ove naredbe izbjegavati što je više moguće.
U većini jezika naredba bezuvjetnog skoka je "go to " naredba. Naredba na koju se prenosi kontrola izvođenja označava se sa "labelom" koja u nekim jezicima može biti ono što se inače koristi kao idetifikator ili neki jezici koriste cjelobrojne konstante.

<table>
<tr><th>Primjer u Visual Basic-u:</th><th>Primjer u FORTRAN-u:</th></tr>
<tr><td><pre>
     naredba_1
     naredba_2
     goto oznaka
     naredba_3
     ....
     naredba_n
oznaka: naredba_n+1
     naredba_n+2
</pre></td><td><pre>
          naredba_1
          naredba_2
          goto 100
          naredba_3
          ....
          naredba_n
100 naredba_n+1
          naredba_n+2
</pre></td></tr>
</table>

Uvjetna grananja (naredbe selekcije)

Naredbe za uvjetna grananja omogućuju upravljame tokovima podataka. Tijekom izvođenja programa, računalo će usmjeriti aktivnosti zavisno od stanja logičkih uvjeta koji su postavljeni u dijagramu toka podataka.

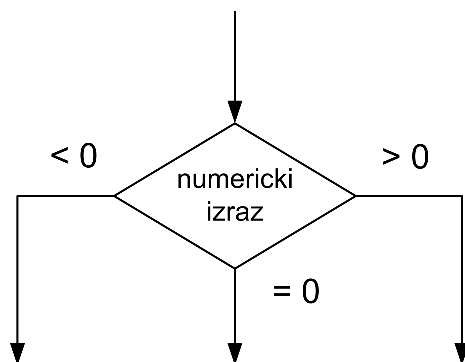
Odlučivanje na temelju vrijednosti numeričkog izraza

Ova naredba je iznimka jer koristi vrijednost numeričkog, a ne logičkog izraza. To je bila prva varijanta naredbe odlučivanja, uvedena u prvoj verziji FORTRAN-a. Vrijednost numeričkog izraza uspoređuje se s nulom, a izraz ima tri izlazne grane (uvjetna skoka): (<0, =0, >0).

Sintaksa za FORTRAN:

```
IF (numerički izraz) n1,n2,n3
```

n1, n2 i n3 su labele naredbi na kojima se nastavlja izvođenje programa, ovisno o vrijednosti numeričkog izraza. Za razliku od bezuvjetnog skoka, ovu naredbu možemo razmatrati i kao "uvjetni skok" jer se labele mogu nalaziti na bilo kojem dijelu programa sa izvršnim naredbama.



Slika 11: dijagram toka naredbe odlučivanja na temelju numeričkog izraza

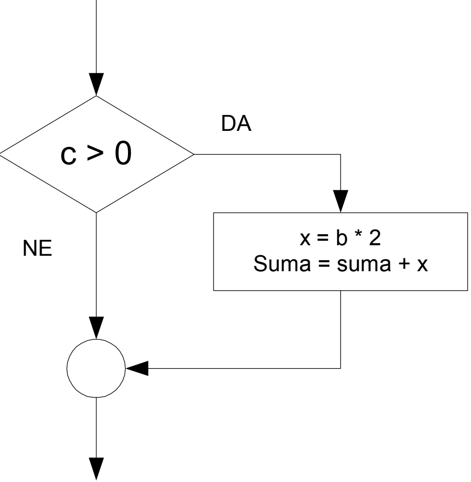
Odlučivanje (uvjetno grananje) – "Ako ..... Onda"

Oblik "Ako ..... Onda" (eng. If …. Then) je najjednostavniji oblik naredbe odlučivanja koji ima dvije izlazne grane, odnosno dvije mogućnosti izbora.
Odlučivanje odnosno izbor izvršava se na temelju vrijednosti logičkog uvjeta. Uvjet je logički izraz čija vrijednost može biti samo istina ili laž.
Mogućnosti izbora su slijdeće:

- Ako je vrijednost logičkog izraza *istina (true)* izvršava se jedna ili više naredbi iza rezervirane riječi **then**.
- Ako je vrijednost logičkog izraza *laž (false)* izvršava se slijedeća naredba iza rezervirane riječi **Endif**.

Ukoliko treba izvršiti samo jednu naredbu za istinitost izraza, nije potrebno pisati rezerviranu riječ **Endif** – tada se za vrijednost logičkog izraza *laž (false)* izvršava prva slijedeća naredba iza **If** naredbe.

<table>
<tr><th>Sintaksa (Visual Basic)</th><th>Primjer (Visual Basic)</th></tr>
<tr><td>If uvjet Then naredba</td><td>If a < b Then i = i + 1</td></tr>
<tr><td>If uvjet Then
     naredbe
Endif</td><td>If c > 0 Then
     x = b * 2
     Suma = suma + x
Endif</td></tr>
</table>



Slika 12: dijagram toka primjera naredbe uvjetnog grananja

Ovakav oblik uvjetnog grananja koristimo u slučajevima kada treba izvršiti jednu ili više naredbi za istinitost logičkog izraza, a u slučaju neistinitosti logičkog izraza ne treba napraviti ništa, nego dalje nastaviti sa normalnim slijedom programa.

Uvjetno granaje: – "Ako ..... Onda ..... Inače"

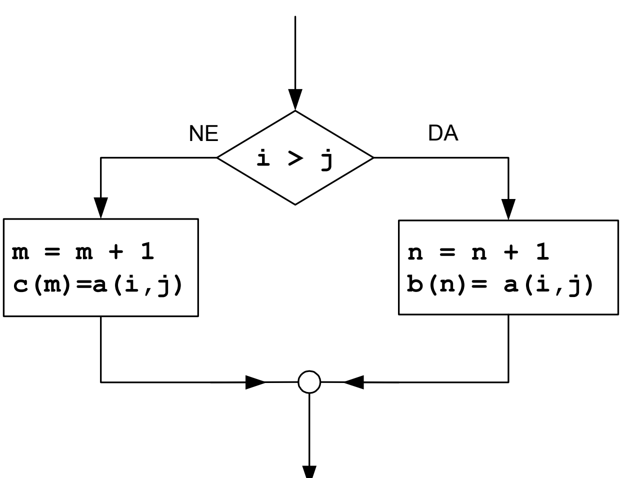
Oblik "Ako ..... Onda …. Inače" (eng. If …. Then …. Else) omogućava izvršavanje više naredbi (određenog niza naredbi) i za istinitost i za neistinitost logičkog izraza. Kao i u prethodno obrazloženom jednostavnijem obliku uvjetnog grananja iza rezervirane riječi **Then** slijedi niz (blok) naredbi koje se izvršavaju ako je vrijednost logičkog izraza istina. Kraj tog niza naredbi i početak bloka naredbi koji se izvršava ako je vrijednost logičkog izraza laž označava se

rezerviranom riječi **Else**. Završetak drugog bloka naredbi označava se rezerviranom riječi **Endif**.
Kao i u prethodnom obliku odlučivanje odnosno izbor izvršava se na temelju vrijednosti logičkog uvjeta.
Mogućnosti izbora su slijdeće:

- Ako je vrijednost logičkog izraza *istina (true)* izvršava se jedna ili više naredbi iza rezervirane riječi **then**, zatim se nastavlja sa izvođenjem naredbe iza rezervirane riječi **Endif**.
- Ako je vrijednost logičkog izraza *laž (false)* izvršava se jedna ili više naredbi iza rezervirane riječi **Else**, zatim se nastavlja sa izvođenjem naredbe iza rezervirane riječi **Endif.**

Sintaksa (Visual Basic)	Primjer (Visual Basic)

If uvjet Then
    blok_ naredbi_1
Else
    blok_ naredbi_2
Endif
	If i > j Then
	    n = n + 1
	    b(n) = a(i,j)
	Else
	    m = m + 1
	    c(m) = a(i,j)
	Endif



Slika 13: dijagram toka primjera naredbe uvjetnog grananja

Uvjetno grananje sa višestrukim ispitivanjem uvjeta

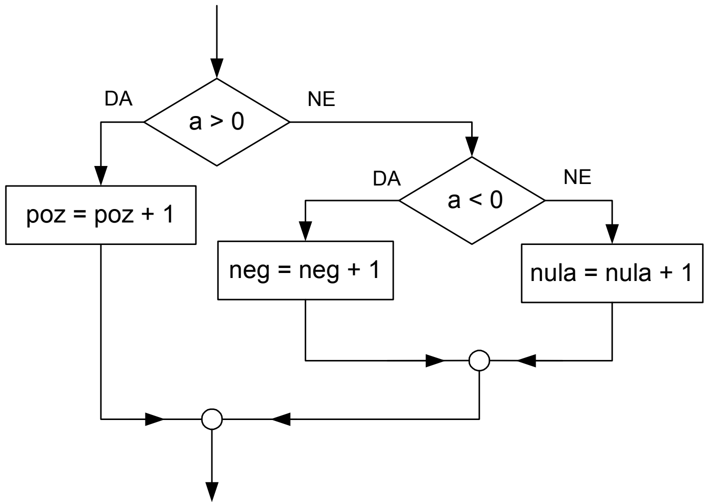
Rezervirana riječ **Elseif** omogućava izvršavanje višestrukih testova, odnosno ispitivanje više logičkih uvjeta jedan za drugim. Prelazak na ispitivanje slijedećeg logičkog uvjeta nakon rezervirane riječi **Elseif** može se ponavljati proizvoljni broj puta. Na taj način izbjegava se nepotrebno korištenje rezervirane riječi **Endif** na kraju ispitivanja svakog pojedinog uvjeta. Ovisno o potrebi, posljednji uvjet koji se ispituje može ili ne mora uključiti rezerviranu riječ **Else**, odnosno blok naredbi koji se izvršava ako i posljednji ispitani uvjet nije ispunjen.

Grananje odnosno izbor izvršava se na slijedeći način:

Redom se ispituju logički uvjeti, ako uvjet nije ispunjen (vrijednost laž), prelazi se na ispitivanje slijedećeg uvjeta u nizu. Izvršava se blok naredbi iza onog logičkog uvjeta čija je vrijednost istina, a nakon toga se nastavlja sa prvom naredbom iza rezervirane riječi **Endif** . Ako niti jedan od uvjeta u nizu nije istinit izvršava se blok naredbi iza rezervirane riječi **Else** (ako postoji) i nakon toga se nastavlja s prvom naredbom iza rezervirane riječi **Endif**.

Sintaksa (Visual Basic)	Primjer (Visual Basic)

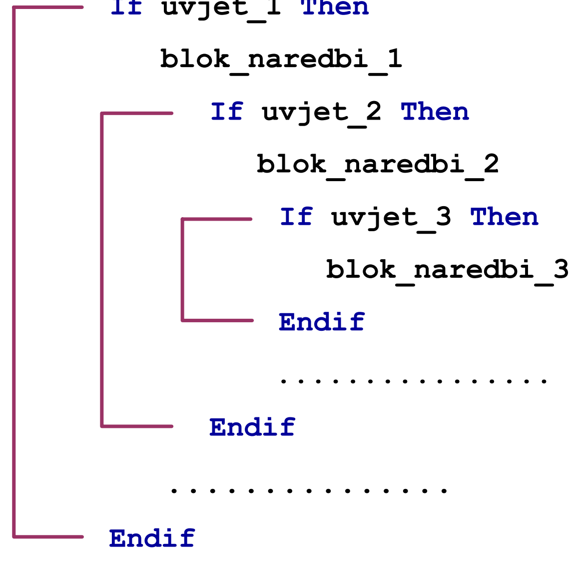
If uvjet_1 **Then**
  blok_naredbi_1
Elseif uvjet_2
  blok_naredbi_2
Else
  blok_naredbi_3
Endif
	If a > 0 **Then**
	 Poz = poz + 1
	Elseif a < 0
	 neg = neg + 1
	Else
	 nula = nula + 1
	Endif



Slika 14: dijagram toka primjera naredbe uvjetnog grananja sa višestrukim ispitivanjem uvjeta

Ugnježđenje kontrolnih struktura - uvjetna grananja jedno unutar drugog

U programiranju se često javlja potreba ispitivanja složenijih struktura logičkih uvjeta. Naredba uvjetnog grananja može se pojaviti kao jedna od naredbi unutar blokova naredbi druge naredbe uvjetnog grananja. Takva struktura naziva se "ugnježđenje" (eng. nesting). Moguće je "ugnijezditi" naredbe uvjetnog grananja i na više razina (Slika 15). Maksimalni broj razina ugnježđenja razlikuje se u različitim programskim jezicima. Pri kodiranju ugnježđenja treba paziti na to da svaka "if" naredba mora imati svoju "endif" naredbu, odnosno oznaku kraja. Svaki "endif" pripada najbližoj if naredbi, odnosno "zatvara"najbližu If naredbu. Običaj je da se sve naredbe na pojedinoj razini ugnježđenja pišu "uvučeno" radi bolje preglednosti strukture naredbi.



Slika 15: primjer "ugnježđenja" više naredbi uvjetnih grananja

Višegransko usmjeravanje

Naredba višegranskog usmjeravanja (**case** ili **switch**) služi za odabir jednog slučaja unutar skupa mogućih slučajeva. Temelj odabira je vrijednost izraza koji može biti numerički i tekstualni. Nakon izraza navode se slučajevi mogućih vrijednosti izraza. Naredba se izvršava tako da se prvo odredi vrijednost izraza, a zatim se ta vrijednost uspoređuje sa slučajevima unutar tijela naredbe. Ako je vrijednost izraza jednaka nekoj od vrijednosti navedenih iza pojedine "case" naredbe, tada se izvršava blok naredbi koji slijede, a nakon toga se nastavlja iza naredbe "end select". Ako niti jedna od navedenih vrijednosti ne odgovara vrijednosti izraza, izvršavaju se naredbe iza naredbe "Case Else". Ako nema potrebe, grana "Case Else" može se ispustiti. Iza svake "Case" naredbe može se navesti više vrijednosti izraza i tada se odvajaju zarezima. Ako se vrijednost izraza podudara sa više "Case" blokova, samo prvi blok naredbi će se izvršiti.

Opća Sintaksa	Primjer (Visual Basic)

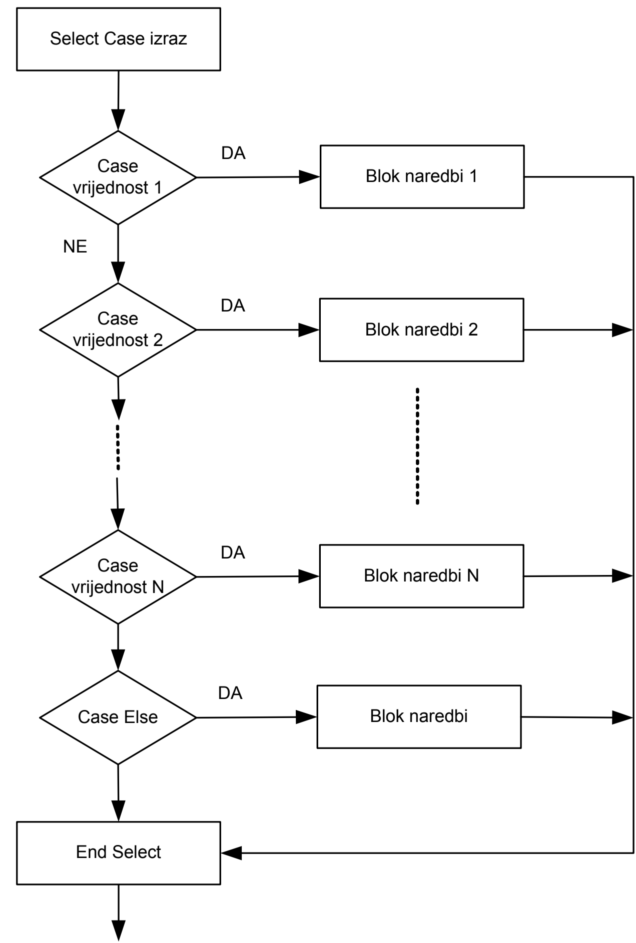
<pre>skretnica (izraz)
  slučaj C1
     blok naredbi1
  slučaj C2
     blok naredbi1
  slučaj C3
     blok naredbi4
. . . . .
  inače
     blok naredbi n
Kraj</pre>
	<pre>Select Case znak
	     Case A,E,I,O,U
	          vrsta = "veliki samog."
	     Case a,e,i,o,u
	          vrsta = "mali samog."
	     Case 1,2,3,4,5,6,7,8,9
	          vrsta = "znamenka"
	     Case Else
	          Vrsta = "nesto drugo"
	End Select</pre>

	Primjer (C)

	<pre>switch (nekiZnak) {
	   case 'a': actionOnA;
	      break;
	   case 'x': actionOnX;
	      break;
	   case 'y':
	   case 'z': actionOnYandZ;
	      break;
	   default: actionOnNista;
	}</pre>

Uočite razliku u označavanju početka i kraja kontrolne strukture između Visual Basic-a i C-a: u Visual Basic-u početak i kraj određuju rezervirane riječi `Select Case` i `End Select`, a u C-u je početak određen rezerviranom riječi `switch`, a grupa naredbi kontrolne strukture je omeđena vitičastim zagradama, tj. nema rezervirane riječi koja određuje kraj.

Ne postoji standardni simbol dijagrama toka za prikaz naredbe višegranskog usmjeravanja. Stoga je ova naredba prikazana kao struktura odlučivanja u kojoj se simbol za logičku odluku koristi za prikaz uspoređivanja izraza sa mogućim slučajevima.



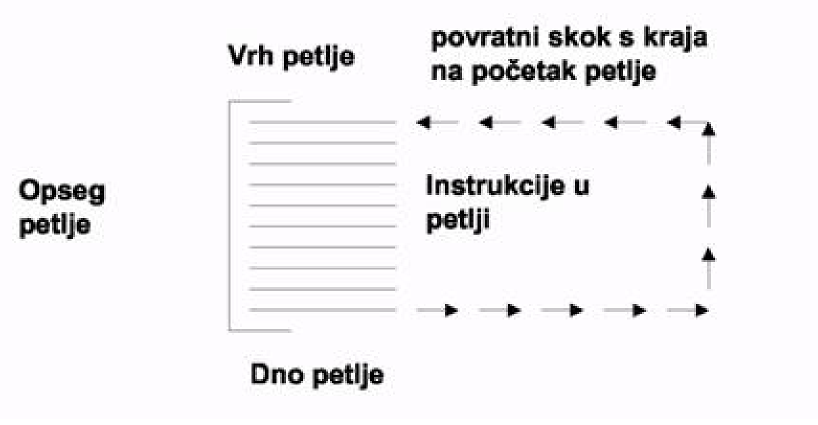
Slika 16: Grafički prikaz naredbe višegranskog usmjeravanja

Naredbe za ponavljanje izvođenja sekvenci programa (petlje)

Pojam petlje u programiranju označava sekvencu (niz) naredbi koji je specificiran jedamput, ali se može izvoditi više puta zaredom. Sekvenca može biti jedna naredba ili skup naredbi.
Petlje dakle omogućavju ponavljanje grupe naredbi unutar programa. Korištenje brzine rada računala manifestira se upravo u petljama. Ponavljanje izvođenja prekida se:

- nakon što je petlja izvedena onoliko puta koliko je to bilo određeno, ili
- ako je u toku izvođenja petlje zadovoljen uvjet koji je prethodno postavljen.

Provjera uvjeta petlje može biti na početku (vrhu), kraju (dnu) ali i u sredini petlje.
Sekvenca naredbi koje se ponavljaju uobičajeno se naziva tijelo petlje (eng. the *body* of the loop). Na slici 17 prikazani su uobičajeni pojmovi vezani uz petlje koji se koriste u programerskoj praksi.



Slika 17: Grafički prikaz pojma petlje u programiranju

U različitim programskim jezicima razvijeno je do danas dosta varijacija petlji, no sve se mogu svesti na dvije vrste:

- kontrolirane brojačima – koje se izvode točno određeni broj puta
- kontrolirane logičkim uvjetima koje se izvode neodređeni (unaprijed nepoznati) broj puta

Razlike u izvedbi petlji između različitih programskih jezika nisu velike, stoga ovdje nije dan prikaz svih varijacija.

Neka jednostavna pravila vrijede za konstrukciju svih vrsta petlji:

- petlje se ne smiju "zapetljati" – npr. ako je više petlji jedna unutar druge, kraj jedne petlje ne može biti unutar instrukcija druge petlje
- u petlju se ne smije "uskočiti", npr. naredbom bezuvjetnog skoka
- iz petlje se ne smije "iskočiti" bezuvjetnim skokom.

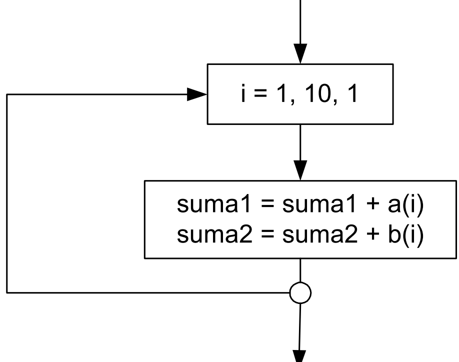
Petlje kontrolirane eksplicitnim brojačem

Ova vrsta petlji upotrebljava se u situacijama kada je u trenutku prije izvođenja petlje točno poznato koliko puta treba ponoviti izvođenje naredbu u tijelu petlje. Broj ponavljanja određuje se varijablom koja se zove "brojač petlje". Brojač petlje ima početnu vrijednost, krajnju vrijednost i korak. Sve tri vrijednosti moraju biti cijeli brojevi, a u većini jezika mogu biti i negativne.

Postupak izvođenja petlje:

1. Postavi brojač na početnu vrijednost
2. Provjeri da li je vrijednost brojača veća od krajnje vrijednosti:
   a. Ako jest – prekini izvođenje – "izađi iz petlje"
   b. Ako nije – izvedi sve naredbe u "tijelu" petlje
3. Povećaj brojač za korak, odnosno za 1 ako korak nije specificiran
4. Ponavljaj korake 2., 3., 4 (vrati se na korak 2.)

<table>
<tr><th>Sintaksa for petlje u Visual Basic-u:</th><th>Primjer for petlje u Visual Basic-u:</th></tr>
<tr><td><pre>For brojac = pv To kv [Step korak]
          naredba_1
          naredba_2
          ....
          naredba_n
Next [brojac]</pre></td><td><pre>For i = 1 To 10 Step 1
     suma1 = suma1 + a(i)
     suma2 = suma2 + b(i)
Next i</pre></td></tr>
</table>



Slika 18: Dijagram toka primjera petlje kontrolirane brojačem

Petlja "For Each .... Next"

Varijacija implementacije "For" petlje je "For Each" petlja koja ponavlja sekvencu naredbi za svaki element iz nekog skupa objekata ili za svaki element polja, dok ne obradi ("prođe kroz") cijeli skup odnosno polje. Takva naredba osobito je korisna ako nije poznat broj elemenata skupa ili polja.
Sintaksa "For Each"petlje u Visual Basic-u:

```
For Each element In group
          naredba_1
          naredba_2
          .......
          naredba_n
Next element
```

Primjer "For Each" petlje u Visual Basic-u: (Ova petlja ispisuje nazive svih tablica u kolekciji tablica koje čine bazu podataka).

```
For Each MyTableDef In objDb.TableDefs()
     List1.AddItem MyTableDef.Name
Next MyTableDef
```

Petlje kontrolirane logičkim uvjetima

Ova vrsta petlji upotrebljava se u situacijama kada je broj ponavljanja nepoznat, odnosno ne može se točno unaprijed odrediti. Petlja se kontrolira logičkim uvjetom, s time da postoje slijedeće varijacije:

- uvjet se ispituje na početku petlje, petlja se izvodi dok je uvjet istinit
- uvjet se ispituje na kraju petlje, petlja se izvodi dok je uvjet istinit
- uvjet se ispituje na početku petlje, petlja se izvodi dok je uvjet nije istinit
- uvjet se ispituje na kraju petlje, petlja se izvodi dok je uvjet nije istinit

Logički uvjet koji kontrolira petlju treba formirati tako da sadrži bar jednu varijablu čija će se vrijednost promjeniti u toku ponavljanja naredbi u tijelu petlje. Posljedica promjene vrijednosti te varijable (ili više njih) treba biti promjena vrijednosti logičkog uvjeta – sa istine na laž (ili obratno, ovisno o varijaciji petlje). Ako se logički uvjet ne formira na opisani način, velika je vjerojatnost da će se petlja izvršavati beskonačni broj puta, tj. neće biti prekida izvođenja petlje – u tom slučaju treba naredbom operacijskog sustava prekinuti izvršavanje programa.

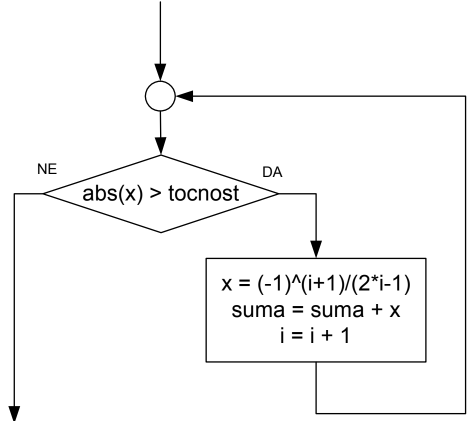
Petlja sa ispitivanjem logičkog uvjeta na početku

Logički uvjet se provjerava na početku petlje, a petlja se izvodi tako dugo dok je logički uvjet istinit. Ukoliko pri prvom ispitivanju ("ulazak u petlju") uvjet nije ispunjen, naredbe u tijelu petlje neće se izvršiti niti jednom. U Visual Basic-u takva petlja započinje rezerviranim riječima **Do While** a završava sa rezerviranom riječi **Loop.**

Sintaksa (Visual Basic)	Primjer (Visual Basic)

Do While logički uvjet
    naredba_1
    naredba_2
    ....
    naredba_n
Loop
	Do While Math.Abs(x) > tocnost
	        x = (-1)^(i+1)/(2*i-1)
	        suma = suma + x
	        i = i + 1
	Loop

Ne postoji standardni simbol dijagrama toka za prikaz ove vrste petlje, uobičajeno je ispred ispitivanja uvjeta staviti oznaku čvorišta, nakon toga simbol za ispitivanje logičkog uvjeta, te označiti povratni skok sa kraja petlje na čvorište iza kojeg slijedi ponovno ispitivanje uvjeta.



Slika 19: Dijagram toka primjera petlje sa ispitivanjem logičkog uvjeta

Petlja sa ispitivanjem logičkog uvjeta na kraju

Petlja s ispitivanjem logičkog uvjeta na kraju prvo izvodi sve naredbe u tijelu petlje, a potom kontrolira uvjet za izlazak iz petlje, odnosno prekid. Na taj način osigurava se bar jedno izvršavanje naredbi u tijelu petlje. U Visual Basic-u takva petlja započinje rezerviranom riječi `Do` a završava sa rezerviranim riječima `Loop While`.

Sintaksa (Visual Basic)	Primjer (Visual Basic)

Do
    naredba_1
    naredba_2
    ....
    naredba_n
Loop While logički uvjet
	Do
	    k = k + 1
	    b(k) = a(k) + 1
	Loop While m < k

Petlja sa ispitivanjem logičkog uvjeta koja se izvodi dok je uvjet neistinit

Varijacija petlje koja se izvodi tako dugo dok je logički uvjet neistinit u Visual Basic-u započinje rezerviranim riječima `Do Until` i završava sa `Loop` ako se uvje provjerava na početku petlje. Ako se uvjet provjerava na kraju petlje, petlja zaqpočinje sa rezerviranom riječi `Do`, a završava sa rezerviranim riječima `Loop Until`.

Provjera uvjeta na početku petlje	Provjera uvjeta na kraju petlje

Do Until logički uvjet
    naredba_1
    naredba_2
    ....
    naredba_n
Loop
	Do
	    naredba_1
	    naredba_2
	    ....
	    naredba_n
	Loop Until logički uvjet

Naredbe za izlaz iz petlje

Unutar tijela petlje mogu se koristiti i naredbe koje će prekinuti izvođenje petlje, tj. izvođenje programa će nastaviti s prvom naredbom iza kraja petlje. U nekim jezicima se za takve situacije mogu koristiti naredbe bezuvjetnog skoka, a u Visual Basic-u postoji naredba **`Exit For`** za izlaz iz "For" petlje i naredba **`Exit Do`** za izlaz iz petlji kontroliranih logičkim uvjetima. **`Exit For`** i **`Exit Do`** smisleno je koristiti unutar naredbe uvjetnog grananja ("If" naredbe) ili "Select Case" naredbe koje se nalaze unutar tijela pelje.

Ugnježđenje petlji - više petlji jedna unutar druge

U programiranju se vrlo često javlja potreba formiranja struktura u kojima se jedna ili više petlji nalazi unutar druge petlje. Takva struktura uobičajeno se koristi za "generiranje" svih kombinacija vrijednosti dvaju ili više skupova elemenata, odnosno nizova diskretnih vrijednosti.
Npr. za manipuliranje sa matricom (tablicom) koriste se dvije petlje – jedna "vrti" redni broj retka, a druga "vrti" redni broj stupca, da bi se "obradili" svi elementi matrice.
Takva struktura naziva se "ugnježđenje" petlji (eng. nesting). Moguće je "ugnijezditi" petlje i na više razina (Slika 15). Maksimalni broj razina ugnježđenja razlikuje se u različitim programskim jezicima. Pri kodiranju ugnježđenja treba paziti na slijedeće:

- Svaki početak petlje mora imati svoj kraj (npr. svaki "For" mora imati svoj "Next")
- Svaka "unutarnja" petlja (niže razine) mora završiti prije "vanjske" petlje (na višoj razini) – ne smije biti "preklapanja" "tijela" tih petlji
- Nije dozvoljen skok iz "vanjske" petlje u "unutarnju" – u tom slučaju ne dolazi do inicijalizacije brojača ili se ne ispituje uvjet kod petlji sa logičkim uvjetom
- "između početaka i krajeva" petlji može biti više naredbi ili niti jedna naredba

Uobičajena je praksa da se sve naredbe na pojedinoj razini ugnježđenja pišu "uvučeno" za nekoliko znakova, da bi na taj način struktura naredbi bila preglednija. "Ugnijezditi" se mogu sve prethodno opisane vrste petlji.

```
For br_1 = pv_1 To kv_1
      blok_naredbi_1
   For br_2 = pv_2 To kv_2
        blok_naredbi_2
       For br_3 = pv_3 To kv_3
           For br_4 = pv_4 To kv_4
             blok_naredbi_3
          Next br_4
       Next br_3
      blok_naredbi_4
   Next br_2
Next br_1
```

Slika 20: Ugnježđenje više petlji kontroliranih brojačem

Već je prije objašnjeno da kao i petlje, više naredbi uvjetnog grananja može biti jedna unutar druge, tj mogu biti "ugnježđene". Na isti način jedna ili više naredbi uvjetnog grananja može se smjestiti unutar petlje ili jedna ili više petlji može biti unutar naredbe uvjetnog grananja.

Vrlo je važno usvojiti i razumjeti ovakve načine kombiniranja kontrolnih struktura jer velika većina algoritama zahtijeva formiranje složenih "isprepletenih" struktura petlji i višestrukih ispitivanja uvjeta. Za rješavanje programerskih zadataka koji uključuju i najjednostavnije algoritme manipuliranja sa nizovima i matricama nužno je detaljno poznavati kontrolne strukture i naredbe, a posebno je važno znati sva pravila i mogućnosti kombiniranja odnosno ugnježđenja kontrolnih struktura. To je jedna od prvih i osnovnih stepenica koje se moraju savladati pri učenju programiranja.