Uvod

Kratak pregled razvoja programskih jezika

Računarski sistem se sastoji od hardvera i softvera. Hardver čine delovi računara, a softver programi koje računar izvršava. Računar je mašina bez inteligencije, on izvršava samo ono što mu je zadato i to na način na koji mu je zadato. Programom zadajemo računaru način na koji izvršava poslove. Da bismo rešili neki problem korišćenjem računara, moramo računaru precizno opisati sve korake-instrukcije (naredbe) koje on izvršava zadatim redosledom. Skup instrukcija napisan za rešavanje nekog problema naziva se program, a pisanje instrukcija programiranje.

Ljudi za međusobnu komunikaciju koriste jezik. Jezik je sredstvo za predstavljanje i prenošenje informacija. Čovek mora na neki način da saopšti računaru niz instrukcija koje treba da izvrši. Programski jezik je sredstvo kojim čovek saopštava računaru program. Čovek i računar komuniciraju pomoću programskog jezika. Prirodni jezici dopuštaju nejednoznačnost i nepreciznost. Računar može „razumeti" samo formalni zapis, ne tolerišući ni najmanje nepreciznosti. Programski jezici računaru omogućavaju zapis niza instrukcija koje se ne mogu višeznačno protumačiti. Jednoznačnost svake konstrukcije programskog jezika je njegova najbitnija karakteristika. Programski jezik je skup pravila kojim se računaru predstavljaju instrukcije i opisuju podaci.

Programske jezike možemo podeliti po stepenu zavisnosti programskog jezika i računara na:

- mašinski jezik
- simbolički jezik
- jezici višeg nivoa

Mašinski i simbolički jezici su zavisni od računara i njih nazivamo mašinski zavisnim jezicima. Jezici višeg nivoa pripadaju skupini koje nazivamo mašinski nezavisni jezici.

Mašinski zavisni jezici

Najjednostavniji programski jezik je interni jezik računara – mašinski jezik. Najvažniji delovi računara izgrađeni su od elektronskih i magnetnih komponenti, koje karakterišu dva stabilna stanja (ima napona – nema napona, teče struja - ne teče struja, severni pol – južni pol). Zbog jednostavnije tehničke realizacije u računaru se sve predstavlja pomoću dva simbola, 0 i 1, binarnom azbukom. Jedan znak binarne azbuke

predstavlja bit (skraćeno od **bi**nary digi**t**). Sve informacije u računaru predstavljaju se nizom bitova grupisanim u grupe od 8, 16, 32, 64 bita. Grupa od 8 bitova naziva se bajt.

Mašinski jezik je izgrađen nad binarnom azbukom. Sve operacije i svi podaci predstavljeni su nizovima bitova. U zavisnosti od arhitekture računara, ti nizovi bitova imaju različita tumačenja i različite su veličine.

Kako se mašinski program sastoji od niza nula i jedinica i zahteva dobro poznavanje načina rada i arhitekture određenog računara, vrlo je teško programirati na njemu. Programi na prvim računarima bili su zapisani mašinskim jezikom što je uslovilo da uzak krug ljudi piše i održava programe.

Da bi se premostile poteškoće pisanja programa na mašinskom jeziku, dolazi do razvoja simboličkih jezika. Umesto instrukcija pisanih nizom bitova, uvedene su mnemotehničke skraćenice za operacije i simboličke oznake podataka, npr. naredbom ADD a, b vrši se sabiranje podataka a i b. Na taj način proces programiranja je u znatnoj meri olakšan, ali i dalje zavisi od konkretnog procesora, tj. i dalje je potrebno poznavati tehničke karakteristike konkretnog računara za koji pišemo program. Da bi se program napisan na simboličkom jeziku izvršavao na računaru, mora se prethodno prevesti na mašinski jezik. Kako svakoj naredbi simboličkog jezika odgovara jedna naredba mašinskog jezika, posao je automatizovan tako što je napisan program koji kao ulaz dobija program napisan u simboličkom jeziku, a kao izlaz odgovarajući program na mašinskom jeziku. Program koji vrši prevođenje iz simboličkog u mašinski jezik naziva se asembler (translator). Zato se simbolički jezik često naziva asemblerski jezik ili kraće asembler. Napomenimo da skup naredbi simboličkog jezika zavisi od arhitekture računara, pa program napisan u simboličkom jeziku za jedan računar ne može se koristiti za računar druge arhitekture, već se mora ponovo pisati. Često za mašinske i simboličke jezike kažemo da su mašinski zavisni jezici. Važno je istaći da pri programiranju na mašinski zavisnim jezicima programer mora vrlo detaljno, u malim koracima opisati rešavanje problema. Možemo to uporediti sa procesom gradnje kuće pri čemu moramo voditi računa o položaju svake cigle, daske i slično. Jasno je da prilikom izgradnje kuće u jednom momentu moramo detaljno opisati položaj svake cigle, tj. elemente kuće predstavljati na taj način. Međutim, pri izradi samog projekta, lakše i razumljivije je govoriti o većim jedinicama, npr. soba, kuhinja, vrata, prozor i slično. Kako to izgleda kada koristimo računare upoznaćemo na sledećem primeru. Pretpostavimo da treba da saberemo četiri broja. U višim programskim jezicima to radimo korišćenjem jedne instrukcije (npr. x=a+b+c+d), a kada koristimo simbolički jezik, prvom instrukcijom šaljemo prvi broj u poseban registar (akumulator) a zatim sledećom instrukcijom dodajemo drugi broj sadržaju registra i tako dok ne dodamo i poslednji broj. Poslednja instrukcija je premeštanje konačnog rezultata iz registra u memoriju.

Programski jezici višeg nivoa

Da bi se premostile poteškoće u programiranju na asemblerskim jezicima, veoma rano počinju da se razvijaju mašinski nezavisni jezici, drugim rečima jezici višeg nivoa. Korišćenjem jezika višeg nivoa opis naredbi i podataka vrši se na način blizak prirodnom (engleskom) jeziku. U ovim jezicima jednoj naredbi odgovara više naredbi simboličkog jezika. Važno je napomenuti da ovi jezici imaju visok stepen nezavisnosti u odnosu na arhitekturu računara i operativni sistem na kojem se izvršavaju.

S obzirom na to da računar razume samo program napisan na mašinskom jeziku, svaki program pisan jezikom višeg nivoa mora se prevesti na mašinski jezik. Na osnovu načina prevođenja i izvršavanja, jezike višeg nivoa delimo na kompajlerske i interpreterske jezike. Najpre su nastali kompajlerski jezici Algol, Fortran, Cobol, PL/I... Kod ovih jezika izgrađuju se programi za prevođenje (kompajleri) kojim se ceo program napisan na višem programskom jeziku prevodi u njemu ekvivalentan, mašinski program koji se može izvršavati na računaru. Kod interpreterskih jezika, program na višem programskom jeziku se prevodi i izvršava instrukcija po instrukcija. Primeri interpreterskih jezika su Lisp, Prolog, Basic, ...

U početku se razlikovala primena računara u oblasti poslovanja od primene računara u nauci i tehnici. Prvu je karakterisao veliki broj ulazno/izlaznih podataka i relativno jednostavan opis obrade podataka, pa su i jezici namenjeni toj primeni zadovoljavali te karakteristike (Cobol...). Za primenu u nauci i tehnici karakterističan je mali broj ulazno/izlaznih podataka, ali i veoma složen opis obrade, pa su razvijani jezici za te namene poput Fortrana, Algola,... U to vreme programski jezik PL/I bio je dovoljno dobar za primenu u poslovanju kao i za primenu u oblasti složenih numeričkih izračunavanja. Današnjim razvojem jezika gubi se ova podela i savremeni programski jezici mogu se koristiti ravnopravno u svim oblastima.

Prema načinu rešavanja problema, možemo izvesti podelu viših programskih jezika na proceduralne i deklarativne programske jezike. U proceduralnim jezicima jezikom dajemo računaru kompletan skup instrukcija kojim se rešava problem, tj. dajemo mu algoritam za rešavanje zadataka. Ovim jezicima opisujemo **kako** se rešava dati problem. Manje-više svi poznatiji viši programski jezici su ovog tipa: Fortran, Cobol, Basic, Pascal, C kao i mašinski zavisni jezici. Kasnije, u razvoju programskih jezika dolazi se do ideje da se problemi opisuju, a da sam interpreter ima ugrađene postupke kako da reši opisani problem što dovodi do razvoja deklarativnih jezika. Ovim jezicima opisujemo **šta** znamo o problemu i šta želimo da dobijemo rešavajući ga, a sistem (interpreter) sam dolazi do postupka za rešavanje problema. Primeri deklarativnih jezika su Prolog i SQL. Sistem ima ugrađen algoritam koji dovodi do rešenja. Postojanje i pisanje opšteg algoritma za nalaženje rešenja je glavna poteškoća u razvoju ovih jezika, pa su zato deklarativni jezici obično specijalizovani za određene vrste problema. Koliko god da je ova ideja napredna, u praksi je vrlo teško u potpunosti sprovesti deklarativnost.

Neke deklarativne jezike možemo podeliti na relacijske i funkcijske. Osnovni objekti u prvima su relacije, a u drugima su funkcije. Ideja razvoja funkcijskih jezika je u spajanju unapred definisanih funkcija, koje imaju svoje ulaze i izlaze, u cilju dobijanja rešenja problema. Programer koji koristi takav programski jezik gleda na razvoj programa kao na način spajanja elementarnih funkcija, čime se dobija sistem koji izračunava željeni rezultat. Primeri funkcijskih jezika su LISP, LOGO, ML.

Posebnu klasu čine objektno orijentisani jezici, kod kojih su prisutni i proceduralni i neproceduralni elementi. U proceduralnim jezicima isticani su postupci, način realizacije (procedura), neke više ili manje složene akcije. Podaci su na neki način u podređenom položaju u odnosu na algoritam. Moramo priznati da su podaci razlog postojanja programa. Svaki postupak obrađuje neke podatke (čita ih, prikazuje, menja...). Podaci i akcije koje se izvode nad njima predstavljaju jednu nerazdvojnu celinu, jer nam podaci bez postupaka kojim ih obrađujemo, kao i postupci za obradu bez podataka, ne znače puno. Kod objektno orijentisanih jezika podaci i postupci su objedinjeni u jednu celinu i čine aktivan objekat, za razliku od podataka u proceduralnim jezicima koji su krajnje pasivni. Svaki takav objekat izgleda kao mali računar: on ima unutrašnje stanje i ima operacije koje možemo zahtevati da izvrši. Jasno možemo uočiti analogiju između takvih objekata i objekata u realnom svetu.

Posmatrajmo niz imena učenika jednog odeljenja. Kod proceduralnog programiranja niz imena se posmatra kao skup pasivnih podataka. Potom pišemo razne programe za rad sa ovim nizom u cilju njegovog abecednog uređivanja, modifikovanja (npr. dodavanje novog učenika, brisanje postojećeg učenika...). S jedne strane imamo podatke, a sa druge, od njih odvojene postupke (procedure) kojima obrađujemo te podatke. Kod objektno orijentisanih jezika, prethodni primer posmatra se objedinjeno: postoji objekat niz koji s jedne strane ima skup imena, a sa druge strane skup akcija koje se mogu primenjivati nad tim imenima i jedno od drugog je neodvojivo. Program koji koristi objekat niz ne mora imati nijedan od postupaka za obradu niza jer su oni ugrađeni u sam objekat.

Mnoštvo različitih objekata možemo uočiti u okruženju grafičkih operativnih sistema (GUI, Graphic User Interface). Ikone su objekti. Svakoj je pridružen skup osobina i akcija koje opisuju kako reaguju na spoljašnje događaje, šta se dešava kada jednom kliknemo, kada dva puta kliknemo, kad pritisnemo desni taster, kako se ponašaju kada držimo pritisnut taster na mišu i slično.

U objektno orijentisanom programiranju cilj programera je da sakupi podatke i akcije u neku vrstu „kutije sa alatom" koju će koristiti za brzi razvoj aplikacija. Primeri objektno orijentisanih jezika su SmallTalk, Java, C#. C++ je hibridni jezik; jeste iz klase objektno orijentisanih jezika ali nije čisto objektno orijentisan.

Iz ovog kratkog prikaza razvoja programskih jezika možemo uočiti da objektno orijentisanim pristupom na prirodan način opisujemo realan svet. Ipak, prirodnost opisa nije glavni razlog ozbiljnijeg razvoja objektno orijentisanih jezika, već je to „softverska kriza" nastala osamdesetih godina. U to vreme, razvoj hardvera doveo je do

znatnog povećanja primene računara u raznim oblastima, pa su samim tim čitavi timovi programera morali da rade na razvoju jednog projekta. Svaki deo tima razvijao je relativno nezavisne module, a zatim su se ti moduli uklapali u funkcionalnu celinu, korisničku aplikaciju. Svaki od modula moguće je koristiti u različitim aplikacijama. Na taj način dolazi se do standardizacije aplikativnog softvera.

S druge strane, pored razvijanja objektno orijentisane metodologije u programiranju, razvojem interneta javila se i potreba da napisani projekti rade identično na različitim platformama (operativnim sistemima...). Razvojem programskog jezika Java, koji je primer objektno orijentisanog pristupa, na neki način se prevazilaze različite platforme na kojima bi se softver izvršavao. Java tehnologija izvorno je bila zamišljena za prenos multimedije mrežom raznovrsnih uređaja.

Ponovimo ukratko:

Program u proceduralnom jeziku je niz naredbi koji određuje KAKO se određena akcija obavlja (npr. „otvori datoteku, ako nije EOF pročitaj podatak, ..., zatvori datoteku")

Program u neproceduralnom jeziku je niz naredbi koji određuje ŠTA treba učiniti (npr. „izdvoji podatke ... koji zadovoljavaju dati uslov...")

Program u objektno orijentisanom jeziku možemo shvatiti kao skup objekata koji između sebe i sa spoljnim svetom komuniciraju putem „poruka".

Algoritmi

Rešavanje problema primenom računara

Da bismo rešili neki problem korišćenjem računara, neophodno je kreirati program u nekom programskom jeziku. Taj proces je kompleksan i u njemu možemo izdvojiti sledeće faze:

- Postavka problema

  Problem koji se rešava treba precizno i potpuno opisati na prirodnom jeziku. To radi korisnik kome je namenjen program u saradnji sa programerima. Jasno se određuje cilj, analiziraju se postojeće informacije i vrši se izbor neophodnih početnih informacija, opisuju se svi podaci.

- Analiza problema

  U ovoj fazi definišu se ulazni i izlazni podaci, ograničenja njihovih vrednosti i precizno definišu veze između podataka. Rezultat ove faze je formalan opis problema, izrada matematičkog modela koji se može realizovati na računaru.

- Razrada algoritma

  U ovoj fazi se definiše način na koji se od početnih, ulaznih podataka dolazi do traženih, izlaznih. Većina problema može se rešiti na više načina, zato treba proanalizirati sve načine i izabrati optimalan. Složenost rešenja problema možemo posmatrati sa aspekta broja potrebnih operacija (vremena potrebnog za izvršavanje algoritma), i sa aspekta potrebnih memorijskih resursa da se algoritam izvrši (prostorna složenost algoritma). Dokaz o korektnosti izabranog postupka za rešavanje problema neophodno je precizno izvesti. O ovim aspektima treba posebno voditi računa pri rešavanju složenih problema, dok ih kod jednostavnih problema zanemarujemo.

- Projektovanje opšte strukture programa

  U ovoj fazi vrši se izbor programskog jezika. Razmatra se ceo problem i deli se u logičke celine; svaka celina se dalje deli u zavisnosti od njene složenosti. Precizno se definiše način čuvanja informacija, tj. definiše se struktura podataka.

- Kodiranje

  U ovoj fazi opisuju se, u nekom programskom jeziku, podaci i postupak rešavanja problema, koji su definisani u prethodnim fazama. Program zapisan u nekom programskom jeziku često se zove kôd, zato se ova faza i naziva kodiranje. Ako smo u prethodnim fazama precizno razradili algoritam, strukturu programa i strukturu podataka, ova faza je relativno jednostavna za izvođenje.

- Faza prevođenja, izvršavanja i testiranja programa

  Podsetimo se da se na računaru može izvršavati samo program koji je napisan pomoću binarnih cifara (na mašinskom jeziku). Zato je potrebno program napisan na programskom jeziku višeg nivo prevesti na mašinski jezik; prevođenje se vrši pomoću posebnih programa: kompajlera, odnosno interpretera. Da bi se program izvršavao, potrebno je formirati izvršnu verziju (EXE verzija). To se postiže pomoću specifičnih programskih alata koji se razlikuju od jezika do jezika (povezivač, bilder). Često su faza prevođenja i faza pravljenja izvršnog fajla objedinjene. Testiranjem programa treba proveriti da li program rešava postavljeni zadatak. To je vrlo važna faza u kojoj treba otkriti eventualno skrivene greške. Za otkrivanje grešaka koristimo testove kojima treba da obuhvatimo sve moguće situacije, svakako i one kad postavljeni problem nema rešenja. Po potrebi se vraćamo na prethodne faze.

- Izrada dokumentacije

  U dokumentaciji programa opisuje se šta program radi, daju se uputstva kako se koristi, detaljnije se opisuje algoritam kada su u pitanju složeniji problemi i slično. Dokumentaciju nikako ne treba započinjati kada je pro-

gram već napisan, već je treba pisati tokom izrade programa, ali je svakako treba dopuniti na kraju kada je program završen.

- Održavanje programa

  Tokom korišćenja programa mogu se uočiti neke greške, koje se moraju otkloniti. Ponekad treba izvršiti neke promene u programu usled novih okolnosti. Ako je program čitko napisan i ako ima dobru dokumentaciju, onda je ova faza jednostavna, kako za same autore programa, tako i za druge programere. U suprotnom mnogim programerima je lakše da napišu novi program nego da prepravljaju već postojeći. U ovoj fazi dolazi i do proširivanja programa dodavanjem novih funkcionalnosti.

Kada se razvija veliki projekat, sve prethodno navedene faze su jednako važne. Osposobljavanje programera za uspešno realizovanje velikih projekata je dugotrajan proces. U edukaciji polazimo od jednostavnih primera ka složenijim. U ovom procesu problemi koje rešavamo su jednostavni pa su sve faze, osim faze razrade algoritma, jednostavne.

Pojam i karakteristike algoritma

Termin algoritam potiče od imena arapskog matematičara Al Horezmija (oko 825. godine nove ere). On je izvršavanje aritmetičkih operacija prikazivao u obliku uputstava koja se sastoje od osnovnih koraka. Takav način opisivanja rešenja od tada se u matematici naziva algoritam. Daljim razvojem nauke algoritam je postao jedan od osnovnih pojmova u matematici i računarstvu.

Algoritam je konačan niz jednostavnih koraka kojim se opisuje postupak rešavanja određenog realnog problema, ali takav da se posle konačno mnogo vremena postupak završava.

Mnoge realne situacije se odvijaju po tačno određenim koracima tako da možemo reći da su algoritmi sastavni deo svakodnevnog života. Postoji relativno precizan postupak po kome pokrećemo automobil, telefoniramo, prelazimo ulicu, kuvamo kafu... Čak možemo reći da se i sam život odvija po nekim pravilima. Dobar primer algoritma je svaki kulinarski recept; u njemu jasno razlikujemo šta je ulaz (potrebni sastojci), šta je obrada (precizno opisan postupak za kuvanje) i šta je izlaz (gotovo jelo).

Algoritam možemo u najopštijim crtama prikazati slikom.

Primer 1:

Posmatrajmo postupak telefoniranja:

i) Podižemo slušalicu.

ii) Biramo broj.

iii) Razgovaramo.

iv) Prekidamo vezu.

U opisanom postupku možemo uočiti niz nedostataka. Šta ako nemamo odgovarajući signal prilikom podizanja slušalice, ili ako je linija zauzeta posle biranja broja? Šta u slučaju kada uspostavimo vezu a pozvani korisnik se ne javlja? To su situacije koje prethodnim opisom postupka nisu predviđene.

Realne situacije su po pravilu znatno složenije i da bismo ih opisali, neophodan je precizniji pristup svim detaljima. U procesu rešavanja određenih problema, često se susrećemo sa uslovima koji određuju dalji tok rešavanja. Zato je pri izradi algoritma neophodno omogućiti takozvane uslovne korake. Neke nedostatke uočene u prethodnom opisu postupka telefoniranja možemo prevazići na sledeći način:

i) Podižemo slušalicu.

ii) Ako nemamo odgovarajući signal, kraj postupka (neuspešno telefoniranje).

iii) Biramo broj.

iv) Ako se pozvani korisnik javi, razgovaramo.

v) Prekidamo vezu, kraj postupka.

Opisani postupak otklanja neke od navedenih nedostataka, ali i dalje ostaju nerazjašnjene mnoge situacije. U postupku telefoniranja uobičajeno je da u slučaju zauzeća linije ponavljamo postupak dok se veza ne uspostavi ili dok ne odustanemo od poziva. Potreba za ponavljanjem delova procesa vrlo je česta u realnim situacijama. Zato je neophodno pri opisu algoritma koristiti takozvane ciklične korake. Jedan od načina kako ciklični korak možemo ugraditi u prethodni primer je sledeći:

i) Podižemo slušalicu.

ii) Ako nemamo odgovarajući signal, kraj postupka (neuspešno telefoniranje).

iii) Biramo broj.

iv) Ako dobijamo signal zauzeća, poništavamo poziv i ponavljamo korak iii.

v) Ako se pozvani korisnik javi, razgovaramo

vi) Spuštamo slušalicu, kraj postupka.

U prethodnom opisu postupka telefoniranja, koraci **iii** i **iv** čine ciklus u kome je korak **iii** telo ciklusa (korak koji se izvršava pri svakom prolasku kroz ciklus) a korak **iv** uslovni korak kojim je određeno trajanje ciklusa. I ovako opisanim postupkom telefoniranja nisu u potpunosti obuhvaćene sve situacije koje se mogu javiti pri tom procesu, ali zbog velikog broja tih situacija u ovom trenutku nećemo se baviti daljom razradom ovog algoritma.

Svaki algoritam mora da ima sledeća svojstva:

- definisanost

  svaki korak u algoritmu mora biti jednoznačno definisan;

- determinisanost

  vrednost koja se dobija posle izvršavanja svakog koraka jednoznačno je određena vrednostima iz prethodnog koraka;

- konačnost

  rad algoritma mora se završiti u konačnom broju koraka;

- rezultativnost

  algoritam mora dati rezultat za sve situacije za koje je kreiran;

- masovnost

  algoritam treba da obezbedi rešavanje cele grupe problema koji se razlikuju samo po ulaznim veličinama.

Algoritam je upotrebljiv ako se njegovom primenom dobija rezultat u konačnom „razumnom" vremenu. Algoritam koji bi birao potez igrača šaha tako što će ispitati sve moguće posledice poteza, zahtevao bi milijarde godina na najbržem računaru.

Postupci izrade algoritama nisu jednoznačni, inače, već bi postojali generatori algoritama. Samim tim, algoritmizacija problema zahteva i određenu dozu kreativnosti.

U praksi algoritmi se na računaru predstavljaju izvornim kodom (source) nekog višeg programskog jezika. Svi programski jezici podležu strogim pravilima pisanja instrukcija i definisanja podataka koji su za svaki programski jezik različiti.