PREDMET

CS111 OBJEKTO-ORIJENTISANO PROGRAMIRANJE

Predavanje broj 1

CS111-P01

UVOD U RAČUNARSKE SISTEME

Nedelja	Čas	Tematska jedinica	Predavanja Lekcija ili aktivnost	Rezultat – znanja ili veštine koje student treba da dobije
1	1	Uvod	O predmetu	Definisanje predmeta
	2	Istorija računarstva (SP1)	Uvod u računarske sisteme	Hardver i softver. Analogni i binarni signali. Mačinski jezik i jezici višeg nivoa. Prevođenje (kompilacija) i interpretacija programa. Istorijski pregled programskih jezika

Oktobar 2011.

SADRŽAJ

UVOD **3**
HARDVERSKE KOMPONENTE 3
MEMORIJA 4
ULAZNI I IZLAZNI UREĐAJI 4
SOFTVER 5
Tipovi programa *5*
Operativni sistemi *5*
Pokretanje programa *5*
ANALOGNI I BINARNI SIGNALI **6**
BAJT-OVI 8
PROGRAMI I MEMORIJA 9
MAŠINSKE INSTRUKCIJE 9
JEZICI VIŠEG NIVOA **10**
KOMPILACIJA (PREVOĐENJE) PROGRAMA 10
MOBILNOST PROGRAMA 11
INTERPRETERI 11
VIRTUELNA MAŠINA 12
ISTORIJSKI PREGLED RAZVOJA PROGRAMSKIH JEZIKA VIŠEG NIVOA 13
KONCEPTI I PARADIGME PROGRAMSKIH JEZIKA 17
SINTAKSA I SEMANTIKA PROGRAMSKIH JEZIKA 18

Predavanje br. 1

UVOD U RAČUNARSKE SISTEME

Uvod

Računar je sistem koji se sastoji iz više komponenti; čine ga hardverske i softverske komponente. Ako se prisećate nekog dobrog filma koji ste gledali, vi razmišljate o nečemu što je *konceptualno*. Film je nešto što se ne može opipati; ne možete ga uzeti i okačiti na zid ili prelomiti na pola; znači, on fizički ne postoji. Mogli biste da ga snimite na video traku – traka je fizička. Kada govorite o filmu, obično mislite na konceptualni film, a ne na traku na kojoj je on snimljen. Televizor koji koristite za gledanje filmova ima fizičku predstavu. Na njemu možete gledati tv program koji je opet konceptualan. Televizija znači kombinuje opipljive (televizijski aparat) i neopipljive predstave (tv program).

Računarski sistemi rade slično kao televizija. Oni kombinuju opipljive komponente koje nazivamo *hardverom* i konceptualne komponente, koje nazivamo *softverom*.

Hardverske komponente računarskog sistema su njegovi elektronski i mehanički delovi.

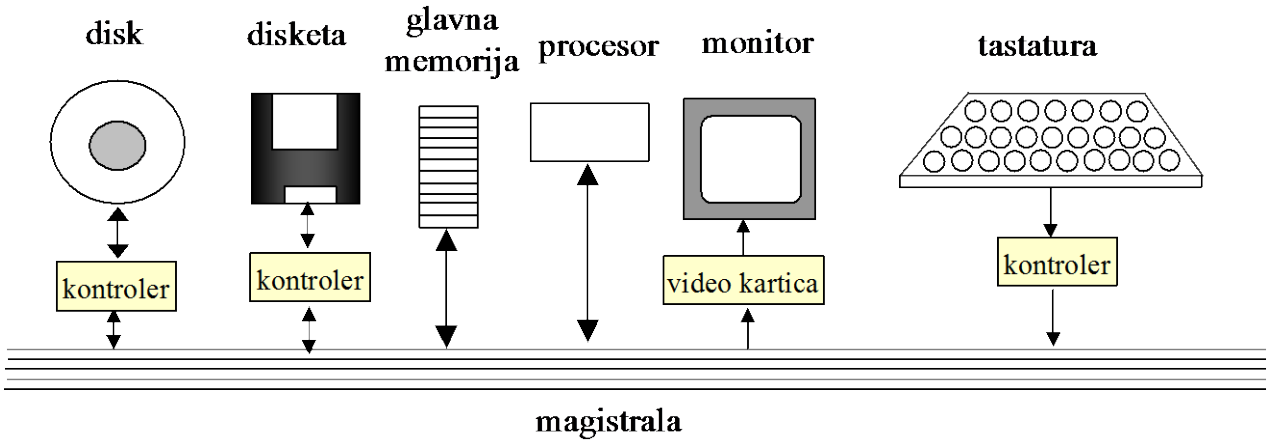
Softverske komponente sistema su podaci i programi koji postoje na tom računaru.

Osnovne hardverske komponente računarskog sistema su:

- procesor
- glavna memorija
- sekundarna memorija
- ulazni uređaji
- izlazni uređaji

Hardverske komponente

Termini *ulaz* i *izlaz* ukazuju na tok podataka *u* i *iz* sistema. Na sledećoj slici su prikazane osnovne hardverske komponente računarskog sistema. Strelice ukazuju na tok podataka.



Slika 1: Osnovne komponente računara

Magistrala predstavlja grupu žica na matičnoj ploči računara. Preko nje teku podaci između komponenti. Većina uređaja je sa magistralom povezana preko kontrolora koji koordinišu aktivnosti uređaja i magistrale.

Procesor je elektronski uređaj koji u sebi sadrži milione drugih elektronskih uređaja – tranzistora. Procesor je mozak računara. On obavlja osnovne računske operacije u sistemu direktno, ili indirektno kontroliše sve ostale komponente. Procesor se često naziva **centralnom procesorskom jedinicom** (CPU - Central Processor Unit). Konkretan računar ima konkretan tip procesora. Personalni računari na primer, najčešće koriste procesore tipa ''Pentium''.

Memorija

Procesor je mozak računara. Sve računske operacije se obavljaju u njemu. Druge komponente učestvuju u radu (na primer, prebacuju podatke *u* i *iz* procesora), ali je procesor mesto gde se obavljaju osnovne akcije.

Za razliku od ljudskog mozga koji kombinuje memoriju sa mogućnošću obrade, procesor računara ima veoma malo memorije. On se zbog toga mora osloniti na druge komponente koje će skladištiti podatke i programe, kao i rezultate njegovih izračunavanja. Kod računarskih sistema postoje dva osnovna tipa memorije:

Glavna memorija

- tesno povezana sa procesorom;
- sadržaj se brzo i lako menja;
- sadrži programe i podatke sa kojima procesor trenutno radi;
- interakcija sa procesorom se obavlja u milionitim delovima sekunde.

Sekundarna memorija

- sa glavnom memorijom povezana preko magistrale i kontrolora;
- sadržaj se lako menja, ali je to u poređenju sa glavnom memorijom, vrlo sporo;
- koristi se za dugotrajno skladištenje programa i podataka;
- sa ovom memorijom procesor komunicira samo povremeno.

Glavna memorija je mesto gde se podaci i programi nalaze kada ih procesor aktivno koristi. Kada programi i podaci postaju aktivni, oni se kopiraju iz sekundarne memorije u glavnu. Ta kopija ostaje u glavnoj memoriji. Ova memorija je blisko povezana sa procesorom, tako da je prebacivanje instrukcija iz programa i podataka do procesora vrlo brzo. Glavna memorija se često naziva *RAM* memorijom. To je skraćenica za Random Access Memory. "Random" (slučajno) znači da se memorijskim ćelijama može pristupiti bilo kojim redosledom.

Kada ljudi kažu da računar ima "256 megabajta RAM-a", oni govore o njegovoj glavnoj memoriji. U glavnoj memoriji se ništa ne čuva za stalno. Ponekad se podaci tu nalaze samo nekoliko sekundi, odnosno samo onoliko vremena koliko su bili potrebni.

Sekundarna memorija je mesto gde se podaci i programi trajno skladište. Najčešće se kao uređaji za skladištenje koriste diskovi i diskete. Razlog postojanja dve vrste memorije se može ustanoviti iz sledeće tabele koja daje uporedan pregled karakteristika glavne i sekundarne memorije.

Glavna memorija	**Sekundarna memorija**
Brza	Spora
Skupa	Jeftina
Mali kapacitet	Veliki kapacitet
Povezana direktno sa procesorom	Nije povezana direktno sa procesorom

Ulazni i izlazni uređaji

Ulazni i izlazni uređaji omogućavaju računaru da komunicira sa spoljašnjim svetom. Tom prilikom se podaci prenose *u* i *iz* sistema. Ulazni uređaj se koristi za unos podataka u sistem. Neki ulazni uređaji su na primer, tastatura, miš, mikrofon, čitač bar koda.

Izlazni uređaji se koriste za slanje podataka iz sistema. Neki izlazni uređaji su, na primer, monitor, štampač i sl.

Softver

Softver obuhvata programe i podatke koje računar koristi; nalazi se na nekom uređaju, na primer, na disku, ali je sam po sebi neopipljiv. Softver se sastoji od programa i podataka. Programi su lista instrukcija koje procesor treba da izvrši. Podaci mogu biti bilo koje informacije koje program koristi. To mogu biti brojčani podaci, tekst, zvučni ili neki drugi podaci. Razlika između programa i podataka nije baš tako jasna kako bismo možda zaključili.

I programi, i podaci se u memoriji računara predstavljaju na isti način. Elektronika memorije računara ne pravi razliku između programa i podataka. Upravo ta činjenica da se i programi, i podaci skladište na isti način, jedna je od najvažnijih ideja u računarskim naukama. Računarski sistemi svoju memoriju koriste za ono što potrebe nalažu.

Tipovi programa

Postoje dve kategorije programa: *Aplikacioni programi* (aplikacije) su programi koje ljudi koriste za obavljanje nekog konkretnog posla. Računari i postoje upravo zato što ljudi imaju potrebu da koriste takve programe. *Sistemski programi* omogućavaju da hardver i softver jednog računarskog sistema rade kako treba. U sledećoj tabeli su dati primeri jedne i druge vrste programa.

Aplikacioni programi	**Sistemski programi**
Programi za obradu teksta	Operativni sistem
Igre	Sistem za kontrolu mreže
Sistemi za upravljanje bazama podataka	Program za izvršavanje nekog programskog jezika
Pretraživači web-a	

Operativni sistemi

Najvažniji sistemski program je operativni sistem. Ovaj program je uvek prisutan kada računar radi. On koordinira rad svih hardverskih i softverskih komponenti sistema. Operativni sistem je odgovoran za pokretanje aplikacionih programa; za pronalaženje resursa koji su potrebni za rad tih programa. Tokom rada nekog aplikacionog programa operativni sistem rukuje detaljima vezanim za hardver koji su, tom prilikom - potrebni. Na primer, kada na tastaturi unesete neki karakter, operativni sistem određuje kojem programu je to namenjeno.

Savremeni operativni sistemi obično imaju grafički korisnički interfejs koji korisnicima omogućava da lako komuniciraju sa aplikacionim programima. Komunikacija se obično obavlja preko prozora, dugmadi, menija i tastature. Primeri operativnih sistema su "Unix", "Windows", "Linux", "Solaris" itd.

Operativni sistem je program. Isti hardver se može koristiti sa više operativnih sistema. Ponekad se operativni sistem može oštetiti; u tom slučaju ga morate ponovo instalirati. Sve dok operativni sistem ponovo ne bude u funkciji, ne možete koristiti druge programe na računaru.

Pokretanje programa

Šta se dešava kada korisnik pokrene neku aplikaciju? Pod pretpostavkom da je operativni sistem već pokrenut, dešava se sledeće:

1. Korisnik pokreće aplikaciju (klikom na ikonu, izborom iz menija ili na neki drugi način);
2. OS oodređuje ime aplikacije;
3. OS pronalazi deo na disku gde se nalazi taj aplikacioni program i negovi podaci;
4. OS pronalazi deo glavne memorije koji se ne koristi i koji je dovoljno velik za tu aplikaciju;
5. OS pravi kopiju aplikacije i njenih podataka i smešta to u glavnu memoriju;
6. Operativni sistem definiše resurse potrebne za rad te aplikacije;

7. OS pokreće aplikaciju.

Tokom rada aplikacije operativni sistem u pozadini rukuje resursima, obavlja ulazne i izlazne operacije i omogućava rad cele aplikacije.

Analogni i binarni signali

Reč "binarno" znači "dva stanja". Ova dva stanja se nekad označavaju sa "0 i 1"; ponekad sa "tačno i netačno" ili "uključeno i isključeno". Najvažnija karakteristika svakog binarnog uređaja je da on može imati dva stanja.

Bit je signal, odnosno najmanji element programa koji može imati samo dve vrednosti: 'uključeno/isključeno" ili 1 ili 0.

Zamislite prekidač za svetlo. Taj prekidač možete samo uključiti ili isključiti. Prekidač znači nosi jedan bit informacija.

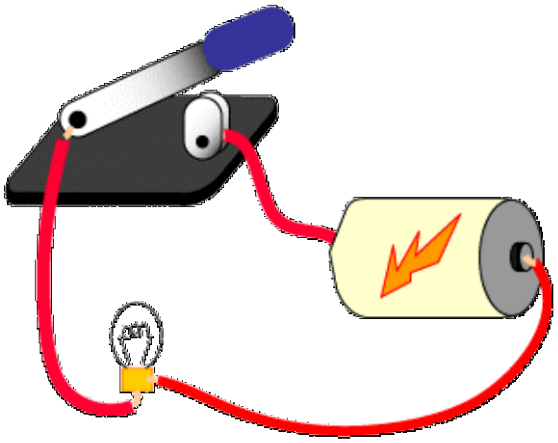
Prekidač sa postepenim podešavanjima nije binarni uređaj. On ima više položaja koji se nalaze između *uključenog* i *isključenog*.

Prekidač za paljenje nekog automobila je diskretan. On ima konačan broj stanja, ali je to obično više od dva stanja. Dugme na kalkulatoru je binarni uređaj. Ono može biti u stanju *uključeno* ili *isključeno*. Kada pritisnete dugme, ono je u stanju *uključeno*. Kada otpustite dugme, ono se vraća u stanje *isključeno*.

Zašto se koriste binarni signali? Neki od razloga su:

1. **Jednostavni, laki za izradu**

Prekidač za uključivanje i isključivanje se lako pravi. Takav prekidač spaja ili odvaja dva metalna dela.



Slika 2 Vrednosti bita: »uključeno« ili »isključeno«

Ako biste želeli da napravite prekidač koji bi postepeno menjao jačinu svetla, to bi bilo mnogo teže za rad, utoliko što bi takav prekidač morao da ima i više delova.

Isto važi i za silikonski čip. Prekidači sa samo dva stanja se relativno lako prave. To znači da su uređaji mali, jeftini i pouzdani, tako da se milioni njih mogu staviti na mali prostor.

2. **Nedvosmisleni signali**

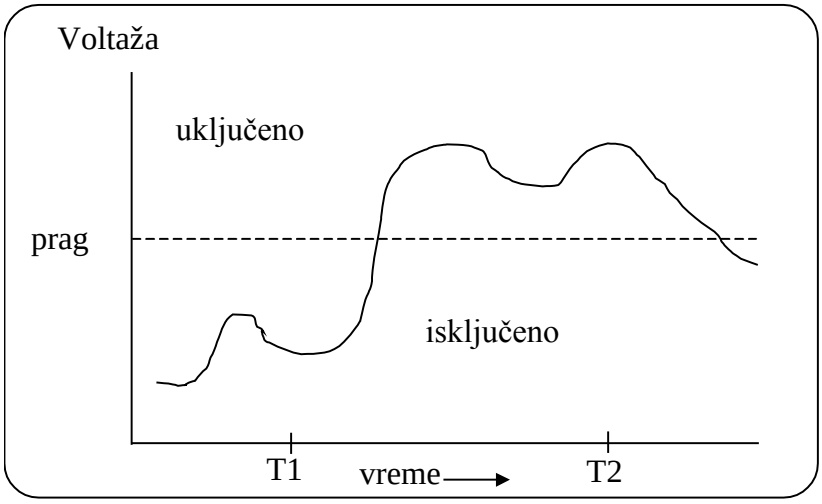
Godine 1775., kada su se Amerikanci borili za nezavisnost, Pol Revere je čekao na vesti o tome da li Englezi dolaze kopnom ili morem. O tome je trebalo da izvesti Vrhovnu komandu. Signal je trebalo da dođe preko svetionika. Zamislite da je dogovoreno da signal bude sledeći:

1.32456, ako dolaze kopnom i 1.71922, ako dolaze morem.

Signal je došao, ali glasnik čeka nekoliko sati dok se precizno ne otkrije šta je javljeno, odnosno koliko je jak bio svetlosni signal. Dvosmislenost signala može biti veliki problem. Dogovoreni signal koji je čekao Pol Revere je bio: jedan signal za dolazak kopnom, a dva – za dolazak morem. Takav signal se

lako interpretirao. On je trebalo samo da izbroji. Ovakvi signali su diskretni, pošto imaju fiksni broj konačnih stanja.

Analogni signal kontinuirano menja svoju vrednost. Ovakav signal može imati bilo koju vrednost u nekom opsegu. Ovo je ilustrovano na sledećoj slici:

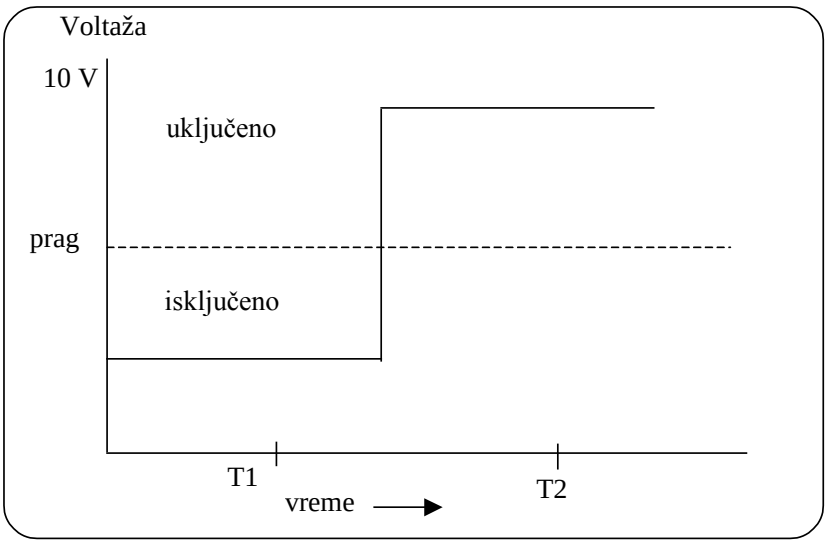


Slika 3 Promena vrednosti kontinualnog signala – analogni dijagram

Ovo je zvučni signal. Vrednost u nekom trenutku morate čitati sa dijagrama.

Sa druge strane, ako posmatramo samo voltažu, možemo ovo iskazati i na sledeći način: Vrednosti ispod nekog praga se smatraju signalom 0, a vrednosti iznad – smatraju se signalom 1.

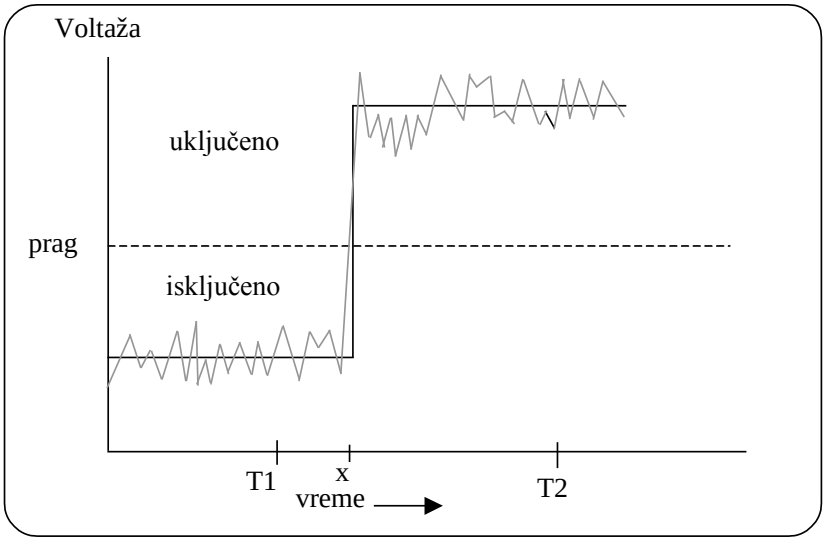
Analogni signali kontinuirano menjaju vrednost. Ako se uvede prag, analogni signali se mogu predstaviti kao binarni podaci. Ovo je lako i brzo, tako da elektronika, (a i ljudi) mogu lako da odrede da li je voltaža iznad ili ispod nekog praga. Na sledećoj slici je prikazan signal ovog tipa.



Slika 4 Digitaliziran dijagram iz analognog dijagrama na slici 3

3. **Mogu se napraviti besprekorne kopije**

Na kraju na kojem se prima signal - zainteresovani smo samo za binarne vrednosti. Sve što treba uraditi je da se proveri da li je signal iznad ili ispod praga. Ovo se može uraditi savršeno precizno.



Slika 5 Rekonstrukcija idealnog digitaliziranog dijagrama

Originalni signal je besprekorno rekonstruisan. Taj proces se može ponoviti onoliko puta koliko je potrebno, pri čemu se svaki put dobija savršena kopija. Za računarske sisteme ovo je vrlo bitno. Razlog je što se kod računara šabloni sa bit-ovima stalno prebacuju od procesora do memorije i obrnuto. Ovo se dešava i po milion puta u sekundi. Te kopije moraju biti savršene.

4. **Binarnim signalom se može predstaviti bilo šta**.

Sve što se može predstaviti nekim obrascem, može se predstaviti obrascem bit-ova.

Ideja je da se bilo koji sistem simbola može prevesti u obrazac sa bit-ovima. Pogledajmo na primer, kako se karakteri predstavljaju preko obrazaca od osam bit-ova. Sporazum o tome šta obrazac predstavlja se naziva ASCII. Hardver i softver računara prate ovaj sporazum kada su u pitanju tekstualni podaci. Drugi tipovi podataka se predstavljaju na drugi način.

Japanski i kineski karakteri se takođe prebacuju u obrasce bit-ova, a računari mogu da njima manipulišu podjednako lako kao da je reč o engleskim slovima. Tada se koristi sporazum nazvan *UNICODE,* koji karaktere predstavlja preko 16 bit-ova (ASCII to radi preko 8 bit-ova). Na primer, karakter 茶 je preko bit-ova predstavljen kao 111110011111110.

Ove karakteristike binarnih signala je prvi predstavio Klod Šanon. Njegov rad iz 1948. godine, pod naslovom ''Matematička teorija komunikacija'', predstavlja osnovu informatičkih teorija i računarskih nauka.

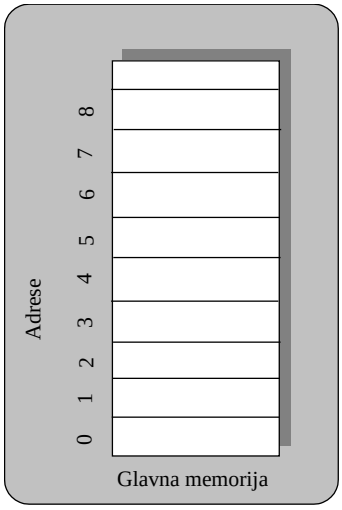
Bajt-ovi

Jedan bit informacija je toliko mali da se memorija računara obično organizuje u grupe od po osam bit-ova. Osam bit-ova čine jedan bajt. Kada je za neke podatke potrebno više od osam bitova koristi se ceo broj bajt-ova. Jedan bajt je otprilike dovoljno memorije za skladištenje jednog karaktera.

Za podatke je često potrebno imati na raspolaganju hiljade, milione ili milijarde bajtova. Veće jedinice od jednog bajta se izražavaju kao umnošci stepena broja 2.

Ime	Broj bajtova	Stepen broja 2
bajt	1	2^0
kilobajt	1024	2^{10}
megabajt	1,048,576	2^{20}
gigabajt	1,073,741,824	2^{30}
terabajt	1,099,511,627,776	2^{40}

Glavna memorija se sastoji od dugačke liste bajtova. Kod većine modernih računara, svaki bajt ima adresu koja se koristi da bi se on pronašao. Na sledećoj slici je prikazan deo glavne memorije.



Slika 6 Glavna memorija sa adresama memorijskih lokacija

Svaki pravougaonik na ovoj slici predstavlja jedan bajt. Svaki bajt ima adresu. Adrese su, na ovoj slici, celi brojevi sa leve strane. Adrese na većini računara počinju od 0 i rastu naviše sve dok svaki bajt ne dobije adresu.

Adresa nekog bajta nije deo njegovog sadržaja. Kada je procesoru potrebno da pristupi određenom bajtu, računar zna kako da u memoriji pronađe taj bajt.

Procesor može da čita ili piše nove bajtove na određenoj memorijskoj lokaciji. U istom trenutku može da čita ili piše veći broj bajtova. Ako se kaže da je neki procesor 64-bitni, to znači da on u istom trenutku može da piše ili čita 64/8=8 bajtova.

Programi i memorija

Tokom rada procesor stalno upisuje nove podatke i čita postojeće. Program tokom rada ima deo memorije koji koristi. Ako program, na primer, sabira brojeve, onda će se i suma naći u glavnoj memoriji (verovatno će se upotrebiti nekoliko bajtova). Kako se sumi budu dodavali novi brojevi, suma će se menjati, a samim tim se menja i glavna memorija.

Drugi delovi glavne memorije se možda neće menjati; na primer, instrukcije koje čine program se obično ne menjaju tokom rada programa. Instrukcije se takođe nalaze u glavnoj memoriji, ali se ove lokacije ne menjaju.

Kada u nekom programskom jeziku pišete program, niste Vi taj koji brine o memorijskim lokacijama i njihovom sadržaju. O tome automatski brine programski jezik (Java, C) u kojem je program pisan. Cilj programskog jezika višeg nivoa, kao što je *Java,* upravo je da male elektronske operacije organizuje u veće jedinice predstavljene iskazima programskog jezika.

Mašinske instrukcije

Korisnici i programeri obično ne razmišljaju o milionima elektronskih operacija koje se dešavaju svake sekunde. Situacija je slična sa vožnjom automobila. Tokom vožnje razmišljate o velikim operacijama, kao što su - ubrzati, skrenuti levo i sl. Ne razmišljate o delovima automobila i o tome kako oni funkcionišu.

Jedna elektronska operacija koju procesor izvodi se naziva *mašinskom operacijom*. Procesor u jednom trenutku obavlja jednu operaciju, ali u jednoj sekundi izvodi milione takvih operacija.

Mašinska instrukcija se sastoji od nekoliko bajtova u memoriji koji procesoru govore da obavi jednu mašinsku operaciju. Procesor pronalazi mašinske instrukcije u glavnoj memoriji i izvršava ih sekvencijalno. Skup mašinskih instrukcija u glavnoj memoriji računara naziva se *izvršnim programom*.

Program pisan u mašinskom jeziku predstavlja niz mašinskih instrukcija koje se nalaze u glavnoj memoriji računara. Jedan program sadrži milione mašinskih instrukcija.

Jezici višeg nivoa

Vrlo retko (skoro nikad) će programeri program pisati u mašinskom jeziku. Izvršne datoteke za većinu aplikacija sadrže milione instrukcija u mašinskom jeziku. Čoveku bi bilo vrlo teško da prati i kreira takve instrukcije.

Većina programa se pravi u programskim jezicima višeg nivoa, kao što su Java, C, C++. Kod ovakvih jezika programer pravi program pomoću velikih operacija, koje se kasnije mogu pretvoriti u male mašinske operacije.

Pogledajte, na primer, sledeću liniju koda, napisanu u programskom jeziku Java ili C:

```
int sum = 0;
```

Ovoj jednoj liniji koda odgovaraju mašinske operacije zauzimanja dela glavne memorije za skladištenje broja, za smeštanje broja na to mesto, kao i za obaveštavanje drugih delova programa gde se taj broj nalazi. Očigledno je da je za čoveka mnogo lakše da napiše ovakav program, nego da piše mnogobrojne detaljne instrukcije.

Programeri programe pišu preko komandi u programskom jeziku višeg nivoa. Program pisan u takvom jeziku se sastoji od redova teksta koji je napravljen u nekom editoru teksta. Taj program je posle pisanja upamćen u nekoj datoteci koja se čuva na disku. Sledeći primer je kompletan program pisan u programskom jeziku C.

```
#include <stdio.h>
main()
{
  int sum = 0;
  sum = 2 + 2;
  printf( "%d\n", sum );
}
```

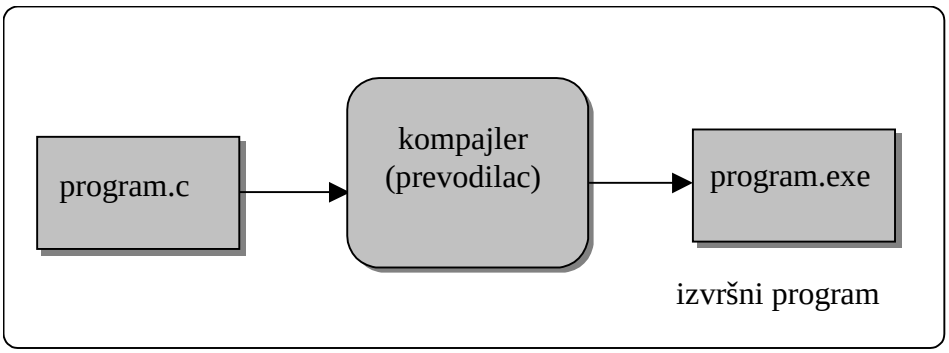
Ovaj program bi mogao da se na disk smesti u datoteci *sabiranje.c*. Kao i sve datoteke, i ovaj program sadrži niz bajtova. Pošto je u pitanju tekstualna datoteka, ti bajtovi sadrže podatke tipa *karakter*. Tu datoteku možete menjati preko editora teksta i štampati na štampaču. U ovoj datoteci se ne nalaze mašinske instrukcije. Ako bi se ovi bajtovi iskopirali u glavnu memoriju, oni ne bi mogli da se izvrše kao program, naravno, (ako se pre toga ne uradi još nešto).

Izvorni program (*izvorna datoteka*) je tekstualna datoteka koja sadrži instrukcije pisane u programskom jeziku višeg nivoa. Procesor ovu datoteku ne može da izvrši bez nekih dodatnih koraka.

Izvorni program se obično prevodi na mašinski jezik. U tu svrhu postoji poseban program, kompajler, koji uzima datoteku sa izvornim kodom i od nje pravi izvršni program (program u mašinskom jeziku). Na primer, program, *sabiranje.c* bi mogao da se prevede u izvršni program. Taj izvršni program bi se mogao naći u datoteci *sabiranje.exe*. Sada se ta izvršna verzija može da kopira u glavnu memoriju i da se tamo izvršava.

Kompilacija (prevođenje) programa

Na sledećoj slici je prikazano šta se dešava sa programima koji su pisani u jeziku C.



Slika 7: Prevođenje izvornog koda u izvršni program

Šta se dešava:

1. Uz pomoć editora teksta se pravi datoteka sa izvornim kodom.
    - Ova datoteka sadrži instrukcije u programskom jeziku višeg nivoa.
    - U njoj su bajtovi koji predstavljaju karaktere.
2. Datoteka sa izvornim kodom se smešta na disk.
3. Procesor ne može da izvrši datoteku sa izvornim kodom.
4. Kompajler prevodi datoteku sa izvornim kodom na mašinski jezik.
    - Datoteka sa izvornim kodom se ne menja. Pravi se nova datoteka sa izvršnim kodom.
    - Kompajleri se koriste za programe pisane u programskim jezicima višeg nivoa i rade sa određenim procesorima i operativnim sistemima ("Windows", "Linux").
5. Datoteka sa izvršnim kodom se takođe smešta na disk.
6. Program se izvršava tako što operativni sistem pravi kopiju datoteke sa izvršnim kodom i prebacuje je u glavnu memoriju.

Ovo što je izloženo važi za većinu programskih jezika, kao što su ''Ada'', ''Pascal'', ''C'', ''C++'', ''FORTRAN'' i sl. Kod programskog jezika ''Java'' postoje i neki dodatni koraci, o kojima će kasnije biti više reči.

Mobilnost programa

U idealnim situacijama, u programskom jeziku višeg nivoa potrebno je napisati samo jedan program. Ta datoteka sa izvornim kodom se dalje može prevesti u nekoliko izvršnih datoteka, od kojih svaka sadrži prave mašinske instrukcije za određeni procesor.

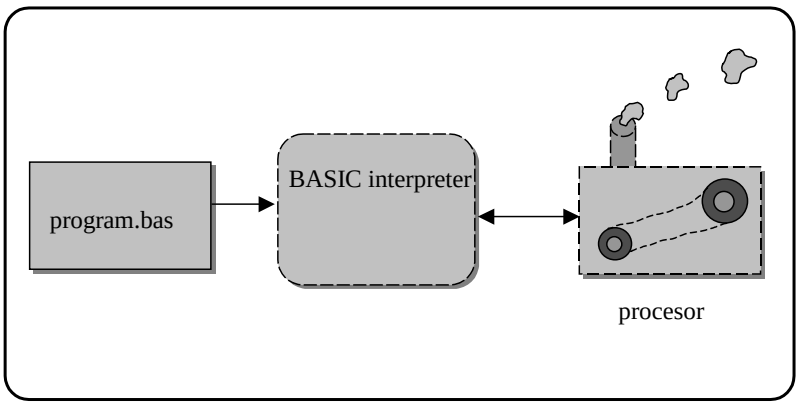
Ideja o upotrebi jedne datoteke sa izvornim kodom za kreiranje izvršnih datoteka koje rade na različitim procesorima, ima za cilj ostvarenje mobilnosti softvera. Najbolje bi bilo kada biste program mogli da napišete jednom (u jeziku višeg nivoa), a da ga onda izvršavate na različitim računarskim sistemima (prevođenjem na konkretan mašinski jezik).

Nažalost, nemamo idealnu situaciju. Ako želite da vam program radi na drugoj mašini, javiće se mnoštvo malih problema koji će Vas u tome sprečiti. To je naravno moguće učiniti, ali je potrebno uložiti dodatni trud. Jedna od prednosti programskog jezika *Java* je upravo ta automatska prenosivost sa jednog na drugi računarski sistem. Više o tome kasnije.

Interpreteri

Procesor nikad direktno ne izvršava programe pisane u programskim jezicima višeg nivoa. Jedan od načina za realizaciju programa ste već videli: to je prevođenje programa pomoću kompajlera.

Drugi način je da se koristi tzv. interpreter (engl. interpreter), tj. specifični prevodilac za određeni jezik. Interpreter je program koji se ponaša kao procesor, odnosno koji može da direktno izvršava programski jezik višeg nivoa.



Slika 8: Interpretiranje izvornog programa

Izvorni program na ovoj slici je *program.bas*; pisan je u programskom jeziku BASIC. Za njegovo pisanje je korišćen običan editor teksta. Ovaj program se interpretira preko interpretera za BASIC koji radi na procesoru. Interpreter čita komande iz izvršne datoteke i radi ono što se tamo nalaže.

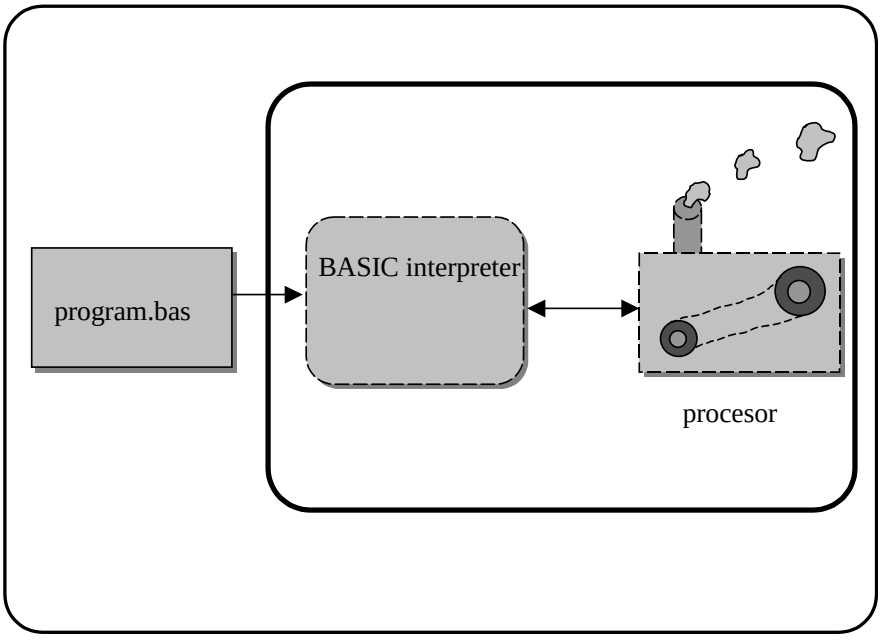
Verovatno ste ovo do sada mnogo puta koristili, a da toga niste ni bili svesni. Može se reći da kompjuterske igrice koje koristite (na primer DOOM) predstavljaju interpreter za komande koje unosite kao korisnik. Ove komande nisu iste kao komande u pravom programskom jeziku, ali im je namena ista.

Interpreter je računarski program koji čita program napisan u izvornom programskom jeziku višeg nivoa (ili kodu) i prevodi ga u instrukcije u tzv, mašinskom jeziku (kodu), tj. u instrukcije koje se direktno izvršavaju u određenom računaru. Upotrebom interpretera, jedan program pisan u izvornom programskom jeziku može da proizvede isti rezultat na različitim računarima. Kada, sa druge strane, koristimo kompajler, program pisan u izvornom programskom jeziku se prevodi u izvršni program specifičnog računara, tj. procesora i operativnog sistema.

Virtuelna mašina

Da bi se povećala efikasnost, pojedini prevodioci, tj. interpreteri prevode originalni program pisan u programskom jeziku višeg nivoa u jedan prelazni oblik, pre prevođenja u mašinski jezik. Najčešće se taj oblik programa može da sačuva u memoriji i kasnije koristi, kada se želi da prevede u određeni mašinski jezik Na primer, to je slučaj kod korišćenja jezika Java, Python, UCSD Pascal. Ovakav oblik programa u kome se program privremeno nalazi se često naziva i bajkodom (engl. bycode). Poseban bajkod interpreter (to je program pisan npr. u C jeziku) prevodi bajkod program u mašinski jezik na konkretnom hardveru, tj. na određenom računaru.

Programi pisani u jezicima višeg nivoa, a prevedeni u bajkod, očigledno su u nekom obliku koji je nezavisan od bilo kog hardvera, te se su oni u obliku koji se može primeniti u tzv. *virtualnoj mašini* koja je nezavisna od bilo kog hardvera. Program u virtualnoj mašini se prevodi u mašinski jezik konkretnog hardvera tek kada se upotrebi bajkod interpreter za određeni hardver. Na slici 9 je prikazan slučaj primene virtualne mašine. Ova slika je ista kao i prethodna, osim što je sada dodat pravougaonik koji uokviruje procesor i interpreter. Ova kombinacija se ponaša kao mašina koja može direktno da izvršava programe pisane u BASIC-u. Kada interpreter izvršava izvorni program - i interpreter, i izvorni program se nalaze u glavnoj memoriji. Interpreter sadrži mašinske instrukcije koje hardver direktno izvršava. Izvorni program sadrži komande u određenom jeziku (ovde BASIC) koje interpreter poznaje. Iz perspektive programa pisanog u BASIC-u izgleda kao da se komande, pisane u BASIC-u – direktno izvršavaju, odnosno kao da ih izvršava neka vrsta mašine.



Slika 9: Interpretacija izvornog programa na virtuelnom procesoru

Ako se posmatra brzina realizacije programa koji se kompajliraju i onih – koji se interpretiraju, situacija je slična kao sa jezicima koje koriste ljudi.

Prevodilac uzima ceo dokument u jednom jeziku i pravi kompletan dokument u drugom jeziku, koji se kasnije može da koristi u bilo kom trenutku.

Interpeter se ponaša kao spona između govornika koji koristi jedan jezik i slušalaca koji koriste drugi jezik. To izgleda kao da oni direktno govore jedan drugom.

Interpreter kod ljudi radi sporije, nego da se direktno govori u određenom jeziku. Isto važi i za interpretere kod računarskih jezika. Interpreter mora da, tokom rada sa jezikom koji interpretira - uradi dodatni posao. Taj dodatni posao dovodi do toga da je virtuelni procesor sporiji nego pravi.

Brzina izvršenja izvornog programa koji koristi interpreter je sporija od izvršnog programa koji je dobijen primenom kompajlera, jer interpreter mora da stalno analizira svaku programsku instrukciju. Ako se jedna isntrukcija više puta ponavlja, interpreter će je više puta, iz početka analizirati, jer radi po principu »instrukcija po instrukcija«. U slučaju primene kompajlera, vršie se analiza celog programa odjednom i njegovo prevođenje u izvršni program u mašinskom jeziku određenog hardvera. Pri prevođenju izvornog u mašinski jezik, kompajler je u situaciji da traći i nađe najbolje mogužu kombinaciju izvršnih instrukcija koja se najbrše izvršava, jer ima uvid u celinu programa. Zato se obično kaže da kompajleri daju optimalan izvršni program. S druge strane, izvorni program koji se pretvara u izvršni program uz pomoć interpretera, kako s eprevođenje vrši po primcipu »instrukcija po instrukcija«, ne može da bude optimalan, sa stanovišta efikasnosti, a javlja se i gubitak vremena na prevođenje isntrukcija, jer se iste instrukcije ponovo prevode i time gubi vreme u prevođenju.

Pristup promenljivim veličinama koje program koristi je takođe sporiji kod programa dobijenih uz pomoć interpretera, jer se preslikavanje memorijskih adresa u stvarne fizike adrese vrši pri svakom izvršenju programa, u tzv. „realnom“ vremenu (engl. run-time), a ne u vreme prevođenja od strane kompajlera, tj. u vreme izrade izvršnog programa od strane kompajlera.

Iz iznetih razloga, postoje različiti odnosi vremena razvoja programa i vremena izvršenja programa. Po pravilu, razvoj programa koji koriste interpreter je brži, jer programer odmah može da utvrdi rezultat instrukcije koju unosi u program. U slučaju korišćenja programa koji koristi kompajler, on mora prvo da napiše ceo program (ili jedan njegov deo) u jeziku višeg nivoa, pa da ga uz pomoć kompajlera prevede u izvršni oblik, i da tako dobijen izvršni program pusti u rad da bi video da li dobija rezultate koje očekuje. Taj proces „piši program-prevodi kompajlerom-aktiviraj izvršni program“ traje mnogo duže nego u slučaju razvoja programa koji se interpretiraju po po principu „instrukcija-po-instrukcija“. Međutim, iz ranije navedenih razloga, konačno izvršenje programa dobijenih interpreterima je najčešće sporije, jer se ne koristi izvršni program koji je optimiziran (kao u slučaju primene kompajlera), već izvršni program koji je automatski generisan prevođenjem programskih instrukcija u mašinske instrukcije.

Neki programski jezici (npr. LISP) dozvoljavaju da se programi dobijeni interpreterima (tzv. interpreter programi) i izvršni programi dobijeni primenom kompajlera (tzv. kompilirani programi) međusobno pozivaju, tj. da komuniciraju i da dela promenljive. Ovo omogućava da se jednom razvijen i testiran program upotrebom interpretera, može prevesti u izvršni program primenom kompajlera da bi dobila veća brzina izvršenja programa, jer, kao što je gore navedeno, izvršni programi dobijeni primenom kompajlera su uvek brži od programa dobijeni primenom interpretera. Pojedini interpreteri koriste različite tehnike kojima obezbeđuju da veću brzinu izvršavanja programa koje prevode, ali je generalno pravilo da su izvršni programi dobijeni primenom kompajlera brži od programa koje daju interpreteri.

Istorijski pregled razvoja programskih jezika višeg nivoa

Wegner je dao jednu klasifikaciju programskih jezika višeg nivoa, koju ćemo i mi ovde koristiti. Kao kriterijum su korišćene karakteristike jezika koje su se po prvi put pojavile.

Tako je prva generacija programskih jezika nastala u periodu od 1954. do 1958. godine. To su bili jezici:

- FORTRAN I podržavao matematičke izraze;
- ALGOL 58 podržavao matematičke izraze;
- Flowmatic podržavao matematičke izraze:
- IPL 5 podržavao matematičke izraze.

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

- FORTRAN II podržavao potprograme, odvojenu kompilaciju;
- ALGOL 60 blokovska struktura, tipovi podataka;
- COBOL opis podataka, rukovanje datotekama;
- Lisp skupljanje otpada, pokazivači, obrada listi.

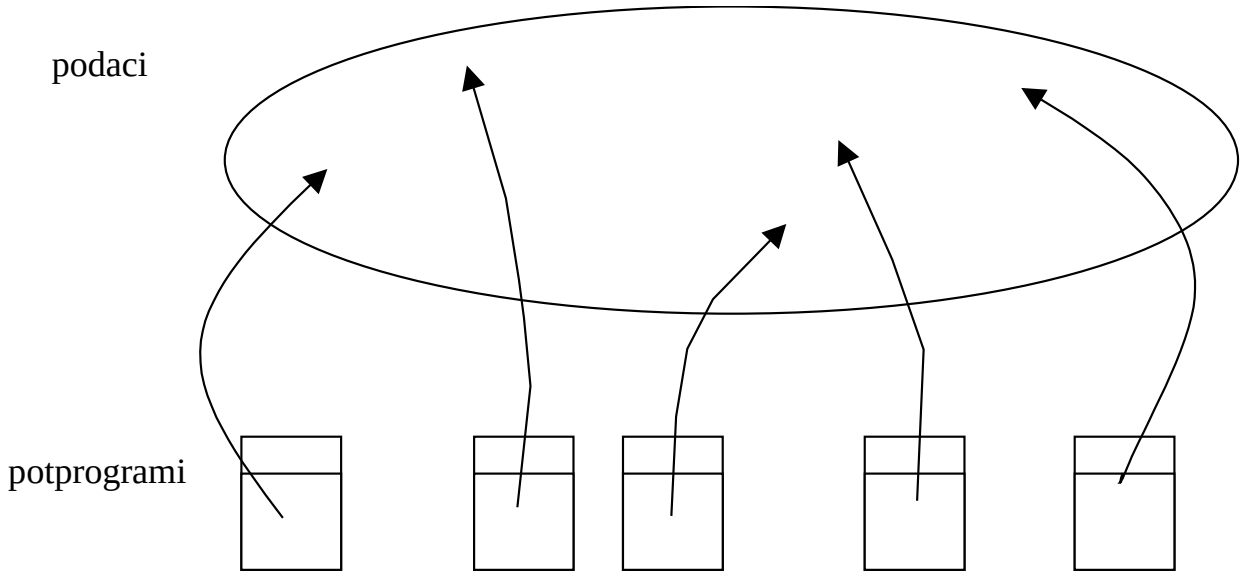
Jezici treće generacije su se koristili tokom perioda od 1962. do 1970. godine. To su bili:

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

Period od 1970. do 1980. beleži zastoj u razvoju programskih jezika. Nastalo je mnogo novih jezika, ali je malo njih zaista zaživelo. Tako su tada nastali, ali i opstali jezici kao što je "Smalltalk" (naslednik jezika" Simula"), "Ada" (naslednik jezika "ALGOL 68" i "Pascal") ili "C++" (izveden iz jezika "C" i "Simula"). Primeri u ovom predmetu biće uglavnom biti rađeni u "Javi". To je programski jezik nastao 90-ih godina prošlog veka, koji korene vuče iz jezika "C++".

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

Topologija jezika prve i rane druge generacije Na sledećoj slici je prikazana topološka struktura jezika prve generacije i rane druge generacije. Pod topologijom se podrazumevaju osnovni gradivni elementi jezika, kao i način njihovog povezivanja.



Slika 10: Topologija programskih jezika prve i rane druge generacije

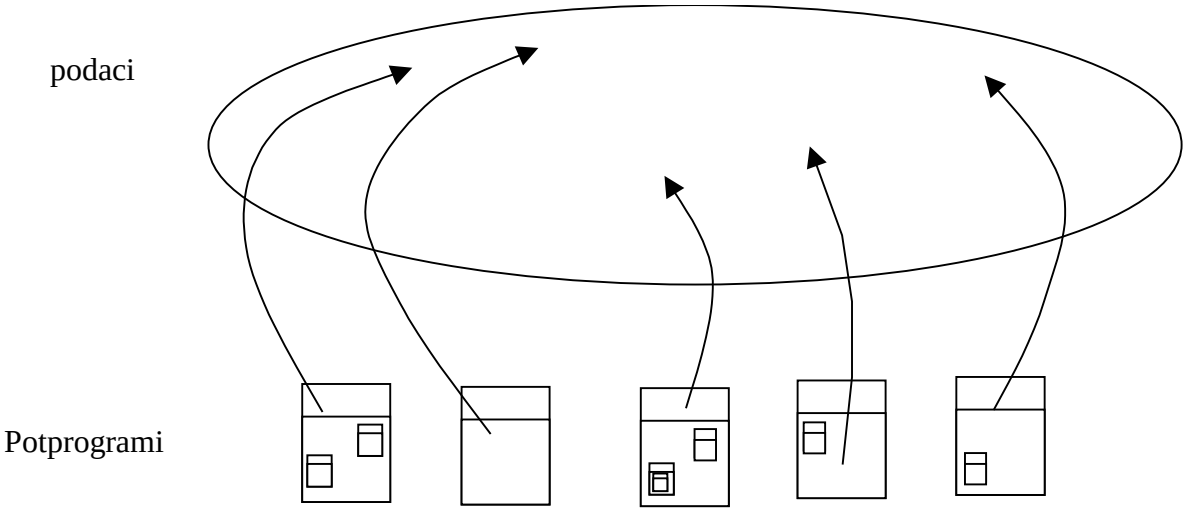
Sa slike se vidi da su kod jezika kao što su FORTRAN ili COBOL, osnovni gradivni elementi programa - potprogrami. Aplikacije pisane u ovim jezicima imaju relativno ravnu fizičku strukturu koju čine samo globalni podaci i potprogrami. Strelice na slici ukazuju na međusobnu zavisnost potprograma i podataka. Programer bi, tokom projektovanja aplikacije, mogao da odvoji različite vrste podataka, ali sami jezici nemaju ugrađenu podršku za to. Greška u jednom delu programa može imati katastrofalne posledice na ostale delove sistema, pošto globalnim strukturama podataka mogu da pristupaju svi potprogrami. Održavanje je poseban problem. Čak i posle vrlo kratkog perioda održavanja dolazi se do mnoštva potprograma koji su međusobno isprepletani, što umanjuje pouzdanost sistema.

Topologija jezika kasne druge i rane treće generacije Sredinom 60-ih godina prošlog veka programi su najzad identifikovani kao posrednik između problema i računara. Prve softverske apstrakcije su direktno proizašle iz takvog pragmatičnog pogleda na softver. Potprogrami su postojali i

ranije, ali nisu shvaćeni kao apstrakcije. Oni su prvobitno korišćeni kao način da se uštedi trud. Sada su potprogrami prihvaćeni i kao način da se izvuku i apstrahuju funkcije programa.

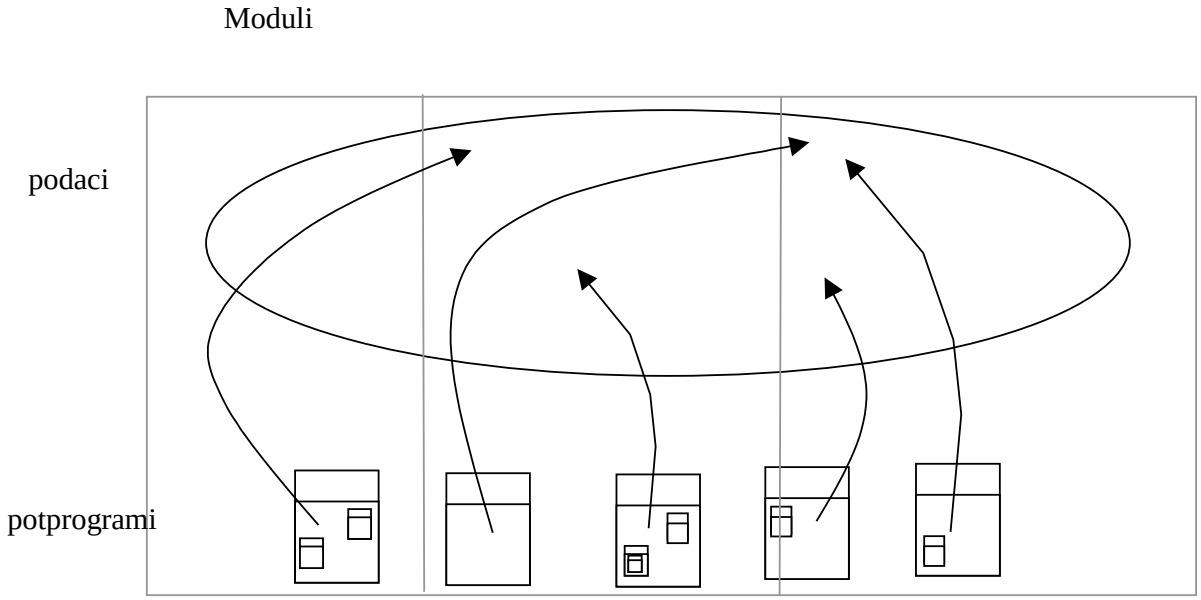
Takva primena potprograma je imala neke važne rezultate. Razvijeni su jezici koji u sebi imaju ugrađen mehanizam prosleđivanja parametara. Pored toga, postavljene su osnove strukturalnog programiranja, odnosno u jezike je ugrađena mogućnost smeštanja potprograma, a razvijena je i teorija domena i vidljivosti deklaracije. Treće, razvijeni su metodi strukturalnog projektovanja, tako da su projektanti velikih sistema ohrabreni da potprograme koriste kao osnovne gradivne elemente.

Sa slike se vidi da topologija ovakvih jezika predstavlja varijaciju prethodnih generacija. Ovom topologijom su se rešili neki problemi prethodnih generacija, posebno bolja kontrolu nad algoritamskim apstrakcijama.



Slika 11 Topologija jezika kasne druge i rane treće generacije

Topologija jezika kasne treće generacije Počevši od FORTRAN-a II, sve više se brinulo o problemima koji nastaju kod programiranja velikih aplikacija. Veliki projekti traže mnogo programera koji moraju da rade u timovima, ali koji moraju i nezavisno da rade na pojedinim delovima programa. Rešenje je pronađeno u modulima koji se zasebno kompajliraju. Na početku je modul predstavljao običan kontejner podataka i potprograma.



Slika 12 Topologija jezika kasne treće generacije

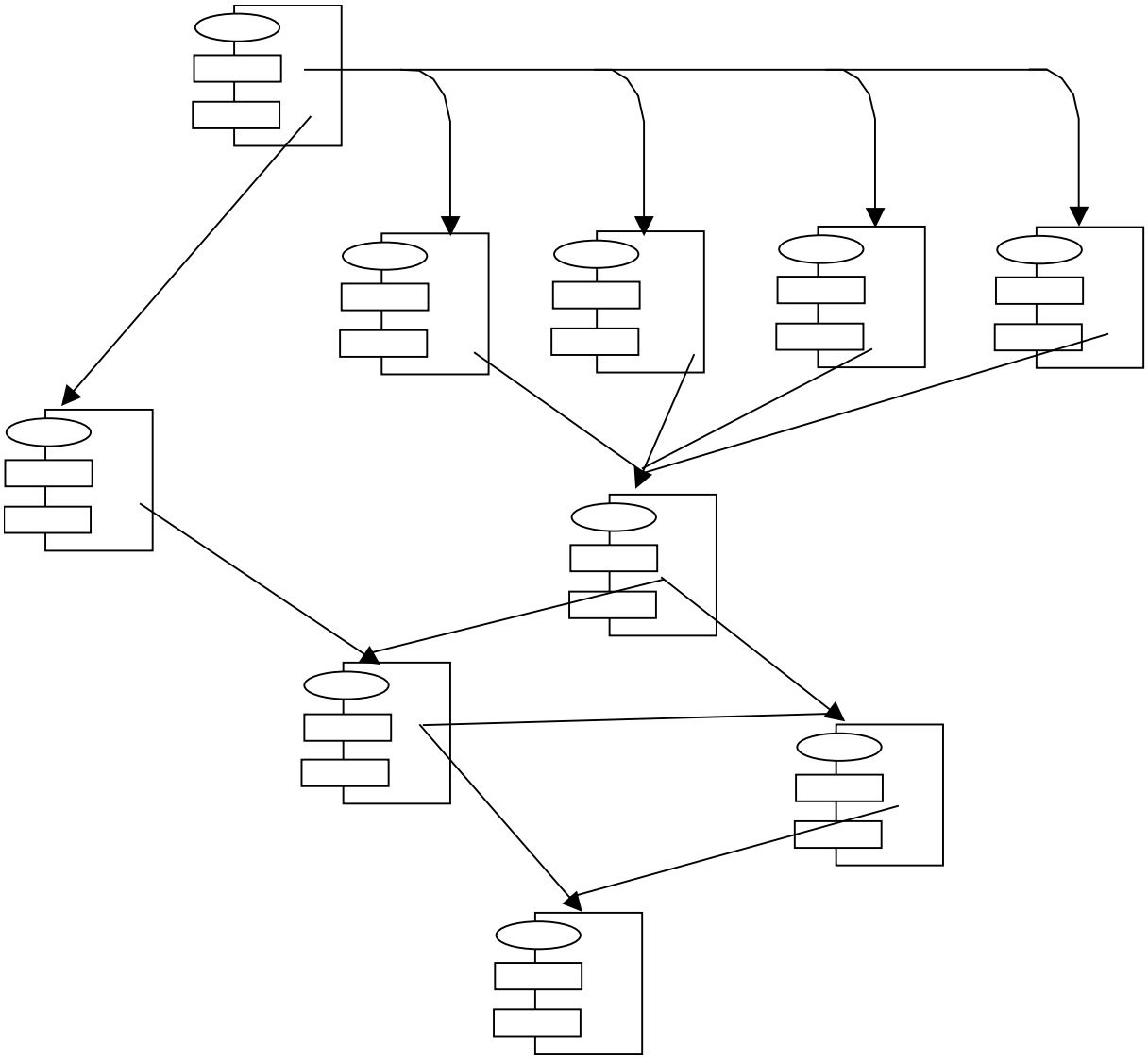
U toj prvoj fazi moduli nisu identifikovani kao mehanizam apstrakcije. U praksi su korišćeni za grupisanje potprograma koji će se, verovatno - zajedno menjati. Između modula nije bilo jasno

definisanih veza. Programer nekog potprograma iz jednog modula je mogao da poziva potprogram primenom jednih parametra, a neki drugi programer je mogao da isti potprogram poziva primenom drugih parametara. Pošto većina jezika ove generacije nije imala podršku apstrakcije podataka i jakih tipova, ovakve greške je bilo moguće otkriti tek prilikom izvršavanja programa.

Topologija objektno orijentisanih programskih jezika Važnost apstrakcije podataka u odnosu na složenost celog problema je jasno definisao Shankar: "Apstrakcije koje se mogu izraziti preko procedura su pogodne za opis apstraktnih operacija, ali nisu dobre za opis apstraktnih objekata. To je ozbiljan nedostatak, pošto složenost podataka (objekata) značajno utiče na ukupnu složenost problema." Rešavanje ovih problema je dovelo do razvoja metoda projektovanja usredsređenih na podatke, a razvijena su i teorijski – koncepti tipova.

Prva realizacija ovih ideja je ostvarena u jeziku ''Simula'', a detaljno razrađena u jezicima kao što su ''Smalltalk'', ''Objektni Pascal'', ''C++'', ''CLOS'' ili ''Ada''. Ovi jezici su nazvani *objektno-orijentisanim jezicima.*

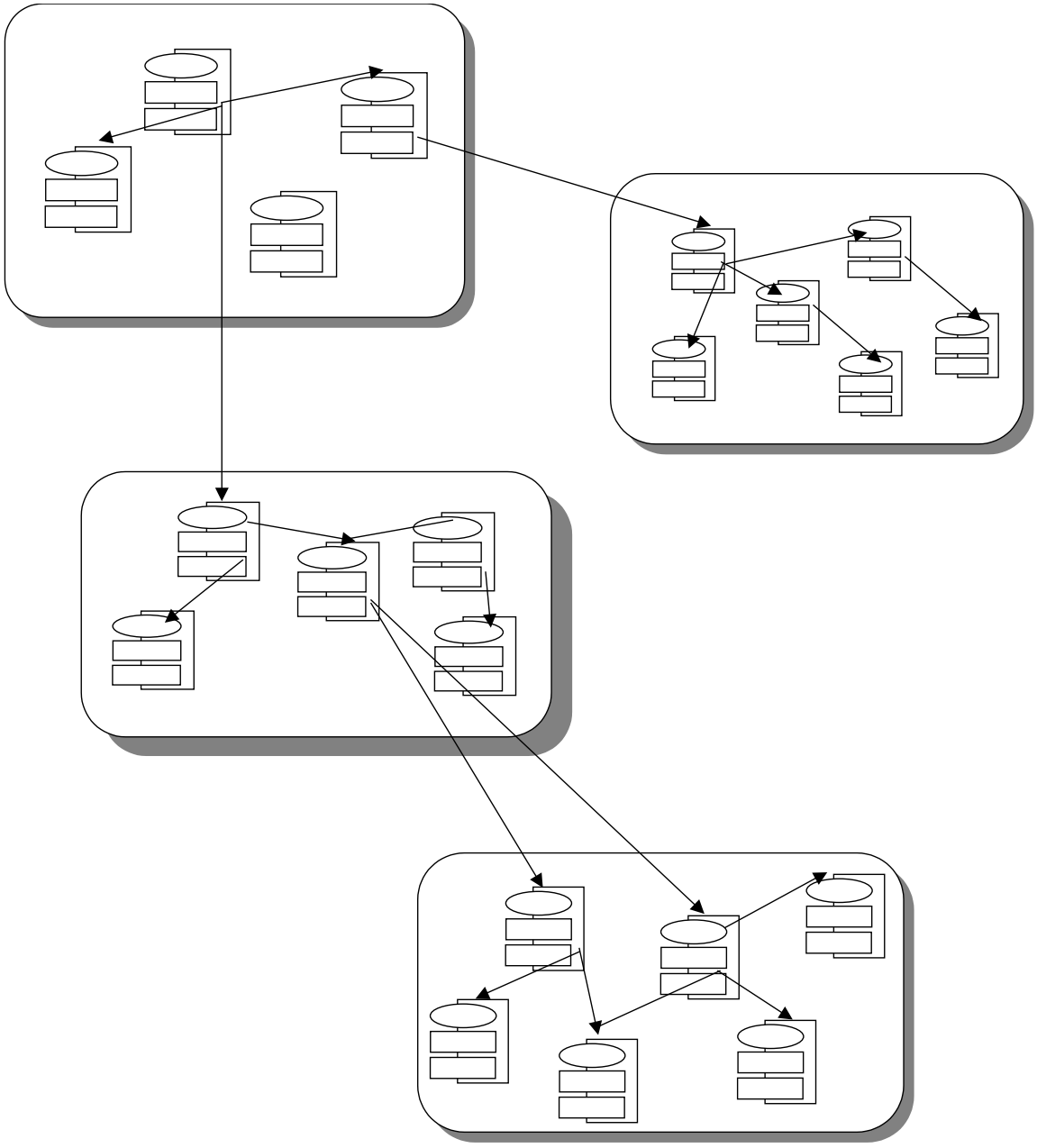
Na sledećoj slici je prikazana topologija malih i srednjih aplikacija, pisanih u ovim jezicima.



Slika 13: Topologija malih i srednjih aplikacija primenom OO tehnoologija

Osnovni gradivni element u ovim jezicima je modul koji predstavlja skup klasa i objekata, a ne potprograma, kao što je ranije bio slučaj. Pokušaćemo da ovo pojasnimo na sledeći način: Ako su procedure i funkcije glagoli, a podaci imenice, proceduralni program je orijentisan prema glagolima, dok je objektno orijentisani program orijentisan prema imenicama. To je razlog što fizička struktura objektno orijentisanih programa liči na graf, a ne na stablo - što je tipično za algoritamske jezike. Pored toga, ovde nema, ili ima vrlo malo globalnih podataka.

Objektno orijentisani jezici su pogodni i za velike aplikacije. Kod složenih sistema se može desiti da klase i objekti ne budu dovoljni za definisanje apstrakcija. U tom slučaju se može iskoristiti mogućnost ovakvih jezika da se naprave skupovi objekata koji zajedno ostvaruju određeno ponašanje definisano na višem nivou. Na sledećoj slici je prikazana topologija velikih aplikacija, rađenih u objektno orijentisanim jezicima.



Slika 14: Topologija velikih aplikacija primenom OO tehnika

Koncepti i paradigme programskih jezika

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

Programski jezik mora bti univerzalan. To znači da svaki problem mora imati rešenje koje se može isprogramirati u jeziku; naravno, ako je problem uopšte rešiv na računaru. Ovo možda izgleda kao strog zahtev, ali ga čak i mali programski jezici mogu ispuniti. Svaki jezik koji može definisati rekurzivne funkcije, univerzalan je. Sa druge strane, jezik koji nema ni rekurziju ni iteraciju, ne može

se smatrati univerzalnim. Programski jezik može imati oblast za koju je namenjen. Na primer, programski jezik čiji su jedini tipovi podataka brojevi i nizovi, može se koristiti za rešavanje numeričkih problema, ali verovatno neće biti pogodan za primenu u veštačkoj inteligenciji.

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

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

Koliko su važni koncepti na kojima se zasnivaju programski jezici, toliko su važni i načini na koji se ti koncepti udružuju da bi formirali programski jezik. *Različiti izbori ključnih koncepata, koji podržavaju različite stilove programiranja - nazivaju se paradigmama.* Postoji šest glavnih paradigmi. To su

1. ***imperativno programiranje*** koje karakteriše upotreba promenljivih, komande i procedura; zatim
2. ***objektno orijentisano programiranje*** koje koristi objekte, klase i nasleđivanje;
3. ***uporedno programiranje*** koje koristi uporedne procese;
4. ***funkcionalno programiranje*** koje koristi funkcije;
5. ***logičko programiranje*** koje koristi relacije i postoje
6. ***skript jezici***, kao posebna kategorija.

Sintaksa i semantika programskih jezika

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

Sintaksa programskog jezika se odnosi na formu programa. Ona prezentuje način na koji se moraju urediti izrazi, komande, deklaracije i ostale programske konstrukcije - da bi se dobio ispravan program.

Semantika programskog jezika se odnosi na značenje programa. Ona ukazuje na to kako se od ispravnog programa može očekivati da se ponaša.

Pragmatika programskog jezika se odnosi na način na koji se jezik namerava da koristi u praksi.

Sintaksa određuje formu u kojoj programeri pišu programe, drugi programeri čitaju i račun obrađuje. Semantika određuje način na koji programer komponuje programe, kako ih drugi programeri razumeju i kako ih računar interpretira. Pragmatika utiče na to kako programeri očekuju da se programi projektuju i implementiraju u praksi. Sintaksa je važna, ali su semantika i pragmatika još važnije.

Ovo se može jasnije sagledati na primeru jednog eksperta koji razmišlja o rešenju datog problema. Programer prvo dekomponuje problem i identifikuje odgovarajuće programske jedinice (procedure, pakete, apstraktne tipove ili klase). Posle toga, on razmatra odgovarajuću primenu svake od programskih jedinica, određuje tipove podataka i promenljivih koji postoje u jeziku, definiše kontrolne strukture, izuzetke itd. Tek na kraju programer kodira program, tj. piše program u sintaksi određenog programskog jezika. Samo u ovoj završnoj fazi sintaksa programskog jezika postaje bitna. Vrlo je važno da primena sintakse određenog programa izvrši što kasnije u procesu razvoja nekog programa, jer svi nedostaci nekog programa nisu posledica sintakse (sem u slučaju elementarnih grešaka u njenoj primeni, što se lako otklanja), već u pogrešno postavljenoj semantici. Zato je projektovanje nekog programskog sistema mnogo značajnije nego čisto programiranje“ tj. kodira nje, kada se primenjuje sintaksa određenog programskog jezika da bi se semantika programa pretvorila u formu (sintaksu) koji prevodilac programa (interpreter ili kompajler) razume i koji prevodi u izvršni, mašinski program.