Filip Marić Predrag Janičić

Programiranje 1

Osnove programiranja kroz programski jezik C

Matematički fakultet

Filip Marić Predrag Janičić

PROGRAMIRANJE 1

Osnove programiranja kroz programski jezik C

Beograd
2022.

Autori:
dr Filip Marić, vanredni profesor na Matematičkom fakultetu u Beogradu
dr Predrag Janičić, redovni profesor na Matematičkom fakultetu u Beogradu

PROGRAMIRANJE 1

Izdavač: Matematički fakultet Univerziteta u Beogradu
Studentski trg 16, 11000 Beograd
Za izdavača: *prof. dr Zoran Rakić*, dekan

Recenzenti:
dr Gordana Pavlović-Lažetić, redovni profesor na Matematičkom fakultetu u Beogradu
dr Miodrag Živković, redovni profesor na Matematičkom fakultetu u Beogradu
dr Dragan Urošević, naučni savetnik na Matematičkom institutu SANU

Obrada teksta, crteži i korice: *autori*

ISBN 978-86-7589-100-0

Sadržaj

Predgovor

Ova knjiga pisana je kao udžbenik za predmet *Programiranje 1* na smeru *Informatika* Matematičkog fakulteta u Beogradu. U ovom predmetu i u ovoj knjizi, centralno mesto ima programski jezik C, ali predmet i knjiga nisu samo kurs ovog jezika, već pokušavaju da dâju šire osnove programiranja, ilustrovane kroz jedan konkretan jezik.

Knjiga je nastala na osnovu materijala za predavanja koja smo na ovom predmetu držali od 2005. godine. Ipak, vremenom je materijal proširen i delovima koji se u okviru tog predmeta ne predaju ili se predaju u vrlo ograničenom obimu. Zahvaljujući tome, ova knjiga i njen nastavak (*Programiranje 2 — Osnove programiranja kroz programski jezik C*) mogu se koristiti kao udžbenici za više različitih kurseva. Za predmet na studijama na kojima nema drugih (ili nema mnogo drugih) računarskih predmeta, predlažemo obrađivanje čitave knjige, s tim što obrađivanje glave 4 preporučujemo samo za studije sa jakom matematičkom orijentacijom. Čitaocima (studentima) koji poznaju osnove računarstva i programiranja, a ne znaju jezik C, preporučujemo čitanje samo drugog dela knjige (*Jezik C*). Za kurs *Programiranje 1* na smeru Informatika preporučujemo samo ubrzano upoznavanje sa glavama 1 i 2 (jer se ti sadržaji izučavaju u okviru drugih predmeta na prvoj godini). Za ovaj kurs preporučujemo i upoznavanje sa glavama 3 i 4, a upoznavanje sa glavama 10 i 12 preporučujemo za drugi semestar.

Na kraju većine poglavlja naveden je veći broj pitanja i zadataka koji mogu da služe za proveru znanja. Među ovim pitanjima su praktično sva pitanja koja su zadata na testovima i ispitima iz predmeta *Programiranje 1* u periodu od pet godina. Odgovori na pitanja nisu eksplicitno navođeni, jer su već implicitno sadržani u osnovnom tekstu knjige. Na kraju knjige navedena su rešenja zadataka, te se može smatrati da ova knjiga obuhvata i potpunu prateću zbirku zadataka. Za zadatka tipa „šta ispisuje naredni program?“ nisu navođeni odgovori jer čitalac to može da proveri na svom računaru (i knjiga pokušava da ohrabri čitaoca da čitanje knjige kombinuje sa radom na računaru).

U pripremi knjige koristili smo mnoge izvore, pre svega sa interneta. Od izvora o jeziku C, pomenimo ovde znamenitu knjigu *Programski jezik C* (The C Programming Language, K&R) Brajana Kernigena i Denisa Ričija koja je dugo služila kao nezvanični standard jezik, knjigu *Knjiga o C-u* (The C Book) Majka Banahana, Deklana Brejdija i Marka Dorana, kao i ISO standarde jezika C. Za pripremu glave *Algoritmi i izračunljivost* koristili smo knjigu *Teorija algoritama, jezika i automata - zbirka zadataka*, Irene Spasić i Predraga Janičića.

Na veoma pažljivom čitanju i brojnim korisnim savetima zahvaljujemo recenzentima Gordani Pavlović-Lažetić, Miodragu Živkoviću i Draganu Uroševiću. Na brojnim sugestijama i ispravkama zahvalni smo i nastavnicima Matematičkog fakulteta Mileni Vujošević-Janičić, Nenadu Mitiću i Mladenu Nikoliću, kao i studentima Nikoli Premčevskom, Mladenu Canoviću, Nemanji Mićoviću, Vojislavu Grujiću, Stefanu Đorđeviću, Petru Vukmiroviću, Bobanu Piskuliću, Jani Protić, Ljubici Peleksić, Ivanu Baleviću, Danielu Doži, Milošu Samardžiji, Stefanu Saviću, Petru Kovrliji i Tomislavu Milovanoviću.

Ova knjiga dostupna je (besplatno) u elektronskom obliku preko internet strana autora. Sadržaj štampanog i elektronskog izdanja je identičan. Besplatna dostupnost elektronskog oblika knjige odražava stav autora o otvorenim sadržajima — kodu programa i sadržaju knjiga.

Beograd, april 2015. godine *Autori*

Predgovor drugom izdanju

U drugom izdanju nema novog materijala, već su samo ispravljene greške uočene u prvom izdanju. Na ispravkama i sugestijama zahvalni smo nastavnici Matematičkog fakulteta Vesni Marinković, kao i studentima Dalmi

Beara, Đorđu Stanojeviću i Marku Spasiću. I ovo, drugo izdanje knjige dostupno je (besplatno) u elektronskom obliku preko internet strana autora. Sadržaj štampanog i elektronskog izdanja je identičan.

Beograd, septembar 2017. godine *Autori*

Predgovor trećem izdanju

U trećem izdanju ispravljene su greške uočene u drugom. Na ispravkama i sugestijama zahvalni smo studentima Đorđu Milićeviću i Darku Zoriću. I ovo, treće izdanje knjige dostupno je (besplatno) u elektronskom obliku preko internet strana autora. Sadržaj štampanog i elektronskog izdanja je identičan.

Beograd, septembar 2018. godine *Autori*

Predgovor četvrtom izdanju

U četvrtom izdanju ispravljene su greške uočene u trećem i dodato nekoliko objašnjenja. Na ispravkama i sugestijama zahvalni smo studentima Luki Jovičiću i Veljku Kučinaru. I ovo, četvrto izdanje knjige dostupno je (besplatno) u elektronskom obliku preko internet strana autora. Sadržaj štampanog i elektronskog izdanja je identičan.

Beograd, septembar 2019. godine *Autori*

Predgovor petom izdanju

U petom izdanju ispravljene su greške, uglavnom sitne, uočene u četvrtom izdanju. Na ispravkama i sugestijama zahvalni smo nastavnicima Matematičkog fakulteta Jeleni Graovac i Ivanu Čukiću, kao i studentima Aleksandru Zečeviću i Stefanu Škanati. I ovo izdanje knjige dostupno je (besplatno) u elektronskom obliku preko internet strana autora.

Beograd, septembar 2021. godine *Autori*

Predgovor šestom izdanju

U šestom izdanju ispravljene su greške, uglavnom sitne, uočene u petom izdanju. Na novim komentarima i sugestijama zahvalni smo koleginici Jeleni Graovac i kolegi Ivanu Čukiću. I ovo izdanje knjige dostupno je (besplatno) u elektronskom obliku preko internet strana autora.

Beograd, april 2022. godine *Autori*

Deo I

Osnovni pojmovi računarstva i programiranja

Glava 1

Računarstvo i računarski sistemi

Računarstvo i informatika predstavljaju jednu od najatraktivnijih i najvažnijih oblasti današnjice. Život u savremenom društvu ne može se zamisliti bez korišćenja različitih *računarskih sistema*: stonih i prenosnih računara, tableta, pametnih telefona, ali i računara integrisanih u različite mašine (automobile, avione, industrijske mašine, itd). Definicija računarskog sistema je prilično široka. Može se reći da se danas pod digitalnim računarskim sistemom (računarom) podrazumeva mašina koja može da se programira da izvršava različite zadatke svođenjem na elementarne operacije nad brojevima. Brojevi se, u savremenim računarima, zapisuju u binarnom sistemu, kao nizovi nula i jedinica tj. binarnih cifara, tj. *bitova* (engl. bit, od *binary digit*). Koristeći n bitova, može se zapisati 2^n različitih vrednosti. Na primer, jedan *bajt* (B) označava osam bitova i može da reprezentuje 2^8, tj. 256 različitih vrednosti.[1]

Računarstvo se bavi izučavanjem računara, ali i opštije, izučavanjem teorije i prakse procesa računanja i primene računara u raznim oblastima nauke, tehnike i svakodnevnog života[2].

Računari u današnjem smislu nastali su polovinom XX veka, ali koreni računarstva su mnogo stariji od prvih računara. Vekovima su ljudi stvarali mehaničke i elektromehaničke naprave koje su mogle da rešavaju neke numeričke zadatke. Današnji računari su *programabilni*, tj. mogu da se isprogramiraju da vrše različite zadatke. Stoga je oblast *programiranja*, kojom se ova knjiga bavi, jedna od najznačajnijih oblasti računarstva. Za funkcionisanje modernih računara neophodni su i *hardver* i *softver*. Hardver (tehnički sistem računara) čine opipljive, fizičke komponente računara: procesor, memorija, matična ploča, hard disk, DVD uređaj, itd. Softver (programski sistem računara) čine računarski programi i prateći podaci koji određuju izračunavanja koja vrši računar. Računarstvo je danas veoma široka i dobro utemeljena naučna disciplina sa mnoštvom podoblasti.

1.1 Rana istorija računarskih sistema

Programiranje u savremenom smislu postalo je praktično moguće tek krajem Drugog svetskog rata, ali je njegova istorija znatno starija. Prvi precizni postupci i sprave za rešavanje matematičkih problema postojali su još u vreme antičkih civilizacija. Na primer, kao pomoć pri izvođenju osnovnih matematičkih operacija korišćene su računaljke zvane *abakus*. U IX veku persijski matematičar *Al Horezmi*[3] precizno je opisao postupke računanja u indo-arapskom dekadnom brojevnom sistemu (koji i danas predstavlja najkorišćeniji brojevni sistem). U XIII veku *Leonardo Fibonači*[4] doneo je ovaj način zapisivanja brojeva iz Azije u Evropu i to je bio jedan od ključnih preduslova za razvoj matematike i tehničkih disciplina tokom renesanse. Otkriće logaritma omogućilo je svođenje množenja na sabiranje, dodatno olakšano raznovrsnim analognih spravama (npr. *klizni lenjir — šiber*)[5]. Prve mehaničke sprave koje su mogle da potpuno automatski izvode aritmetičke operacije i pomažu u rešavanju matematičkih zadataka su napravljene u XVII veku. *Blez Paskal*[6] konstruisao je 1642. godine mehaničke sprave, kasnije nazvane *Paskaline*, koje su služile za sabiranje i oduzimanje celih brojeva. *Gotfrid Lajbnic*[7] konstruisao je 1672. godine mašinu koja je mogla da izvršava sve četiri osnovne aritmetičke operacije (sabiranje, oduzimanje,

[1] Količina podataka i kapacitet memorijskih komponenti savremenih računara obično se iskazuje u bajtovima ili izvedenim jedinicama. Obično se smatra da je jedan *kilobajt* (KB) jednak 1024 bajtova (mada neke organizacije podrazumevaju da je jedan KB jednak 1000 bajtova). Slično, jedan *megabajt* (MB) je jednak 1024 KB ili 1024^2 B, jedan *gigabajt* (GB) je jednak 1024 MB, a jedan *teraabajt* (TB) je jednak 1024 GB.

[2] Često se kaže da se računarstvo bavi računarima isto onoliko koliko se astronomija bavi teleskopima, a biologija mikroskopima. Računari nisu sami po sebi svrha i samo su sredstvo koje treba da pomogne u ostvarivanju različitih zadataka.

[3] Muhammad ibn Musa al-Khwarizmi (780–850), persijski matematičar.

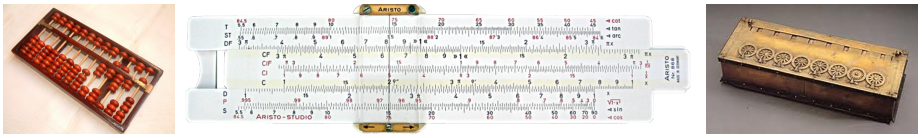
[4] Leonardo Pisano Fibonacci, (1170–1250), italijanski matematičar iz Pize.

[5] Zanimljivo je da su klizni lenjiri nošeni na pet Apolo misija, uključujući i onu na Mesec, da bi astronautima pomagali u potrebnim izračunavanjima.

[6] Blaise Pascal (1623–1662), francuski filozof, matematičar i fizičar. U njegovu čast jedan programski jezik nosi ime *PASCAL*.

[7] Gottfried Wilhelm Leibniz (1646–1716), nemački filozof i matematičar.

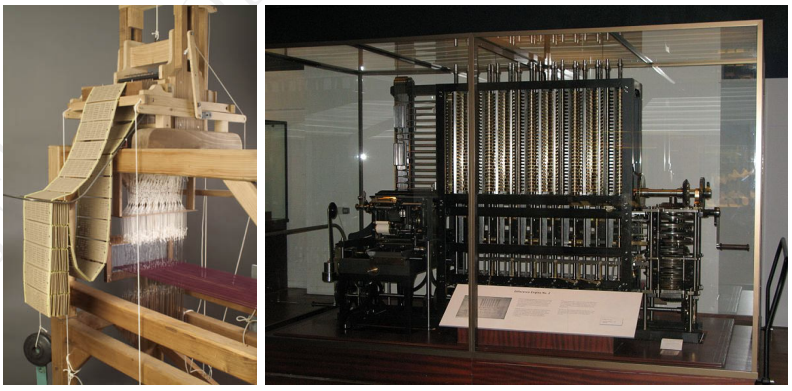
množenje i deljenje) nad celim brojevima. Ova mašina bila je zasnovana na dekadnom brojevnom sistemu, ali Lajbnic je prvi predlagao i korišćenje binarnog brojevnog sistema u računanju.



Slika 1.1: Abakus. Šiber. Paskalina

Mehaničke mašine. *Žozef Mari Žakard*[8] konstruisao je 1801. godine prvu programabilnu mašinu — mehanički tkački razboj koji je koristio bušene kartice kao svojevrsne programe za generisanje kompleksnih šara na tkanini. Svaka rupa na kartici određivala je jedan pokret mašine, a svaki red na kartici odgovarao je jednom redu šare.

U prvoj polovini XIX veka, *Čarls Bebidž*[9] dizajnirao je, mada ne i realizovao, prve programabilne računske mašine. Godine 1822. započeo je rad na *diferencijskoj mašini* koja je trebalo da računa vrednosti polinomijalnih funkcija (i eliminiše česte ljudske greške u tom poslu) u cilju izrade što preciznijih logaritamskih tablica. Ime je dobila zbog toga što je koristila tzv. metod konačnih razlika da bi bila eliminisana potreba za množenjem i deljenjem. Mašina je trebalo da ima oko 25000 delova i da se pokreće ručno, ali nije nikada završena[10]. Ubrzo nakon što je rad na prvom projektu utihnuo bez rezultata, Bebidž je započeo rad na novoj mašini nazvanoj *analitička mašina*. Osnovna razlika u odnosu na sve prethodne mašine, koje su imale svoje specifične namene, bila je u tome što je analitička mašina zamišljena kao računska mašina opšte namene koja može da se *programira* (programima zapisanim na bušenim karticama, sličnim Žakardovim karticama). Program zapisan na karticama kontrolisao bi mehanički računar (pokretan parnom mašinom) i omogućavao sekvencijalno izvršavanje naredbi, grananje i skokove, slično programima za savremene računare. Osnovni delovi računara trebalo je da budu *mlin* (engl. *mill*) i *skladište* (engl. *store*), koji po svojoj funkcionalnosti sasvim odgovaraju procesoru i memoriji današnjih računara. *Ada Bajron*[11] zajedno sa Bebidžem napisala je prve programe za analitičku mašinu i, da je mašina uspešno konstruisana, njeni programi bi mogli da računaju određene složene nizove brojeva (takozvane Bernulijeve brojeve). Zbog ovoga se ona smatra prvim programerom u istoriji (i njoj u čast jedan programski jezik nosi ime *Ada*). Ona je bila i prva koja je uvidela da se računske mašine mogu upotrebiti i za nematematičke namene, čime je na neki način anticipirala današnje namene digitalnih računara.



Slika 1.2: Žakardov razboj. Bebidževa diferencijska mašina.

Interesantno je da je krajem XIX veka naš veliki matematičar Mihailo Petrović Alas konstruisao *hidrointegrator* – analogni hidraulični računar koji je mogao da rešava određene klase diferencijalnih jednačina i koji je nagrađen na Svetskoj izložbi u Parizu 1900. godine.

[8] Joseph Marie Jacquard (1752–1834), francuski trgovac.

[9] Charles Babbage (1791–1871), engleski matematičar, filozof i pronalazač.

[10] Dosledno sledeći Bebidžev dizajn, 1991. godine (u naučno-popularne svrhe) uspešno je konstruisana diferencijska mašina koja radi besprekorno. Nešto kasnije, konstruisan je i „štampač“ koji je Bebidž dizajnirao za diferencijsku mašinu, tj. štamparska presa povezana sa parnom mašinom koja je štampala izračunate vrednosti.

[11] Augusta Ada King (rođ. Byron), Countess of Lovelace, (1815–1852), engleska matematičarka. U njenu čast nazvan je programski jezik ADA.

Elektromehaničke mašine. Elektromehaničke mašine za računanje koristile su se od sredine XIX veka do vremena Drugog svetskog rata.

Jedna od prvih je mašina za čitanje bušenih kartica koju je konstruisao *Herman Holerit*[12]. Ova mašina korišćena je 1890. za obradu rezultata popisa stanovništva u SAD. Naime, obrada rezultata popisa iz 1880. godine trajala je više od 7 godina, a zbog naglog porasta broja stanovnika procenjeno je da bi obrada rezultata iz 1890. godine trajala više od 10 godina, što je bilo neprihvatljivo mnogo. Holerit je sproveo ideju da se podaci prilikom popisa zapisuju na mašinski čitljivom medijumu (na bušenim karticama), a da se kasnije obrađuju njegovom mašinom. Koristeći ovaj pristup obrada rezultata popisa uspešno je završena za godinu dana. Od Holeritove male kompanije kasnije je nastala čuvena kompanija *IBM*.

Godine 1941, *Konrad Cuze*[13] konstruisao je 22-bitnu mašinu za računanje *Z3* koja je imala izvesne mogućnosti programiranja (podržane su bile petlje, ali ne i uslovni skokovi), te se često smatra i prvim realizovanim programabilnim računarom[14]. Cuzeove mašine tokom Drugog svetskog rata naišle su samo na ograničene primene. Cuzeova kompanija proizvela je oko 250 različitih tipova računara do kraja šezdesetih godina, kada je postala deo kompanije *Simens* (nem. Siemens).

U okviru saradnje kompanije IBM i univerziteta Harvard, tim *Hauarda Ejkena*[15] završio je 1944. godine mašinu *Harvard Mark I*. Ova mašina čitala je instrukcije sa bušene papirne trake, imala je preko 760000 delova, dužinu 17m, visinu 2.4m i masu 4.5t. Mark I mogao je da pohrani u memoriji (korišćenjem elektromehaničkih prekidača) 72 broja od po 23 dekadne cifre. Sabiranje i oduzimanje dva broja trajalo je trećinu, množenje šest, a deljenje petnaest sekundi.

Slika 1.3: Holeritova mašina. Harvard Mark I. ENIAC (proces reprogramiranja).

Elektronski računari. Elektronski računari koriste se od kraja 1930-ih do danas.

Jedan od prvih elektronskih računara *ABC* (specijalne namene — rešavanje sistema linearnih jednačina) napravili su 1939. godine *Atanasov*[16] i *Beri*[17]. Mašina je prva koristila binarni brojevni sistem i električne kondenzatore (engl. capacitor) za skladištenje bitova — sistem koji se u svojim savremenim varijantama koristi i danas u okviru tzv. DRAM memorije. Mašina nije bila programabilna.

Krajem Drugog svetskog rata, u Engleskoj, u *Blečli parku* (engl. *Bletchley Park*) u kojem je radio i *Alan Tjuring*[18], konstruisan je računar *Kolos* (engl. *Colossus*) namenjen dešifrovanju nemačkih poruka. Računar je omogućio razbijanje nemačke šifre zasnovane na mašini *Enigma*, zahvaljujući čemu su saveznici bili u stanju da prate komunikaciju nemačke podmorničke flote, što je značajno uticalo na ishod Drugog svetskog rata.

U periodu između 1943. i 1946. godine od strane američke vojske i tima univerziteta u Pensilvaniji koji su predvodili *Džon Mokli*[19] i *Džej Ekert*[20] konstruisan je prvi elektronski računar opšte namene — *ENIAC („Electronic Numerical Integrator and Calculator")*. Imao je 1700 vakuumskih cevi, dužinu 30m i masu 30t.

[12] Herman Hollerith (1860–1929), američki pronalazač.
[13] Konrad Zuse (1910–1995), nemački inženjer.
[14] Mašini Z3 prethodile su jednostavnije mašine Z1 i Z2, izgrađene 1938. i 1940. godine.
[15] Howard Hathaway Aiken (1900–1973).
[16] John Vincent Atanasoff (1903–1995).
[17] Clifford Edward Berry (1918–1963).
[18] Alan Turing (1912–1954), britanski matematičar.
[19] John William Mauchly (1907–1980).
[20] J. Presper Eckert (1919–1995).

Računske operacije izvršavao je hiljadu puta brže od elektromehaničkih mašina. Osnovna svrha bila mu je jedna specijalna namena — računanje trajektorije projektila. Bilo je moguće da se mašina preprogramira i za druge zadatke ali to je zahtevalo intervencije na preklopnicima i kablovima koje su mogle da traju danima.

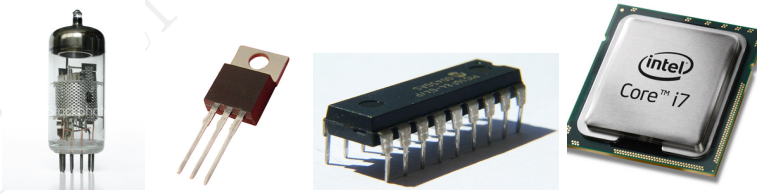
1.2 Računari Fon Nojmanove arhitekture

Rane mašine za računanje nisu bile programabilne već su radile po unapred fiksiranom programu, određenom samom konstrukcijom mašine. Takva arhitektura se i danas koristi kod nekih jednostavnih mašina, na primer, kod kalkulatora („digitrona"). Da bi izvršavali nove zadatke, rani elektronski računari nisu programirani u današnjem smislu te reči, već su suštinski redizajnirani. Tako su, na primer, operaterima bile potrebne nedelje da bi prespojili kablove u okviru kompleksnog sistema ENIAC i tako ga instruisali da izvršava novi zadatak.

Potpuna konceptualna promena došla je kasnih 1940-ih, sa pojavom računara koji programe na osnovu kojih rade čuvaju u memoriji zajedno sa podacima — *računara sa skladištenim programima* (engl. *stored program computers*). U okviru ovih računara, postoji jasna podela na hardver i softver. Iako ideje za ovaj koncept datiraju još od Čarlsa Bebidža i njegove analitičke mašine i nastavljaju se kroz radove Tjuringa, Cuzea, Ekerta, Moklija, za rodonačelnika ovakve arhitekture računara smatra se *Džon fon Nojman*[21]. Fon Nojman se u ulozi konsultanta priključio timu Ekerta i Moklija i 1945. godine je u svom izveštaju *EDVAC („Electronic Discrete Variable Automatic Computer")* opisao arhitekturu budućeg računara EDVAC u kojoj se programi mogu učitavati u memoriju isto kao i podaci koji se obrađuju. Ta arhitektura se od tada koristi u najvećem broju računara a jedan od prvih računara zasnovanih na njoj bio je sâm računar EDVAC (iako je dizajn računara EDVAC bio prvi opis Fon Nojmanove arhitekture, pre njegovog puštanja u rad 1951. godine, već je bilo konstruisano i funkcionalno nekoliko računara slične arhitekture). Računar EDVAC, naslednik računara ENIAC, koristio je binarni zapis brojeva i u memoriju je mogao da upiše hiljadu 44-bitnih podataka.

Osnovni elementi Fon Nojmanove arhitekture računara su *procesor* (koji čine aritmetičko-logička jedinica, kontrolna jedinica i registri) i *glavna memorija*, koji su međusobno povezani. Ostale komponente računara (npr. ulazno-izlazne jedinice, spoljašnje memorije, ...) smatraju se pomoćnim i povezuju se na centralni deo računara koji čine procesor i glavna memorija. Sva obrada podataka vrši se u procesoru. U memoriju se skladište podaci koji se obrađuju, ali i programi, predstavljeni nizom elementarnih instrukcija (kojima se procesoru zadaje koju akciju ili operaciju da izvrši). I podaci i programi se zapisuju obično kao binarni sadržaj i nema nikakve suštinske razlike između zapisa programa i zapisa podataka. Tokom rada, podaci i programi se prenose između procesora i memorije. S obzirom na to da i skoro svi današnji računari imaju Fon Nojmanovu arhitekturu, način funkcionisanja ovakvih računara biće opisan detaljnije u poglavlju o savremenim računarskim sistemima.

Moderni programabilni računari se, po pitanju tehnologije koju su koristili, mogu grupisati u četiri generacije, sve zasnovane na Fon Nojmanovoj arhitekturi.



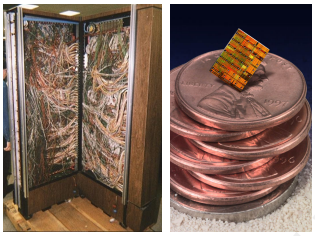
Slika 1.4: Osnovni gradivni elementi korišćeni u četiri generacije računara: vakuumska cev, tranzistor, integrisano kolo i mikroprocesor

I generacija računara (od kraja 1930-ih do kraja 1950-ih) koristila je *vakuumske cevi* kao logička kola i *magnetne doboše* (a delom i magnetne trake) za memoriju. Za programiranje su korišćeni mašinski jezik i asembler a glavne primene su bile vojne i naučne. Računari su uglavnom bili unikatni (tj. za većinu nije postojala serijska proizvodnja). Prvi realizovani računari Fon Nojmanove arhitekture bili su *Mančesterska „Beba"* (engl. *Manchester „Baby"*) — eksperimentalna mašina, razvijena 1949. na Univerzitetu u Mančesteru, na kojoj je testirana tehnologija vakuumskih cevi i njen naslednik *Mančesterski Mark 1* (engl. *Manchester Mark 1*), *EDSAC* — razvijen 1949. na Univerzitetu u Kembridžu, MESM razvijen 1950. na Kijevskom elektrotehničkom institutu i EDVAC koji je prvi dizajniran, ali napravljen tek 1951. na Univerzitetu u Pensilvaniji. Tvorci računara EDVAC, počeli su 1951. godine proizvodnju prvog komercijalnog računara *UNIVAC – UNIVersal Automatic Computer* koji je prodat u, za to doba neverovatnih, 46 primeraka.

[21] John Von Neumann (1903–1957), američki matematičar.

II generacija računara (od kraja 1950-ih do polovine 1960-ih) koristila je *tranzistore* umesto vakuumskih cevi. Iako je tranzistor otkriven još 1947. godine, tek sredinom pedesetih počinje da se koristi umesto vakuumskih cevi kao osnovna elektronska komponenta u okviru računara. Tranzistori su izgrađeni od tzv. *poluprovodničkih elemenata* (obično silicijuma ili germanijuma). U poređenju sa vakuumskih cevima, tranzistori su manji, zahtevaju manje energije te se manje i greju. Tranzistori su unapredili ne samo procesore i memoriju već i spoljašnje uređaje. Počeli su da se široko koriste magnetni diskovi i trake, započeto je umrežavanje računara, pa čak i korišćenje računara u zabavne svrhe (implementirana je prva računarska igra *Spacewar* za računar *PDP-1*). U ovo vreme razvijeni su i prvi jezici višeg nivoa (FORTRAN, LISP, ALGOL, COBOL). U to vreme kompanija IBM dominirala je tržištem — samo računar *IBM 1401*, prodat u više od deset hiljada primeraka, pokrivao je oko trećinu tada postojećeg tržišta.

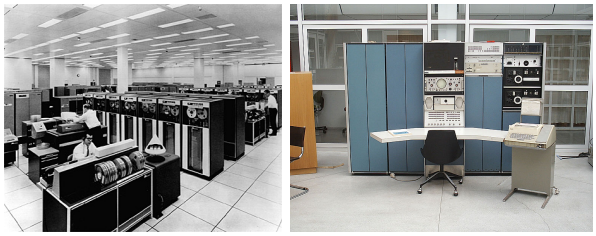
III generacija računara (od polovine 1960-ih do sredine 1970-ih) bila je zasnovana na *integrisanim kolima* smeštenim na silicijumskim *(mikro)čipovima*. Prvi računar koji je koristio ovu tehnologiju bio je IBM 360, napravljen 1964. godine.



Slika 1.5: Integrisana kola dovela su do minijaturizacije i kompleksni žičani spojevi su mogli biti realizovani na izuzetno maloj površini.

Nova tehnologija omogućila je poslovnu primenu računara u mnogim oblastima. U ovoj eri dominirali su *mejnfrejm* (engl. *mainframe*) računari koji su bili izrazito moćni za to doba, čija se brzina merila milionima instrukcija u sekundi (engl. MIPS; na primer, neki podmodeli računara IBM 360 imali su brzinu od skoro 1 MIPS) i koji su imali mogućnost skladištenja i obrade velike količine podataka te su korišćeni od strane vlada i velikih korporacija za popise, statističke obrade i slično. Kod računara ove generacije uveden je sistem *deljenja vremena* (engl. *timesharing*) koji dragoceno procesorsko vreme raspodeljuje i daje na uslugu različitim korisnicima koji istovremeno rade na računaru i komuniciraju sa njim putem specijalizovanih *terminala*. U ovo vreme uvedeni su prvi standardi za jezike višeg nivoa (npr. ANSI FORTRAN). Korišćeni su različiti operativni sistemi, uglavnom razvijeni u okviru kompanije IBM. Sa udelom od 90%, kompanija IBM je imala apsolutnu dominaciju na tržištu ovih računara.

Pored mejnfrejm računara, u ovom periodu široko su korišćeni i *mini računari* (engl. *minicomputers*) koji se mogu smatrati prvim oblikom ličnih (personalnih) računara. Procesor je, uglavnom, bio na raspolaganju isključivo jednom korisniku. Obično su bili veličine ormana i retko su ih posedovali pojedinci (te se ne smatraju kućnim računarima). Tržištem ovih računara dominirala je kompanija *DEC – Digital Equipment Corporation* sa svojim serijama računara poput *PDP-8* i *VAX*. Za ove računare, obično se vezuje operativni sistem *Unix* i programski jezik *C* razvijeni u *Belovim laboratorijama* (engl. *Bell Laboratories*), a često i *hakerska*[22] kultura nastala na univerzitetu *MIT* (engl. *Massachusetts Institute of Technology*).



Slika 1.6: Mejnfrejm računar: IBM 7094. Mini računar: DEC PDP 7

[22]Termin *haker* se obično koristi za osobe koje neovlašćeno pristupaju računarskim sistemima, ali *hakeraj* kao programerska podkultura podrazumeva anti-autoritaran pristup razvoju softvera, obično povezan sa pokretom za slobodan softver. U oba slučaja, hakeri su pojedinici koji na inovativan način modifikuju postojeće hardverske i softverske sisteme.

I u Jugoslaviji su tokom 1960-ih i 1970-ih konstruisani elektronski računari. Pomenimo seriju računara CER (Cifarski elektronski računar) i računar TARA koji su razvijeni na institutu Mihajlo Pupin iz Beograda. Naša zemlja bila je u to vreme jedna od retkih koje su proizvodile elektronske računare.

IV generacija računara (od ranih 1970-ih) zasnovana je na visoko integrisanim kolima kod kojih je na hiljade kola smešeno na jedan silikonski čip. U kompaniji Intel 1971. godine napravljen je prvi *mikroprocesor Intel 4004* — celokupna centralna procesorska jedinica bila je smeštena na jednom čipu. Iako prvobitno namenjena za ugradnju u kalkulatore, ova tehnologija omogućila je razvoj brzih a malih računara pogodnih za ličnu tj. kućnu upotrebu.

Časopis *Popular electronics* nudio je 1975. godine čitaocima mogućnost naručivanja delova za sklapanje mikroračunara *MITS Altair 8800* zasnovanog na mikroprocesoru *Intel 8080* (nasledniku mikroprocesora *Intel 4004*). Interesovanje među onima koji su se elektronikom bavili iz hobija bio je izuzetno pozitivan i samo u prvom mesecu prodato je nekoliko hiljada ovih „uradi-sâm“ računara. Smatra se da je Altair 8800 bio inicijalna kapisla za „revoluciju mikroračunara“ koja je usledila narednih godina. Altair se vezuje i za nastanak kompanije *Microsoft* — danas jedne od dominantnih kompanija u oblasti proizvodnje softvera. Naime, prvi proizvod kompanije *Microsoft* bio je interpretator za programski jezik *BASIC* za Altair 8800.

Nakon Altaira pojavljuje se još nekoliko računarskih kompleta na sklapanje. Prvi mikroračunar koji je prodavan već sklopljen bio je *Apple*, na čijim temeljima je nastala istoimena kompanija, danas jedan od lidera na tržištu računarske opreme.

Kućni računari koristili su se sve više — uglavnom od strane entuzijasta — za jednostavnije obrade podataka, učenje programiranja i igranje računarskih igara. Kompanija *Commodore* je 1977. godine predstavila svoj računarom *Commodore PET* koji je zabeležio veliki uspeh. *Commodore 64*, jedan od najuspešnijih računara za kućnu upotrebu, pojavio se 1982. godine. Iz iste kompanije je i serija *Amiga* računara sa kraja 1980-ih i početka 1990-ih. Pored kompanije *Commodore*, značajni proizvođači računara toga doba bili su i *Sinclair* (sa veoma popularnim modelom *ZX Spectrum*), *Atari*, *Amstrad*, itd. Kućni računari ove ere bili su obično jeftini, imali su skromne karakteristike i najčešće koristili kasetofone i televizijske ekrane kao ulazno-izlazne uređaje.

Slika 1.7: Prvi mikroprocesor: Intel 4004. Naslovna strana časopisa *„Popular electronics“* sa Altair 8800. Commodore 64. IBM PC 5150.

Kućni „uradi-sam“ računari pravili su se i u Jugoslaviji. Najpoznatiji primer je računar „Galaksija“ iz 1983. godine koji je dizajnirao Voja Antonić.

Najznačajnija računarska kompanija toga doba — IBM — uključila se na tržište kućnih računara 1981. godine, modelom *IBM PC 5150*, poznatijem jednostavno kao *IBM PC* ili *PC* (engl. *Personal computer*). Zasnovan na Intelovom mikroprocesoru *Intel 8088*, ovaj računar veoma brzo je zauzeo tržište računara za ličnu poslovnu upotrebu (obrada teksta, tabelarna izračunavanja, ...). Prateći veliki uspeh IBM PC računara, pojavio se određen broj *klonova* — računara koji nisu proizvedeni u okviru kompanije IBM, ali koji su kompatibilni sa IBM PC računarima. PC arhitektura vremenom je postala standard za kućne računare. Sredinom 1980-ih, pojavom naprednijih grafičkih (VGA) i zvučnih (SoundBlaster) kartica, IBM PC i njegovi klonovi stekli su mogućnost naprednih multimedijalnih aplikacija i vremenom su sa tržišta istisli sve ostale proizvođače. I naslednici originalnog IBM PC računara (IBM PC/XT, IBM PC/AT, ...) bili su zasnovani na Intelovim mikroprocesorima, pre svega na *x86* seriji (Intel 80286, 80386, 80486) i zatim na seriji *Intel Pentium*. Operativni sistemi koji se tradicionalno vezuju za PC računare dolaze iz kompanije *Microsoft* — prvo *MS DOS*, a zatim *MS Windows*. PC arhitektura podržava i korišćenje drugih operativnih sistema (na primer, GNU/Linux).

Jedini veliki konkurent IBM PC arhitekturi koji se sve vreme održao na tržištu (pre svega u SAD) je serija računara *Macintosh* kompanije *Apple*. *Macintosh*, koji se pojavio 1984., je prvi komercijalni kućni računar sa grafičkim korisničkim interfejsom i mišem. Operativni sistem koji se i danas koristi na Apple računarima je *Mac OS*.

Iako su prva povezivanja udaljenih računara izvršena još krajem 1960-ih godina, pojavom *interneta* (engl. *internet*) i *veba* (engl. *World Wide Web — WWW*), većina računara postaje međusobno povezana sredinom 1990-ih godina. Danas se veliki obim poslovanja izvršava u internet okruženju, a domen korišćenja računara je veoma širok. Došlo je do svojevrsne informatičke revolucije koja je promenila savremeno društvo i svakodnevni život. Na primer, tokom prve decenije XXI veka došlo je do pojave *društvenih mreža* (engl. *social networks*) koje postepeno preuzimaju ulogu osnovnog medijuma za komunikaciju.

Slika 1.8: Stoni računar. Prenosni računar: IBM ThinkPad. Tablet: Apple Ipad 2. Pametni telefon: Samsung Galaxy S2.

Tržištem današnjih računara dominiraju računari zasnovani na *PC* arhitekturi i *Apple Mac* računari. Pored stonih (engl. desktop) računara popularni su i prenosni (engl. notebook ili laptop) računari. U najnovije vreme, javlja se trend *tehnološke konvergencije* koja podrazumeva stapanje različitih uređaja u jedinstvene celine, kao što su *tableti* (engl. *tablet*) i *pametni telefoni* (engl. *smartphone*). Operativni sistemi koji se danas uglavnom koriste na ovim uređajima su *IOS* kompanije Apple, kao i *Android* kompanije Google.

Pored ličnih računara u IV generaciji se i dalje koriste mejnfrejm računari (na primer, IBM Z serija) i superračunari (zasnovani na hiljadama procesora). Na primer, kineski superračunar Tianhe-2 radi brzinom od preko 30 petaflopsa (dok prosečni lični računar radi brzinom reda 10 gigaflopsa).[23]

1.3 Oblasti savremenog računarstva

Savremeno računarstvo ima mnogo podoblasti, kako praktičnih, tako i teorijskih. Zbog njihove isprepletenosti nije jednostavno sve te oblasti sistematizovati i klasifikovati. U nastavku je dat spisak nekih od oblasti savremenog računarstva (u skladu sa klasifikacijom američke asocijacije ACM — *Association for Computing Machinery*, jedne od najvećih i najuticajnijih računarskih zajednica):

- *Algoritmika* (procesi izračunavanja i njihova složenost);
- *Strukture podataka* (reprezentovanje i obrada podataka);
- *Programski jezici* (dizajn i analiza svojstava formalnih jezika za opisivanje algoritama);
- *Programiranje* (proces zapisivanja algoritama u nekom programskom jeziku);
- *Softversko inženjerstvo* (proces dizajniranja, razvoja i testiranja programa);
- *Prevođenje programskih jezika* (efikasno prevođenje viših programskih jezika, obično na mašinski jezik);

[23] Flops je mera računarskih performansi, posebno pogodna za izračunavanja nad brojevima u pokretnom zarezu (i pogodnija nego generička mera koja se odnosi na broj instrukcija u sekundi). Broj flopsa govori koliko operacija nad brojevima u pokretnom zarezu može da izvrši računar u jednoj sekundi. Brzina današnjih računara se obično izražava u gigaflopsima (10^9 flopsa), teraflopsima (10^{12} flopsa) i petaflopsima (10^{15} flopsa).

- *Operativni sistemi* (sistemi za upravljanje računarom i programima);
- *Mrežno računarstvo* (algoritmi i protokoli za komunikaciju između računara);
- *Primene* (dizajn i razvoj softvera za svakodnevnu upotrebu);
- *Istraživanje podataka* (pronalaženje relevantnih informacija u velikim skupovima podataka);
- *Veštačka inteligencija* (rešavanje problema u kojima se javlja kombinatorna eksplozija);
- *Robotika* (algoritmi za kontrolu ponašanja robota);
- *Računarska grafika* (analiza i sinteza slika i animacija);
- *Kriptografija* (algoritmi za zaštitu privatnosti podataka);
- *Teorijsko računarstvo* (teorijske osnove izračunavanja, računarska matematika, verifikacija softvera, itd).

1.4 Hardver savremenih računara

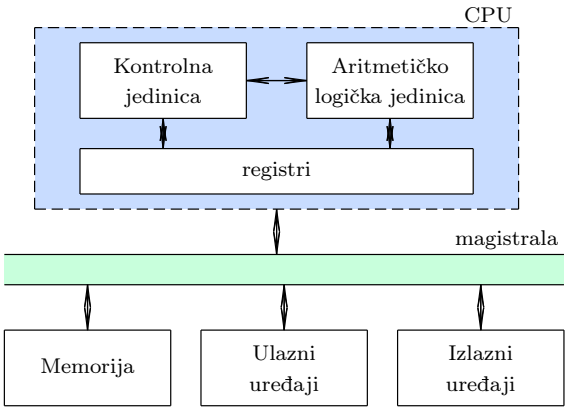
Hardver čine opipljive, fizičke komponente računara. Iako je u osnovi savremenih računarskih sistema i dalje Fon Nojmanova mašina (procesor i memorija), oni se danas ne mogu zamisliti bez niza hardverskih komponenti koje olakšavaju rad sa računarom.

Iako na prvi pogled deluje da se jedan uobičajeni stoni računar sastoji od kućišta, monitora, tastature i miša, ova podela je veoma površna, podložna promenama (već kod prenosnih računara, stvari izgledaju znatno drugačije) i nikako ne ilustruje koncepte bitne za funkcionisanje računara. Mnogo značajnija je podela na osnovu koje računar čine:

- *procesor tj. centralna procesorska jedinica* (engl. *Central Processing Unit, CPU*), koja obrađuje podatke;
- *glavna memorija* (engl. main memory), u kojoj se istovremeno čuvaju i podaci koji se obrađuju i trenutno pokrenuti programi (takođe zapisani binarno, u obliku podataka);
- različiti *periferijski uređaji* ili *ulazno-izlazne jedinice* (engl. *peripherals, input-output devices, IO devices*), kao što su miševi, tastature, ekrani, štampači, diskovi, a koje služe za komunikaciju korisnika sa sistemom i za trajno skladištenje podataka i programa.

Sve nabrojane komponente međusobno su povezane i podaci se tokom rada računara prenose od jedne do druge. Veza između komponenti uspostavlja se hardverskim sklopovima koji se nazivaju *magistrale* (engl. *bus*). Magistrala obuhvata provodnike koji povezuju uređaje, ali i čipove koji kontrolišu protok podataka. Svi periferijski uređaji se sa memorijom, procesorom i magistralama povezuju hardverskim sklopovima koji se nazivaju *kontrolori*. *Matična ploča* (engl. *motherboard*) je štampana ploča na koju se priključuju procesor, memorijski čipovi i svi periferijski uređaji. Na njoj se nalaze čipovi magistrale, a danas i mnogi kontrolori periferijskih uređaja. Osnovu hardvera savremenih računara, dakle, čine sledeće komponente:

Procesori. Procesor je jedna od dve centralne komponente svakog računarskog sistema Fon Nojmanove arhitekture. Svi delovi procesora su danas objedinjeni u zasebnu jedinicu (CPU) realizovanu na pojedinačnom čipu – mikroprocesoru. Procesor se sastoji od *kontrolne jedinice* (engl. *Control Unit*) koja upravlja njegovim radom i *aritmetičko-logičke jedinice* (engl. *Arithmetic Logic Unit*) koja je zadužena za izvođenje aritmetičkih operacija (sabiranje, oduzimanje, množenje, poređenje, ...) i logičkih operacija (konjunkcija, negacija, ...) nad brojevima. Procesor sadrži i određeni, manji broj, *registara* koji privremeno mogu da čuvaju podatke. Registri su obično fiksirane širine (8 bitova, 16 bitova, 32 bita, 64 bita). Komunikacija sa memorijom se ranije vršila isključivo preko specijalizovanog registra koji se nazivao akumulator. Aritmetičko logička jedinica sprovodi operacije nad podacima koji su smešteni u registrima i rezultate ponovo smešta u registre. Kontrolna jedinica procesora čita instrukciju po instrukciju programa zapisanog u memoriji i na osnovu njih određuje sledeću akciju sistema (na primer, izvrši prenos podataka iz procesora na određenu memorijsku adresu, izvrši određenu aritmetičku operaciju nad sadržajem u registrima procesora, uporedi sadržaje dva registra i ukoliko su jednaki izvrši instrukciju koja se nalazi na zadatoj memorijskoj adresi i slično). Brzina procesora meri se u *milionima operacija u sekundi* (engl. *Million Instructions Per Second, MIPS*) tj. pošto su operacije u pokretnom zarezu najzahtevnije, u *broju operacija u pokretnom zarezu u sekundi* (engl. *FLoating Point Operations per Second, FLOPS*). Današnji standardni procesori rade oko 10 GFLOPS (deset milijardi operacija u pokretnom zarezu po sekundi). Današnji procesori mogu da imaju i nekoliko *jezgara* (engl. *core*) koja istovremeno izvršavaju instrukcije i time omogućuju tzv. paralelno izvršavanje.



Slika 1.9: Shema računara Fon Nojmanove arhitekture

Važne karakteristike procesora danas su broj jezgara (obično 1, 2, 4, 6, 8, pa i više), širina reči (obično 32 bita ili 64 bita) i radni takt (obično nekoliko gigaherca (GHz)) — veći radni takt obično omogućava izvršavanje većeg broja operacija u jedinici vremena.

Za intenzivna izračunavanja sve više se koriste i specijalizovani *grafički procesori* (engl. *Graphics Processing Unit*, *GPU*). Iako su prvobitno bili namenjeni isključivo za izračunavanja u vezi sa grafikom računara, njihova visoko paralelizovana struktura čini ih pogodnim za različite oblike izračunavanja koja se vrše paralelno nad velikim blokovima podataka (koriste se, na primer, u oblastima mašinskog učenja, istraživanja i obrade velike količine podataka i slično).

Memorijska hijerarhija. Druga centralna komponenta Fon Nojmanove arhitekture je *glavna memorija* u koju se skladište podaci i programi. Memorija je linearno uređeni niz registara (najčešće bajtova), pri čemu svaki registar ima svoju adresu. Kako se kod ove memorije sadržaju može pristupati u slučajnom redosledu (bez unapred fiksiranog redosleda), ova memorija se često naziva i *memorija sa slobodnim pristupom* (engl. *random access memory, RAM*). Osnovni parametri memorija su *kapacitet* (danas obično meren gigabajtima (GB)), *vreme pristupa* koje izražava vreme potrebno da se memorija pripremi za čitanje odnosno upis podataka (danas obično mereno u nanosekundama (ns)), kao i *protok* koji izražava količinu podataka koji se prenose po jedinici merenja (danas obično mereno u GBps).

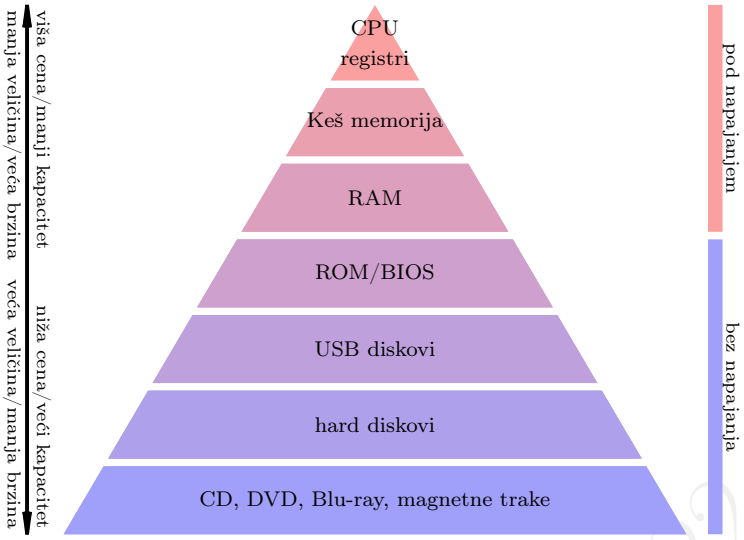
U savremenim računarskim sistemima, uz glavnu memoriju uspostavlja se čitava *hijerarhija memorija* koje služe da unaprede funkcionisanje sistema. Memorije neposredno vezane za procesor koje se koriste isključivo dok je računar uključen nazivaju se *unutrašnje memorije*, dok se memorije koje se koriste za skladištenje podataka u trenucima kada računar nije uključen nazivaju *spoljne memorije*. Procesor obično nema načina da direktno koristi podatke koji se nalaze u spoljnim memorijama (jer su one znatno sporije od unutrašnjih), već se pre upotrebe svi podaci prebacuju iz spoljnih u unutrašnju memoriju.

Memorijska hijerarhija predstavlja se piramidom. Od njenog vrha ka dnu opadaju kvalitet i brzina memorija, ali zato se smanjuje i cena, pa se kapacitet povećava.

Registri procesora predstavljaju najbržu memoriju jer se sve aritmetičke i logičke operacije izvode upravo nad podacima koji se nalaze u njima.

Keš (engl. *cache*) je mala količina brze memorije (nekoliko hiljada puta manjeg kapaciteta od glavne memorije; obično nekoliko megabajta) koja se postavlja između procesora i glavne memorije u cilju ubrzanja rada računara. Keš se uvodi jer su savremeni procesori postali znatno brži od glavnih memorija. Pre pristupa glavnoj memoriji procesor uvek prvo pristupa kešu. Ako traženi podatak tamo postoji, u pitanju je tzv. pogodak keša (engl. cache hit) i podatak se dostavlja procesoru. Ako se podatak ne nalazi u kešu, u pitanju je tzv. promašaj keša (engl. cache miss) i podatake se iz glavne memorije prenosi u keš zajedno sa određenim brojem podataka koji za njim slede (glavni faktor brzine glavne memorije je njeno kašnjenje i praktično je svejedno da li se prenosi jedan ili više podataka jer je vreme prenosa malog broja bajtova mnogo manje od vremena kašnjenja). Motivacija ovog pristupa je u tome što programi često pravilno pristupaju podacima (obično redom kojim su podaci smešteni u memoriji), pa je velika verovatnoća da će se naredni traženi podaci i instrukcije naći u keš-memoriji.

Glavna memorija čuva sve podatke i programe koje procesor izvršava. Mali deo glavne memorije čini ROM (engl. read only memory) – nepromenljiva memorija koja sadrži osnovne programe koji služe za kontrolu određenih komponenata računara (na primer, osnovni ulazno-izlazni sistem BIOS). Znatno veći deo glavne memorije čini RAM – privremena promenljiva memorija sa slobodnim pristupom. Terminološki, podela glavne memorije na ROM i RAM nije najpogodnija jer ove vrste memorije nisu suštinski različite — nepromenljivi



Slika 1.10: Memorijska hijerarhija

deo (ROM) je takođe memorija sa slobodnim pristupom (RAM). Da bi RAM memorija bila što brža, izrađuje se uvek od poluprovodničkih (elektronskih) elemenata. Danas se uglavnom realizuje kao sinhrona dinamička memorija (SDRAM). To znači da se prenos podataka između procesora i memorije vrši u intervalima određenim otkucajima sistemskog sata (često se u jednom otkucaju izvrši nekoliko prenosa). Dinamička memorija je znatno jeftinija i jednostavnija, ali zato sporija od statičke memorije od koje se obično gradi keš.

Spoljne memorije čuvaju podatke trajno – i kada računar ostane bez električnog napajanja. Kao centralna spoljna skladišta podataka uglavnom se koriste *hard diskovi* (engl. *hard disk*) koji čuvaju podatke korišćenjem magnetne tehnologije, a u novije vreme se sve više koriste i *SSD uređaji* (engl. *solid state drive*) koji čuvaju podatke korišćenjem elektronskih tzv. fleš memorija (engl. flash memory). Kao prenosne spoljne memorije koriste se uglavnom *USB fleš-memorije* (izrađene u sličnoj tehnologiji kao i SSD) i *optički diskovi* (CD, DVD, Blu-ray).

Ulazni uređaji. Osnovni ulazni uređaji današnjih računara su *tastature* i *miševi*. Prenosni računari imaju ugrađenu tastaturu, a umesto miša može se koristiti tzv. *tačped* (engl. *touchpad*). Tastature i miševi se sa računarom povezuju ili kablom (preko PS/2 ili USB priključaka) ili bežično (najčešće korišćenjem BlueTooth veze). Ovo su uglavnom standardizovani uređaji i nema velikih razlika među njima. *Skeneri* sliku sa papira prenose u računar. Princip rada je sličan digitalnom fotografisanju, ali prilagođen slikanju papira.

Izlazni uređaji. Osnovni izlazni uređaji savremenih računara su monitori. Danas dominiraju monitori tankog i ravnog ekrana (engl. flat panel display), zasnovani obično na tehnologiji tečnih kristala (engl. liquid crystal display, LCD) koji su osvetljeni pozadinskim LED osvetljenjem. Ipak, još uvek su ponegde u upotrebi i monitori sa katodnom cevi (engl. cathode ray tube, CRT). Grafički kontrolori koji služe za kontrolu slike koja se prikazuje na monitoru ili projektoru danas su obično integrisani na matičnoj ploči, a ponekad su i na istom čipu sa samim procesorom (engl. Accelerated Processing Unit, APU).

Što se tehnologije štampe tiče, danas su najzastupljeniji laserski štampači i inkdžet štampači (engl. inkjet). Laserski štampači su češće crno-beli, dok su ink-džet štampači obično u boji. Sve su dostupniji i 3D štampači.

1.5 Softver savremenih računara

Softver čine računarski programi i prateći podaci koji određuju izračunavanja koje vrši računar. Na prvim računarima moglo je da se programira samo na *mašinski zavisnim programskim jezicima* — na jezicima specifičnim za konkretnu mašinu na kojoj program treba da se izvršava. Polovinom 1950-ih nastali su prvi *jezici višeg nivoa* i oni su drastično olakšali programiranje. Danas se programi obično pišu u *višim programskim jezicima* a zatim prevode na *mašinski jezik* — jezik razumljiv računaru. Bez obzira na to kako je nastao, da bi mogao da se izvrši na računaru, program mora da budu smešten u memoriju u obliku binarno zapisanih podataka (tj. u obliku niza nula i jedinica) koji opisuju instrukcije koje su neposredno podržane arhitekturom računara. U ovom poglavlju biće prikazani osnovni principa rada računara kroz nekoliko jednostavnih primera programa.

1.5.1 Primeri opisa izračunavanja

Program specifikuje koje operacije treba izvršiti da bi se rešio neki zadatak. Principi rada programa mogu se ilustrovati na primeru nekoliko jednostavnih izračunavanja i instrukcija koje ih opisuju. Ovi opisi izračunavanja dati su u vidu prirodno-jezičkog opisa ali direktno odgovaraju i programima na višim programskim jezicima.

Kao prvi primer, razmotrimo izračunavanje vrednosti $2x + 3$ za datu vrednost x. U programiranju (slično kao i u matematici) podaci se predstavljaju *promenljivama*. Međutim, promenljive u programiranju (za razliku od matematike) vremenom mogu da menjaju svoju vrednost (tada kažemo da im se *dodeljuje nova vrednost*). U programiranju, svakoj promenljivoj pridruženo je (jedno, fiksirano) mesto u memoriji i tokom izvršavanja programa promenljiva može da menja svoju vrednost, tj. sadržaj dodeljenog memorijskog prostora. Ako je promenljiva čija je vrednost ulazni parametar označena sa x, a promenljiva čija je vrednost rezultat izračunavanja označena sa y, onda se pomenuto izračunavanje može opisati sledećim jednostavnim opisom.

```
y := 2*x + 3
```

Simbol `*` označava množenje, `+` sabiranje, a `:=` označava da se promenljivoj sa njene leve strane dodeljuje vrednost izraza sa desne strane.

Kao naredni primer, razmotrimo određivanje većeg od dva data broja. Računari (tj. njihovi procesori) obično imaju instrukcije za poređenje brojeva, ali određivanje vrednosti većeg broja zahteva nekoliko koraka. Pretpostavimo da promenljive x i y sadrže dve brojevne vrednosti, a da promenljiva m treba da dobije vrednost veće od njih. Ovo izračunavanje može da se izrazi sledećim opisom.

```
ako je x >= y onda
    m := x
inače
    m := y
```

Kao malo komplikovaniji primer razmotrimo stepenovanje. Procesori skoro uvek podržavaju instrukcije kojima se izračunava zbir i proizvod dva cela broja, ali stepenovanje obično nije podržano kao elementarna operacija. Složenije operacije se mogu ostvariti korišćenjem jednostavnijih. Na primer, n-ti stepen broja x (tj. vrednost x^n) moguće je izračunati uzastopnom primenom množenja: ako se krene od broja 1 i n puta sa pomnoži brojem x, rezultat će biti x^n. Da bi moglo da se osigura da će množenje biti izvršeno tačno n puta, koristi se brojačka promenljiva i koja na početku dobija vrednost 0, a zatim se, prilikom svakog množenja, uvećava sve dok ne dostigne vrednost n. Ovaj postupak možemo predstaviti sledećim opisom.

```
s := 1, i := 0
dok je i < n radi sledeće:
    s := s·x, i := i+1
```

Kada se ovaj postupak primeni na vrednosti $x = 3$ i $n = 2$, izvodi se naredni niz koraka.

`s := 1,`	`i := 0,`	pošto je `i(=0)` manje od `n(=2)`, vrše se dalje operacije
`s := s·x = 1·3 = 3,`	`i := i+1 = 0+1 = 1,`	pošto je `i(=1)` manje od `n(=2)`, vrše se dalje operacije
`s := s·x = 3·3 = 9,`	`i := i+1 = 1+1 = 2,`	pošto `i(=2)` nije manje od `n(=2)`, ne vrše se dalje operacije.

1.5.2 Mašinski programi

Mašinski programi su neposredno vezani za procesor računara na kojem se koriste — procesor je konstruisan tako da može da izvršava određene elementarne naredbe. Ipak, razvoj najvećeg broja procesora usmeren je tako da se isti mašinski programi mogu koristiti na čitavim familijama procesora.

Primitivne instrukcije koje podržava procesor su veoma malobrojne i jednostavne (na primer, postoje samo instrukcije za sabiranje dva broja, konjunkcija bitova, instrukcija skoka i slično) i nije lako kompleksne i apstraktne algoritme izraziti korišćenjem tog uskog skupa elementarnih instrukcija. Ipak, svi zadaci koje računari izvršavaju svode se na ove primitivne instrukcije.

Asemblerski jezici. Asemblerski (ili simbolički) jezici su jezici koji su veoma bliski mašinskom jeziku računara, ali se, umesto korišćenja binarnog sadržaja za zapisivanje instrukcija koriste (mnemotehničke, lako pamtljive) simboličke oznake instrukcija (tj. programi se unose kao tekst). Ovim se, tehnički, olakšava unos programa i programiranje (programer ne mora da direktno manipuliše binarnim sadržajem), pri čemu su sve mane mašinski zavisnog programiranja i dalje prisutne. Kako bi ovako napisan program mogao da se izvršava, neophodno je izvršiti njegovo prevođenje na mašinski jezik (tj. zapisati instrukcije binarnom azbukom) i uneti na odgovarajuće mesto u memoriji. Ovo prevođenje je jednostavno i jednoznačno i vrše ga jezički procesori koji se nazivaju *asembleri*.

Sva izračunavanja u primerima iz poglavlja 1.5.1 su opisana neformalno, kao uputstva čoveku a ne računaru. Da bi se ovako opisana izračunavanja mogla sprovesti na nekom računaru Fon Nojmanove arhitekture neophodno je opisati ih preciznije. Svaka elementarna operacija koju procesor može da izvrši u okviru programa zadaje se *procesorskom instrukcijom* — svaka instrukcija instruiše procesor da izvrši određenu operaciju. Svaki procesor podržava unapred fiksiran, konačan *skup instrukcija* (engl. *instruction set*). Svaki program računara predstavljen je nizom instrukcija i skladišti se u memoriji računara. Naravno, računari se razlikuju (na primer, po tome koliko registara u procesoru imaju, koje instrukcije može da izvrši njihova aritmetičko-logička jedinica, koliko memorije postoji na računaru, itd). Međutim, da bi se objasnili osnovni principi rada računara nije neophodno razmatrati neki konkretan računar, već se može razmatrati neki hipotetički računar. Pretpostavimo da procesor sadrži tri registra označena sa `ax`, `bx` i `cx` i još nekoliko izdvojenih bitova (tzv. zastavica). Dalje, pretpostavimo da procesor može da izvršava naredne *aritmetičke instrukcije* (zapisane ovde u asemblerskom obliku):

- Instrukcija `add ax, bx` označava operaciju sabiranja vrednosti brojeva koji se nalaze u registrima `ax` i `bx`, pri čemu se rezultat sabiranja smešta u registar `ax`. Operacija `add` može se primeniti na bilo koja dva registra.
- Instrukcija `mul ax, bx` označava operaciju množenja vrednosti brojeva koji se nalaze u registrima `ax` i `bx`, pri čemu se rezultat množenja smešta u registar `ax`. Operacija `mul` može se primeniti na bilo koja dva registra.
- Instrukcija `cmp ax, bx` označava operaciju poređenja vrednosti brojeva koji se nalaze u registrima `ax` i `bx` i rezultat pamti postavljanjem zastavice u procesoru. Operacija `cmp` se može primeniti na bilo koja dva registra.

Program računara je niz instrukcija koje se obično izvršavaju redom, jedna za drugom. Međutim, pošto se javlja potreba da se neke instrukcije ponove veći broj puta ili da se određene instrukcije preskoče, uvode se *instrukcije skoka*. Da bi se moglo specifikovati na koju instrukciju se vrši skok, uvode se *labele* – označena mesta u programu. Pretpostavimo da naš procesor može da izvršava sledeće dve vrste skokova (bezuslovne i uslovne):

- Instrukcija `jmp label`, gde je `label` neka labela u programu, označava bezuslovni skok koji uzrokuje nastavak izvršavanja programa od mesta u programu označenog navedenom labelom.
- Uslovni skokovi prouzrokuju nastavak izvršavanja programa od instrukcije označene navedenom labelom, ali samo ako je neki uslov ispunjen. Ukoliko uslov nije ispunjen, izvršava se naredna instrukcija. U nastavku će se razmatrati samo instrukcija `jge label`, koja uzrokuje uslovni skok na mesto označeno labelom `label` ukoliko je vrednost prethodnog poređenja brojeva bila *veće ili jednako*.

Tokom izvršavanja programa, podaci se nalaze u memoriji i u registrima procesora. S obzirom na to da procesor sve operacije može da izvrši isključivo nad podacima koji se nalaze u njegovim registrima, svaki procesor podržava i *instrukcije prenosa podataka* između memorije i registara procesora (kao i između samih registara). Pretpostavimo da naš procesor podržava sledeću instrukciju ove vrste.

- Instrukcija `mov` označava operaciju prenosa podataka i ima dva parametra — prvi određuje gde se podaci prenose, a drugi koji određuje koji se podaci prenose. Parametar može biti ime registra (što označava da se pristupa podacima u određenom registru), broj u zagradama (što označava da se pristupa podacima u memoriji i to na adresi određenoj brojem u zagradama) ili samo broj (što označava da je podatak baš taj navedeni broj). Na primer, instrukcija `mov ax bx` označava da se sadržaj registra `bx` prepisuje u registar `ax`, instrukcija `mov ax, [10]` označava da se sadržaj iz memorije sa adrese `10` prepisuje u registar `ax`, instrukcija `mov ax, 1` označava da se u registar ax upisuje vrednost `1`, dok instrukcija označava `mov [10], ax` da se sadržaj registra `ax` upisuje u memoriju na adresu `10`.

Sa ovakvim procesorom na raspolaganju, izračunavanje vrednosti $2x + 3$ može se ostvariti na sledeći način. Pretpostavimo da se ulazni podatak (broj x) nalazi u glavnoj memoriji i to na adresi `10`, a da rezultat y treba smestiti na adresu `11` (ovo su sasvim proizvoljno odabrane adrese). Izračunavanje se onda može opisati sledećim programom (nizom instrukcija).

```
mov ax, [10]
mov bx, 2
mul ax, bx
mov bx, 3
add ax, bx
mov [11], ax
```

Instrukcija `mov ax, [10]` prepisuje vrednost promenljive x (iz memorije sa adrese `10`) u registar `ax`. Instrukcija `mov bx, 2` upisuje vrednost `2` u registar `bx`. Nakon instrukcije `mul ax, bx` vrši se množenje i registar `ax` sadrži vrednost $2x$. Instrukcija `mov bx, 3` upisuje vrednost `3` u registar `bx`, nakon instrukcije `add ax, bx` se vrši sabiranje i u registru `ax` se nalazi tražena vrednost $2x + 3$. Na kraju se ta vrednost instrukcijom `mov [11], ax` upisuje u memoriju na adresu `11`.

Određivanje većeg od dva broja može se ostvariti na sledeći način. Pretpostavimo da se ulazni podaci nalaze u glavnoj memoriji i to broj x na adresi `10`, broj y na adresi `11`, dok rezultat m treba smestiti na adresu `12`. Program (niz instrukcija) kojima može da se odredi maksimum je sledeći:

```
mov ax, [10]
mov bx, [11]
cmp ax, bx
jge vecix
mov [12], bx
jmp kraj
vecix:
mov[12], ax
kraj:
```

Nakon prenosa vrednosti oba broja u registre procesora (instrukcijama `mov ax, [10]` i `mov bx, [11]`), vrši se njihovo poređenje (instrukcija `cmp ax, bx`). Ukoliko je broj x veći od ili jednak broju y prelazi se na mesto označeno labelom `vecix` (instrukcijom `jge vecix`) i na mesto rezultata upisuje se vrednost promenljive x (instrukcijom `mov[12], ax`). Ukoliko uslov skoka `jge` nije ispunjen (ako x nije veće ili jednako y), na mesto rezultata upisuje se vrednost promenljive y (instrukcijom `mov [12], bx`) i bezuslovno se skače na kraj programa (instrukcijom `jmp kraj`) (da bi se preskočilo izvršavanje instrukcije koja na mesto rezultata upisuje vrednost promenljive x).

Izračunavanje stepena može se ostvariti na sledeći način. Pretpostavimo da se ulazni podaci nalaze u glavnoj memoriji i to broj x na adresi `10`, a broj n na adresi `11`, i da konačan rezultat treba da bude smešten u memoriju i to na adresu `12`. Pretpostavimo da će pomoćne promenljive s i i koje se koriste u postupku biti smeštene sve vreme u procesoru, i to promenljiva s u registru `ax`, a promenljiva i u registru `bx`. Pošto postoji još samo jedan registar (`cx`), u njega će naizmenično biti smeštane vrednosti promenljivih n i x, kao i konstanta 1 koja se sabira sa promenljivom i. Niz instrukcija kojim opisani hipotetički računar može da izračuna stepen je sledeći:

```
  mov ax, 1
  mov bx, 0
petlja:
  mov cx, [11]
  cmp bx, cx
  jge kraj
  mov cx, [10]
  mul ax, cx
  mov cx, 1
  add bx, cx
  jmp petlja
kraj:
  mov [12], ax
```

Ilustrujmo izvršavanje ovog programa na izračunavanju vrednosti 3^2. Inicijalna konfiguracija je takva da se na adresi `10` u memoriji nalazi vrednost $x = 3$, na adresi `11` vrednost $n = 2$. Početna konfiguracija (tj. vrednosti memorijskih lokacija i registara) može da se predstavi na sledeći način:

```
10: 3    ax: ?
11: 2    bx: ?
12: ?    cx: ?
```

Nakon izvršavanja prve dve instrukcije (`mov ax, 1` i `mov bx, 0`), postavlja se vrednost registara `ax` i `bx` i prelazi se u sledeću konfiguraciju:

```
10: 3    ax: 1
11: 2    bx: 0
12: ?    cx: ?
```

Sledeća instrukcija (`mov cx, [11]`) kopira vrednost `2` sa adrese `11` u registar `cx`:

```
10: 3    ax: 1
11: 2    bx: 0
12: ?    cx: 2
```

Vrši se poređenje sa registrom `bx` (`cmp bx, cx`) i kako uslov skoka (`jge kraj`) nije ispunjen (vrednost `0` u `bx` nije veća ili jednaka od vrednosti `2` u `cx`), nastavlja se dalje. Nakon kopiranja vrednosti `3` sa adrese `10` u registar `cx` (instrukcijom `mov cx, [10]`), vrši se množenje vrednosti u registrima `ax` i `cx` (instrukcijom `mul ax, cx`) i dolazi se u sledeću konfiguraciju:

```
10: 3    ax: 3
11: 2    bx: 0
12: ?    cx: 3
```

Nakon toga, u `cx` se upisuje `1` (instrukcijom `mov cx, 1`) i vrši se sabiranje vrednosti registara `bx` i `cx` (instrukcijom `add bx, cx`) čime se vrednost u registru `bx` uvećava za `1`.

```
10: 3    ax: 3
11: 2    bx: 1
12: ?    cx: 1
```

Bezuslovni skok (`jmp petlja`) ponovo vraća kontrolu na početak petlje, nakon čega se u `cx` opet prepisuje vrednost `2` sa adrese `11` (`mov cx, [11]`). Vrši se poređenje sa registrom `bx` (`cmp bx, cx`) i kako uslov skoka (`jge kraj`) nije ispunjen (vrednost `1` u `bx` nije veća ili jednaka vrednosti `2` u `cx`), nastavlja se dalje. Nakon još jednog množenja i sabiranja dolazi se do konfiguracije:

```
10: 3    ax: 9
11: 2    bx: 2
12: ?    cx: 1
```

Bezuslovni skok ponovo vraća kontrolu na početak petlje, nakon čega se u `cx` opet prepisuje vrednost `2` sa adrese `11`. Vrši se poređenje sa registrom `bx`, no, ovaj put je uslov skoka ispunjen (vrednost `2` u `bx` je veća ili jednaka vrednosti `2` u `cx`) i skače se na mesto označeno labelom `kraj`, gde se poslednjom instrukcijom (`mov [12], ax`) konačna vrednost iz registra `ax` kopira u memoriju na dogovorenu adresu `12`, čime se stiže u završnu konfiguraciju:

```
10: 3     ax: 9
11: 2     bx: 2
12: 9     cx: 1
```

Mašinski jezik. Fon Nojmanova arhitektura podrazumeva da se i sam program (niz instrukcija) nalazi u glavnoj memoriji prilikom njegovog izvršavanja. Potrebno je svaki program (poput tri navedena) predstaviti nizom nula i jedinica, na način „razumljiv“ procesoru — na mašinskim jeziku. Na primer, moguće je da su binarni kodovi za instrukcije uvedeni na sledeći način:

```
mov  001
add  010
mul  011
cmp  100
jge  101
jmp  110
```

Takođe, pošto neke instrukcije primaju podatke različite vrste (neposredno navedeni brojevi, registri, apsolutne memorijske adrese), uvedeni su posebni kodovi za svaki od različitih vidova adresiranja. Na primer:

neposredno	`00`
registarsko	`01`
apsolutno	`10`

Pretpostavimo da registar `ax` ima oznaku `00`, registar `bx` ima oznaku `01`, a registar `cx` oznaku `10`. Pretpostavimo i da su sve adrese osmobitne. Pod navedenim pretpostavkama, instrukcija `mov [10], ax` se, u ovom hipotetičkom mašinskom jeziku, može kodirati kao `001 10 01 00010000 00`. Kôd `001` dolazi od instrukcije `mov`, zatim slede `10` i `01` koji ukazuju da prvi argument predstavlja memorijsku adresu, a drugi oznaku registra, za čim sledi memorijska adresa $(10)_{16}$ binarno kodirana sa `00010000` i na kraju oznaka `00` registra `ax`. Na sličan način, celokupan prikazani mašinski kôd navedenog asemblerskog programa koji izračunava $2x + 3$ je moguće binarno kodirati kao:

```
001 01 10 00 00010000   // mov ax, [10]
001 01 00 01 00000010   // mov bx, 2
011 00 01               // mul ax, bx
001 01 00 01 00000011   // mov bx, 3
010 00 01               // add ax, bx
001 10 01 00010001 00   // mov [11], ax
```

Između prikazanog asemblerskog i mašinskog programa postoji veoma direktna i jednoznačna korespondencija (u oba smera) tj. na osnovu datog mašinskog koda moguće je jednoznačno rekonstruisati asemblerski kôd.

Specifični hardver koji čini kontrolnu jedinicu procesora dekodira jednu po jednu instrukciju i izvršava akciju zadatu tom instrukcijom. Kod realnih procesora, broj instrukcija i načini adresiranja su mnogo bogatiji a prilikom pisanja programa potrebno je uzeti u obzir mnoge aspekte na koje se u navedenim jednostavnim primerima nije obraćala pažnja. Ipak, mašinske i asemblerske instrukcije stvarnih procesora veoma su slične navedenim hipotetičkim instrukcijama.

1.5.3 Klasifikacija savremenog softvera

Računarski programi veoma su složeni. Hardver računara sačinjen je od elektronskih kola koja mogu da izvrše samo elementarne operacije i, da bi računar mogao da obavi i najjednostavniji zadatak zanimljiv korisniku, neophodno je da se taj zadatak razloži na mnoštvo elementarnih operacija. Napredak računara ne bi bio moguć ako bi programeri morali svaki program da opisuju i razlažu do krajnjeg nivoa elementarnih instrukcija. Zato je poželjno da programeri naredbe računaru mogu zadavati na što apstraktnijem nivou. Računarski sistemi i softver se grade slojevito i svaki naredni sloj oslanja se na funkcionalnost koju mu nudi sloj ispod njega. U skladu sa tim, softver savremenih računara se obično deli na *sistemski* i *aplikativni*. Osnovni zadatak sistemskog softvera je da posreduje između hardvera i aplikativnog softvera koji krajnji korisnici koriste. Granica između sistemskog i aplikativnog softvera nije kruta i postoje programi za koje se može smatrati da pripadaju obema grupama (na primer, editori teksta).

Sistemski softver. Sistemski softver je softver čija je uloga da kontroliše hardver i pruža usluge aplikativnom softveru. Najznačajniji skup sistemskog softvera, danas prisutan na skoro svim računarima, čini *operativni sistem* (OS). Pored OS, sistemski softver sačinjavaju i različiti *uslužni programi*: editori teksta, alat za programiranje (prevodioci, dibageri, profajleri, integrisana okruženja) i slično.

Korisnici OS često identifikuju sa izgledom ekrana tj. sa programom koji koriste da bi pokrenuli svoje aplikacije i organizovali dokumente. Međutim, ovaj deo sistema koji se naziva *korisnički interfejs* (engl. *user interface – UI*) ili *školjka* (engl. *shell*) samo je tanak sloj na vrhu operativnog sistema i OS je mnogo više od onoga što krajnji korisnici vide. Najveći i najznačajni deo OS naziva se *jezgro* (engl. *kernel*). Osim što kontroliše i apstrahuje hardver, operativni sistem tj. njegovo jezgro sinhronizuje rad više programa, raspoređuje procesorsko vreme i memoriju, brine o sistemu datoteka na spoljašnjim memorijama itd. Najznačanjni operativni sistemi danas su Microsoft Windows, sistemi zasnovani na Linux jezgru (na primer, Ubuntu, RedHat, Fedora, Suse) i Mac OS X.

OS upravlja svim resursima računara (procesorom, memorijom, periferijskim uređajima) i stavlja ih na raspolaganje aplikativnim programima. OS je u veoma tesnoj vezi sa hardverom računara i veliki deo zadataka se izrvršava uz direktnu podršku specijalizovanog hardvera namenjenog isključivo izvršavanju OS. Nekada se hardver i operativni sistem smatraju jedinstvenom celinom i umesto podele na hardver i softver razmatra se podela na sistem (hardver i OS) i na aplikativni softver.

Aplikativni softver. Aplikativni softver je softver koji krajnji korisnici računara direktno koriste u svojim svakodnevnim aktivnostima. To su pre svega pregledači Veba, zatim klijenti elektronske pošte, kancelarijski softver (programi za kucanje teksta, izradu slajd-prezentacija, tabelarna izračunavanja), video igre, multimedijalni softver (programi za reprodukciju i obradu slika, zvuka i video-sadržaja) itd.

Programer ne bi trebalo da misli o konkretnim detaljima hardvera, tj. poželjno je da postoji određena *apstrakcija hardvera*. Na primer, mnogo je pogodnije ako programer umesto da mora da kaže „Neka se zavrti ploča diska, neka se glava pozicionira na određenu poziciju, neka se zatim tu upiše određeni bajt itd.“ može da kaže „Neka se u datu datoteku na disku upiše određeni tekst“. OS je taj koji se brine o svim detaljima, dok se programer (tačnije, aplikacije koje on isprogramira), kada god mu je potrebno obraća sistemu da mu tu uslugu pruži. Konkretni detalji hardvera poznati su u okviru operativnog sistema i komande koje programer zadaje izvršavaju se uzimajući u obzir ove specifičnosti. Operativni sistem, dakle, programeru pruža skup funkcija koje on može da koristi da bi postigao željenu funkcionalnost hardvera, sakrivajući pritom konkretne hardverske detalje. Ovaj skup funkcija naziva se *programski interfejs za pisanje aplikacija*[24] (engl. *Aplication Programming Interface, API*). Funkcije se nazivaju i *sistemski pozivi* (jer se OS poziva da izvrši određeni zadatak). Programer nema mogućnost direktnog pristupa hardveru i jedini način da se pristupi hardveru je preko sistemskih poziva. Ovim se osigurava određena bezbednost celog sistema.

Postoji više nivoa na kojima se može realizovati neka funkcionalnost. Programer aplikacije je na vrhu hijerarhije i on može da koristi funkcionalnost koju mu pruža programski jezik koji koristi i *biblioteke* tog jezika. Izvršivi programi često koriste funkcionalnost specijalne *rantajm biblioteke* (engl. *runtime library*) koja koristi funkcionalnost operativnog sistema (preko sistemskih poziva), a zatim operativni sistem koristi funkcionalnost samog hardvera.

Pitanja za vežbu

Pitanje 1.1. *Nabrojati osnovne periode u razvoju računara i navesti njihove osnovne karakteristike i predstavnike.*

Pitanje 1.2. *Ko je i u kom veku konstruisao prvu mehaničku spravu na kojoj je bilo moguće sabirati prirodne brojeve, a ko je i u kom veku konstruisao prvu mehaničku spravu na kojoj je bilo moguće sabirati i množiti prirodne brojeve?*

Pitanje 1.3. *Kojoj spravi koja se koristi u današnjem svetu najviše odgovaraju Paskalove i Lajbnicove sprave?*

Pitanje 1.4. *Kakva je veza između tkačkih razboja i računara s početka XIX veka?*

Pitanje 1.5. *Koji je značaj Čarlsa Bebidža za razvoj računarstva i programiranja? U kom veku je on dizajnirao svoje računske mašine? Kako se one zovu i koja od njih je trebalo da bude programabilna? Ko se smatra prvim programerom?*

Pitanje 1.6. *Na koji način je Herman Holerit doprineo izvršavanju popisa stanovnika u SAD 1890? Kako su bili čuvani podaci sa tog popisa? Koja čuvena kompanija je nastala iz kompanije koju je Holerit osnovao?*

Pitanje 1.7. *Kada su nastali prvi elektronski računari? Nabrojati nekoliko najznačajnijih.*

Pitanje 1.8. *Na koji način je programiran računar ENIAC, a na koji računar EDVAC?*

Pitanje 1.9. *Koje su osnovne komponente računara Fon Nojmanove arhitekture? Šta se skladišti u memoriju računara Fon Nojmanove arhitekture? Gde se vrši obrada podataka u okviru računara Fon Nojmanove arhitekture? Od kada su računari zasnovani na Fon Nojmanovoj arhitekturi?*

Pitanje 1.10. *Šta su to računari sa skladištenim programom? Šta je to hardver a šta softver?*

Pitanje 1.11. *Šta su procesorske instrukcije? Navesti nekoliko primera.*

[24] Ovaj termin se ne koristi samo u okviru operativnih sistema, već i u širem kontekstu, da označi skup funkcija kroz koji jedan programski sistem koristi drugi programski sistem.

Pitanje 1.12. *Koji su uobičajeni delovi procesora? Da li se u okviru samog procesora nalazi određena količina memorije za smeštanje podataka? Kako se ona naziva?*

Pitanje 1.13. *Ukratko opisati osnovne elektronske komponente svake generacije računara savremenih elektronskih računara? Šta su bile osnovne elektronske komponente prve generacije elektronskih računara? Od koje generacije računara se koriste mikroprocesori? Koji tipovi računara se koriste u okviru III generacije?*

Pitanje 1.14. *U kojoj deceniji dolazi do pojave računara za kućnu upotrebu? Koji je najprodavaniji model kompanije Commodore? Da li je IBM proizvodio računare za kućnu upotrebu? Koji komercijalni kućni računar prvi uvodi grafički korisnički interfejs i miša?*

Pitanje 1.15. *Koja serija Intelovih procesora je bila dominantna u PC računarima 1980-ih i 1990-ih godina?*

Pitanje 1.16. *Šta je to tehnološka konvergencija? Šta su to tableti, a šta „pametni telefoni“?*

Pitanje 1.17. *Koje su osnovne komponente savremenog računara? Šta je memorijska hijerarhija? Zašto se uvodi keš-memorija? Koje su danas najkorišćenije spoljne memorije?*

Pitanje 1.18. *U koju grupu jezika spadaju mašinski jezici i asemblerski jezici?*

Pitanje 1.19. *Da li je kôd na nekom mašinskom jeziku prenosiv sa jednog na sve druge računare? Da li asembler zavisi od mašine na kojoj se koristi?*

Pitanje 1.20. *Ukoliko je raspoloživ asemblerski kôd nekog programa, da li je moguće jednoznačno konstruisati odgovarajući mašinski kôd? Ukoliko je raspoloživ mašinski kôd nekog programa, da li je moguće jednoznačno konstruisati odgovarajući asemblerski kôd?*

Zadatak 1.1. *Na opisanom asemblerskom jeziku opisati izračunavanje vrednosti izraza* `x := x*y + y + 3`. *Generisati i mašinski kôd za napisani program.*

Zadatak 1.2. *Na opisanom asemblerskom jeziku opisati izračunavanje:*

```
ako je (x < 0)
   y := 3*x;
inace
   x := 3*y;
```

Zadatak 1.3. *Na opisanom asemblerskom jeziku opisati izračunavanje:*

```
dok je (x <= 0) radi
   x := x + 2*y + 3;
```

Zadatak 1.4. *Na opisanom asemblerskom jeziku opisati izračunavanje kojim se izračunava* $[\sqrt{x}]$, *pri čemu se* x *nalazi na adresi 100, a rezultat smešta na adresu 200.* √

Pitanje 1.21. *Koji su osnovni razlozi slojevite organizacije softvera? Šta je sistemski, a šta aplikativni softver?*

Pitanje 1.22. *Koji su osnovni zadaci operativnog sistema? Šta su sistemski pozivi? Koji su operativni sistemi danas najkorišćeniji?*

Glava 2

Reprezentacija podataka u računarima

Današnji računari su *digitalni*. To znači da su svi podaci koji su u njima zapisani – zapisani kao nizovi brojeva. Zapisati tekstove, slike, zvuk i filmove i vidu brojeva zahteva pogodnu reprezentaciju koja ponekad znači i gubitak dela polaznih informacija. Kada su podaci predstavljeni u vidu brojeva, te brojeve potrebno je zapisati u računarima. Dekadni brojevni sistem koji ljudi koriste u svakodnevnom životu nije pogodan za zapis brojeva u računarima jer zahteva azbuku od 10 različitih simbola (cifara). Bilo da se radi o elektronskim, magnetnim ili optičkim komponentama, tehnologija izrade računara i medijuma za zapis podataka koristi elemente koji imaju dva diskretna stanja, što za zapis podataka daje azbuku od samo dva različita simbola. Tako, na primer, ukoliko između dve tačke postoji napon viši od određenog praga, onda se smatra da tom paru tačaka odgovara vrednost 1, a inače mu odgovara vrednost 0. Takođe, polje hard diska može biti ili namagnetisano što odgovara vrednosti 1 ili razmagnetisano što odgovara vrednosti 0. Slično, laserski zrak na površini kompakt diska „buši rupice“ kojim je određen zapis podataka pa polje koje nije izbušeno predstavlja vrednost 0, a ono koje jeste izbušesno predstavlja vrednost 1. U nastavku će biti pokazano da je azbuka od samo dva simbola dovoljna za zapisivanje svih vrsta brojeva, pa samim tim i za zapisivanje svih vrsta digitalnih podataka.

2.1 Analogni i digitalni podaci i digitalni računari

Kontinualna priroda signala. Većina podataka koje računari koriste nastaje zapisivanjem prirodnih signala. Najznačajniji primeri signala su zvuk i slika, ali se pod signalima podrazumevaju i ultrazvučni signali, EKG signali, zračenja različite vrste itd.

Signali koji nas okružuju u prirodi u većini slučajeva se prirodno mogu predstaviti neprekidnim funkcijama. Na primer, zvučni signal predstavlja promenu pritiska vazduha u zadatoj tački i to kao neprekidnu funkciju vremena. Slika se može opisati intenzitetom svetlosti određene boje (tj. određene talasne dužine) u datom vremenskom trenutku i to kao neprekidna funkcija prostora.

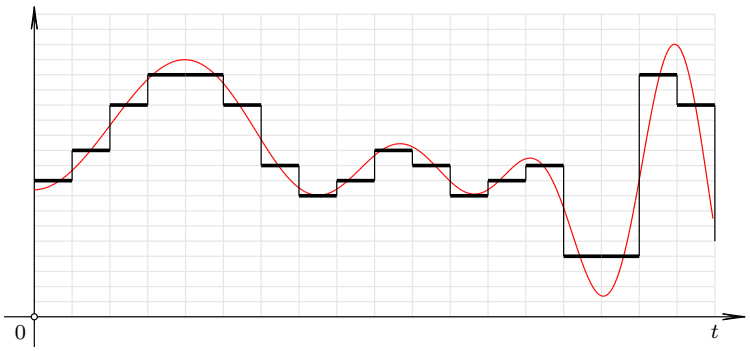
Analogni zapis. Osnovna tehnika koja se primenjuje kod analognog zapisa signala je da se kontinualne promene signala koji se zapisuje opišu kontinualnim promenama određenog svojstva medijuma na kojem se signal zapisuje. Tako, na primer, promene pritiska vazduha koji predstavlja zvučni signal direktno odgovaraju promenama nivoa namagnetisanja na magnetnoj traci na kojoj se zvuk analogno zapisuje. Količina boje na papiru direktno odgovara intenzitetu svetlosti u vremenskom trenutku kada je fotografija bila snimljena. Dakle, analogni zapis uspostavlja *analogiju* između signala koji je zapisan i određenog svojstva medijuma na kome je signal zapisan.

Osnovna prednost analogne tehnologije je da je ona obično veoma jednostavna ukoliko se zadovoljimo relativno niskim kvalitetom (još su drevni narodi mogli da naprave nekakav zapis zvuka uz pomoć jednostavne igle prikačene na trepereću membranu).

Osnovni problem analogne tehnologije je što je izrazito teško na medijumu napraviti veran zapis signala koji se zapisuje i izrazito je teško napraviti dva identična zapisa istog signala. Takođe, problem predstavlja i inherentna nestalnost medijuma, njegova promenljivost tokom vremena i podložnost spoljašnjim uticajima. S obzirom na to da varijacije medijuma direktno dovode do varijacije zapisanog signala, vremenom neizbežno dolazi do pada kvaliteta analogno zapisanog signala. Obrada analogno zapisanih signala je obično veoma komplikovana i za svaku vrstu obrade signala, potrebno je da postoji uređaj koji je specijalizovan za tu vrste obrade.

Digitalni zapis. Osnovna tehnika koja se koristi kod digitalnog zapisa podataka je da se vrednost signala izmeri u određenim vremenskim trenucima ili određenim tačkama prostora i da se onda na medijumu zapišu izmerene vrednosti. Ovim je svaki digitalno zapisani signal predstavljen nizom brojeva koji se nazivaju *odbirci* ili

semplovi (engl. *sample*). Svaki od brojeva predstavlja vrednost signala u jednoj tački diskretizovanog domena. S obzirom na to da izmerene vrednosti takođe pripadaju kontinualnoj skali, neophodno je izvršiti i diskretizaciju kodomena, odnosno dopustiti zapisivanje samo određenog broja nivoa različitih vrednosti.



Slika 2.1: Digitalizacija zvučnog signala

Digitalni zapis predstavlja diskretnu aproksimaciju polaznog signala. Važno pitanje je koliko često je potrebno vršiti merenje da bi se polazni kontinualni signal mogao verno rekonstruisati. Odgovor daje tvrđenje o odabiranju (tzv. Najkvist-Šenonova teorema), koje kaže da je signal dovoljno meriti dva puta češće od najviše frekvencije koja sa u njemu javlja. Na primer, pošto čovekovo uho čuje frekvencije do 20kHz, dovoljno je da frekvencija odabiranja (semplovanja) bude 40kHz. Dok je za analogne tehnologije za postizanje visokog kvaliteta zapisa potrebno imati medijume visokog kvaliteta, kvalitet reprodukcije digitalnog zapisa ne zavisi od toga kakav je kvalitet medija na kome su podaci zapisani, sve dok je medijum dovoljnog kvaliteta da se zapisani brojevi mogu razaznati. Dodatno, kvarljivost koja je inherentna za sve medije postaje nebitna. Na primer, papir vremenom žuti što uzrokuje pad kvaliteta analognih fotografija tokom vremena. Međutim, ukoliko bi papir sadržao zapis brojeva koji predstavljaju vrednosti boja u tačkama digitalno zapisane fotografije, činjenica da papir žuti ne bi predstavljala problem dok god se brojevi mogu razaznati.

Digitalni zapis omogućava kreiranje apsolutno identičnih kopija što dalje omogućava prenos podataka na daljinu. Na primer, ukoliko izvršimo fotokopiranje fotografije, napravljena fotokopija je daleko lošijeg kvaliteta od originala. Međutim, ukoliko umnožimo CD na kojem su zapisani brojevi koji čine zapis neke fotografije, kvalitet slike ostaje apsolutno isti. Ukoliko bi se dva CD-a pregledala pod mikroskopom, oni bi izgledali delimično različito, ali to ne predstavlja problem sve dok se brojevi koji su na njima zapisani mogu razaznati.

Obrada digitalno zapisanih podataka se svodi na matematičku manipulaciju brojevima i ne zahteva (za razliku od analognih podataka) korišćenje specijalizovanih mašina.

Osnovni problem implementacije digitalnog zapisa predstavlja činjenica da je neophodno imati veoma razvijenu tehnologiju da bi se uopšte stiglo do iole upotrebljivog zapisa. Na primer, izuzetno je komplikovano napraviti uređaj koji je u stanju da 40 hiljada puta izvrši merenje intenziteta zvuka. Jedna sekunda zvuka se predstavlja sa 40 hiljada brojeva, za čiji je zapis neophodna gotovo cela jedna sveska. Ovo je osnovni razlog zbog čega se digitalni zapis istorijski javio kasno. Kada se došlo do tehnološkog nivoa koji omogućava digitalni zapis, on je doneo mnoge prednosti u odnosu na analogni.

2.2 Zapis brojeva

Proces digitalizacije je proces reprezentovanja (raznovrsnih) podataka brojevima. Kako se svi podaci u računarima reprezentuju na taj način — brojevima, neophodno je precizno definisati zapisivanje različitih vrsta brojeva. Osnovu digitalnih računara, u skladu sa njihovom tehnološkom osnovom, predstavlja *binarni* brojevni sistem (sistem sa osnovom 2) . U računarstvu se koriste i *heksadekadni* brojevni sistem (sistem sa osnovom 16) a i, nešto ređe, *oktalni* brojevni sistem (sistem sa osnovom 8), zbog toga što ovi sistemi omogućavaju jednostavnu konverziju između njih i binarnog sistema. Svi ovi brojevni sistemi, kao i drugi o kojima će biti reči u nastavku teksta, su *pozicioni*. U pozicionom brojnom sistemu, udeo cifre u celokupnoj vrednosti zapisanog broja zavisi od njene pozicije.

Treba naglasiti da je zapis broja samo konvencija a da su brojevi koji se zapisuju apsolutni i ne zavise od konkretnog zapisa. Tako, na primer, zbir dva prirodna broja je uvek jedan isti prirodni broj, bez obzira na to u kom sistemu su ova tri broja zapisana.

S obzirom na to da je svaki zapis broja u računaru ograničen, ne mogu biti zapisani svi celi brojevi. Ipak, za cele brojeve zapisive u računaru se obično govori samo *celi brojevi*, dok su ispravnija imena *označeni celi brojevi*

(engl. *signed integers*) i *neoznačeni celi brojevi* (engl. *unsigned integers*), koji podrazumevaju konačan zapis. Ni svi realni brojevi (sa potencijalno beskonačnim decimalnim zapisom) ne mogu biti zapisani u računaru. Za zapis zapisivih realnih brojeva (koji su uvek racionalni) obično se koristi konvencija zapisa u pokretnom zarezu. Iako je jedino precizno ime za ove brojeve *brojevi u pokretnom zarezu* (engl. *floating point numbers*), često se koriste i imena *realni* ili *racionalni brojevi*. Zbog ograničenog zapisa brojeva, rezultati matematičkih operacija nad njima sprovedenih u računaru, neće uvek odgovarati rezultatima koji bi se dobili bez tih ograničenja (zbog takozvanih *prekoračenja*). Naglasimo još i da, za razliku od matematike gde se skup celih brojeva smatra podskupom skupa realnih brojeva, u računarstvu, zbog različitog načina zapisa, između zapisa ovih vrsta brojeva ne postoji direktna veza.

2.2.1 Neoznačeni brojevi

Pod *neoznačenim brojevima* podrazumeva se neoznačeni zapis nenegativnih celih brojeva i znak se izostavlja iz zapisa.

Određivanje broja na osnovu datog zapisa. Pretpostavimo da je dat pozicioni brojevni sistem sa osnovom b, gde je b prirodan broj veći od 1. Niz cifara $(a_n\, a_{n-1}\, \ldots\, a_1\, a_0)_b$ predstavlja zapis[1] broja u osnovi b, pri čemu za svaku cifru a_i važi $0 \leq a_i < b$.

Vrednost broja zapisanog u osnovi b definiše se na sledeći način:

$$(a_n\, a_{n-1}\, \ldots\, a_1\, a_0)_b = \sum_{i=0}^{n} a_i \cdot b^i = a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + \ldots a_1 \cdot b + a_0$$

Na primer:
$(101101)_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2 + 1 = 32 + 8 + 4 + 1 = 45$,
$(3245)_8 = 3 \cdot 8^3 + 2 \cdot 8^2 + 4 \cdot 8 + 5 = 3 \cdot 512 + 2 \cdot 64 + 4 \cdot 8 + 5 = 1536 + 128 + 32 + 5 = 1701$.
Navedena definicija daje i postupak za određivanje vrednosti datog zapisa:

x `:=` 0
`za svako` i `od` 0 `do` n
    x `:=` $x + a_i \cdot b^i$

ili, malo modifikovano:

x `:=` a_0
`za svako` i `od` 1 `do` n
    x `:=` $x + a_i \cdot b^i$

Za izračunavanje vrednosti nekog $(n+1)$-tocifrenog zapisa drugim navedenim postupkom potrebno je n sabiranja i $n + (n-1) + \ldots + 1 = \frac{n(n+1)}{2}$ množenja. Zaista, da bi se izračunalo $a_n \cdot b^n$ potrebno je n množenja, da bi se izračunalo $a_{n-1} \cdot b^{n-1}$ potrebno je $n-1$ množenja, itd. Međutim, ovo izračunavanje može da se izvrši i efikasnije. Ukoliko se za izračunavanje člana b^i iskoristi već izračunata vrednost b^{i-1}, broj množenja se može svesti na $2n$. Ovaj način izračunavanja primenjen je u sledećem postupku:

x `:=` a_0
B `:=` 1
`za svako` i `od` 1 `do` n
    B `:=` $B \cdot b$
    x `:=` $x + a_i \cdot B$

Još efikasniji postupak izračunavanja se može dobiti korišćenjem *Hornerove sheme*:

$$(a_n\, a_{n-1}\, \ldots\, a_1\, a_0)_b = (\ldots((a_n \cdot b + a_{n-1}) \cdot b + a_{n-2}) \ldots + a_1) \cdot b + a_0$$

Korišćenjem ove sheme, dolazi se do sledećeg postupka za određivanje vrednosti broja zapisanog u nekoj brojevnoj osnovi:

[1] Ako u zapisu broja nije navedena osnova, podrazumeva se da je osnova 10.

```
x := 0
za svako i od n unazad do 0
    x := x · b + a_i
```

Naredni primer ilustruje primenu Hornerovog postupka na zapis $(9876)_{10}$.

i		3	2	1	0
a_i		9	8	7	6
x	0	$0 \cdot 10 + 9 = 9$	$9 \cdot 10 + 8 = 98$	$98 \cdot 10 + 7 = 987$	$987 \cdot 10 + 6 = 9876$

Međurezultati dobijeni u ovom računu direktno odgovaraju prefiksima zapisa čija se vrednost određuje, a rezultat u poslednjoj koloni je traženi broj.

Navedeni postupak može se primeniti na proizvoljnu brojevnu osnovu. Sledeći primer ilustruje primenu postupka na zapis $(3245)_8$.

i		3	2	1	0
a_i		3	2	4	5
x	0	$0 \cdot 8 + 3 = 3$	$3 \cdot 8 + 2 = 26$	$26 \cdot 8 + 4 = 212$	$212 \cdot 8 + 5 = 1701$

Navedena tabela može se kraće zapisati na sledeći način:

	3	2	4	5
0	3	26	212	1701

Hornerov postupak je efikasniji u odnosu na početni postupak, jer je u svakom koraku dovoljno izvršiti samo jedno množenje i jedno sabiranje (ukupno $n + 1$ sabiranja i $n + 1$ množenja).

Određivanje zapisa datog broja. Za svaku cifru a_i u zapisu broja x u osnovi b važi da je $0 \leq a_i < b$. Dodatno, pri deljenju broja x osnovom b, ostatak je a_0 a celobrojni količnik je broj čiji je zapis $(a_n\, a_{n-1} \ldots a_1)_b$. Dakle, izračunavanjem celobrojnog količnika i ostatka pri deljenju sa b, određena je poslednja cifra broja x i broj koji se dobija uklanjanjem poslednje cifre iz zapisa. Ukoliko se isti postupak primeni na dobijeni količnik, dobija se postupak koji omogućava da se odrede sve cifre u zapisu broja x. Postupak se zaustavlja kada tekući količnik postane 0. Ako se izračunavanje ostatka pri deljenju označi sa `mod`, a celobrojnog količnika sa `div`, postupak kojim se određuje zapis broja x u datoj osnovi b se može formulisati na sledeći način:

```
i := 0
dok je x različito od 0
    a_i := x mod b
    x := x div b
    i := i + 1
```

Na primer, $1701 = (3245)_8$ jer je $1701 = 212 \cdot 8 + 5 = (26 \cdot 8 + 4) \cdot 8 + 5 = ((3 \cdot 8 + 2) \cdot 8 + 4) \cdot 8 + 5 = (((0 \cdot 8 + 3) \cdot 8 + 2) \cdot 8 + 4) \cdot 8 + 5$. Ovaj postupak se može prikazati i tabelom:

i	0	1	2	3	
x	1701	1701 div 8 = 212	212 div 8 = 26	26 div 8 = 3	3 div 8 = 0
a_i	1701	1701 mod 8 = 5	212 mod 8 = 4	26 mod 8 = 2	3 mod 8 = 3

Prethodna tabela može se kraće zapisati na sledeći način:

1701	212	26	3	0
5	4	2	3	

Druga vrsta tabele sadrži celobrojne količnike, a treća ostatke pri deljenju sa osnovom b, tj. tražene cifre. Zapis broja se formira tako što se dobijene cifre čitaju unatrag.

Ovaj algoritam i Hornerov algoritam su međusobno simetrični u smislu da se svi međurezultati poklapaju.

Direktno prevođenje između heksadekadnog i binarnog sistema.
Osnovni razlog korišćenja heksadekadnog sistema je mogućnost jednostavnog prevođenja brojeva između binarnog i heksadekadnog sistema. Pri tome, heksadekadni sistem omogućava da se binarni sadržaj memorije zapiše kompaktnije (uz korišćenje manjeg broja cifara). Prevođenje se može izvršiti tako što se grupišu četiri po četiri binarne cifre, krenuvši unazad, i svaka četvorka se zasebno prevede u odgovarajuću heksadekadnu cifru na osnovu sledeće tabele:

heksa	binarno	heksa	binarno	heksa	binarno	heksa	binarno
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Na primer, proizvoljni 32-bitni sadržaj može se zapisati korišćenjem osam heksadekadnih cifara:
$(1011\,0110\,0111\,1100\,0010\,1001\,1111\,0001)_2 = (B67C29F1)_{16}$

Zapisi fiksirane dužine U računarima se obično koristi fiksirani broj binarnih cifara (sačuvanih u pojedinačnim *bitovima*) za zapis svakog broja. Takve zapise označavamo sa $(\ldots)_b^n$, ako se koristi n cifara. Ukoliko je broj cifara potrebnih za zapis broja kraći od zadate dužine zapisa, onda se broj proširuje vodećim nulama. Na primer, $55 = (0011\,0111)_2^8$. Ograničavanjem broja cifara ograničava se i raspon brojeva koje je moguće zapisati (u binarnom sistemu) i to na raspon od 0 do $2^n - 1$. U sledećoj tabeli su dati rasponi za najčešće korišćene dužine zapisa:

broj bitova	raspon
8	od 0 do 255
16	od 0 do 65535
32	od 0 do 4294967295

2.2.2 Označeni brojevi

Označeni brojevi su celi brojevi čiji zapis uključuje i zapis znaka broja (+ ili −). S obzirom na to da savremeni računari koriste binarni brojevni sistem, biće razmatrani samo zapisi označenih brojeva u binarnom brojevnom sistemu. Postoji više načina zapisivanja označenih brojeva od kojih su najčešće u upotrebi *označena apsolutna vrednost* i *potpuni komplement*.

Označena apsolutna vrednost. Zapis broja se formira tako što se na prvu poziciju zapisa unapred fiksirane dužine n, upiše znak broja, a na preostalih $n-1$ pozicija upiše zapis apsolutne vrednosti broja. Pošto se za zapis koriste samo dva simbola (0 i 1), konvencija je da se znak + zapisuje simbolom 0, a znak − simbolom 1. Ovim se postiže da pozitivni brojevi imaju identičan zapis kao da su u pitanju neoznačeni brojevi. Na primer, $+100 = (0\ 1100100)_2^8$, $-100 = (1\ 1100100)_2^8$.

Osnovni problem zapisa u obliku označene apsolutne vrednosti je činjenica da se osnovne aritmetičke operacije teško izvode ukoliko su brojevi zapisani na ovaj način.

Potpuni komplement. Zapis u potpunom komplementu (engl. two's complement) označenih brojeva zadovoljava sledeće uslove:

1. Nula i pozitivni brojevi se zapisuju na isti način kao da su u pitanju neoznačeni brojevi, pri čemu u njihovom zapisu prva cifra mora da bude 0.

2. Sabiranje se sprovodi na isti način kao da su u pitanju neoznačeni brojevi, pri čemu se prenos sa poslednje pozicije zanemaruje.

Na primer, broj +100 se u potpunom komplementu zapisuje kao $(0\,1100100)_2^8$. Nula se zapisuje kao $(0\,0000000)_2^8$. Zapis broja −100 u obliku $(\ldots)_2^8$ se može odrediti na sledeći način. Zbir brojeva −100 i +100 mora da bude 0.

	binarno	dekadno
	????????	-100
+	01100100	+100
1	00000000	0

Analizom traženog sabiranja cifru po cifru, počevši od poslednje, sledi da se -100 mora zapisati kao $(10011100)_2^8$. Do ovoga je moguće doći i na sledeći način. Ukoliko je poznat zapis broja x, zapis njemu suprotnog broja je moguće odrediti iz uslova da je $x+(-x) = (1\,00\ldots00)_2^{n+1}$. Pošto je $(1\,00\ldots00)_2^{n+1} = (11\ldots11)_2^n+1$, zapis broja $(-x)$ je moguće odrediti tako što se izračuna $(11\ldots11)_2^n - x + 1$. Izračunavanje razlike $(11\ldots11)_2^n - x$ se svodi na *komplementiranje* svake pojedinačne cifre broja x. Tako se određivanje zapisa broja -100 može opisati na sledeći način:

	01100100	$+100$
	10011011	komplementiranje
$+$	1	
	10011100	

Kao što je traženo, zapisi svih pozitivnih brojeva i nule počinju cifrom 0 dok zapisi negativnih brojeva počinju sa 1.

Broj -2^{n-1} je jedini izuzetak u opštem postupku određivanja zapisa u potpunom komplementu. Zapis broja $(100\ldots00)_2^n$ je sam sebi komplementaran, a pošto počinje cifrom 1, uzima se da on predstavlja zapis najmanjeg zapisivog negativnog broja, tj. broja -2^{n-1}. Zahvaljujući ovoj konvenciji, u zapisu potpunog komplementa $(\ldots)_2^n$ moguće je zapisati brojeve od -2^{n-1} do $2^{n-1}-1$. U sledećoj tabeli su dati rasponi za najčešće korišćene dužine zapise u potpunom komplementu:

broj bitova	raspon
8	od -128 do $+127$
16	od -32768 do $+32767$
32	od -2147483648 do $+2147483647$

Kao što je rečeno, za sabiranje brojeva zapisanih u potpunom komplementu može se koristiti opšti postupak za sabiranje neoznačenih brojeva (pri čemu se podrazumeva da može doći do prekoračenja, tj. da neke cifre rezultata ne mogu biti upisane u raspoloživ broj mesta). To ne važi samo za sabiranje, već i za oduzimanje i množenje (pri čemu kod realizacije množenja u procesoru tj. u mašinski zavisnim jezicima postoje određene razlike između množenja označenih i neoznačenih brojeva). Ovo pogodno svojstvo važi i kada je jedan od argumenata broj -2^{n-1} (koji se je jedini izuzetak u opštem postupku određivanja zapisa). Sve ovo su važni razlozi za korišćenje potpunog komplementa u računarima.

2.2.3 Zapis realnih brojeva

Pored celobrojnih dostupni u programskim jezicima su po pravilu podržani i realni brojevni tipovi. Realne brojeve je u računaru mnogo komplikovanije predstaviti nego cele brojeve. Obično je moguće predstaviti samo određeni podskup skupa realnih brojeva i to podskup skupa racionalnih brojeva. Interni zapis celog broja i njemu odgovarajućeg realnog se ne poklapaju (nule i jedinice kojima se oni zapisuju nisu iste), čak i kada se isti broj bitova koristi za njihov zapis. Fiksiranjem bitova koji će se odvojiti za zapis nekog realnog broja, određuje se i broj mogućih različitih brojeva koji se mogu zapisati. Svakoj kombinaciji nula i jedinica pridružuje se neki realan broj. Kod celih brojeva odlučivano je da li će se takvim kombinacijama pridruživati samo pozitivni ili i pozitivni i negativni brojevi i kada je to odlučeno, u principu je jasno koji skup celih brojeva može biti zapisan (taj skup vrednosti je uvek neki povezan raspon celih brojeva). U zapisu realnih brojeva stvari su komplikovanije jer je potrebno napraviti kompromis između širine raspona brojeva koji se mogu zapisati (slično kao i kod celih brojeva), ali i između preciznosti brojeva koji se mogu zapisati. Dakle, kao i celobrojni tipovi, realni tipovi imaju određen raspon vrednosti koji se njima može predstaviti, ali i određenu preciznost zapisa (nju obično doživljavamo kao broj decimala, mada to tumačenje, kao što ćemo uskoro videti, nije uvek u potpunosti tačno).

Osnovni načini zapisivanja realnih brojeva su zapis u *fiksnom zarezu* i zapis u *pokretnom zarezu*. Zapis u fiksnom zarezu podrazumeva da se posebno zapisuje znak broja, zatim ceo deo broja i zatim njegov razlomljeni deo (njegove decimale). Broj cifara za zapis celog dela i za zapis razlomljenog dela je fiksiran i jednak je za sve brojeve u okviru istog tipa koji koristi zapis u fiksnom zarezu. Zapis u pokretnom zarezu podrazumeva oblik $\pm m \cdot b^e$. Vrednost b je osnova koja se podrazumeva (danas je to obično 2, mada se nekada koristila i vrednost 16, dok se u svakodnevnoj matematičkoj praksi često brojevi izražavaju na ovaj način uz korišćenje osnove 10), m se naziva mantisa, a vrednost e naziva se eksponent. Broj cifara (obično binarnih) za zapis mantise i broj cifara za zapis eksponenta je fiksiran i jednak je za sve brojeve u okviru istog tipa koji koristi zapis u pokretnom zarezu. Zapisivanje brojeva u pokretnom zarezu propisano je standardom *IEEE 754* iz 1985. godine, međunarodne asocijacije *Institut inženjera elektrotehnike i elektronike*, IEEE (engl. *Institute of Electrical and Electronics Engineers*). Ovaj standard predviđa da se određene kombinacije bitova odvoje za zapis tzv. specijalnih vrednosti (beskonačnih vrednosti, grešaka u izračunavanju i slično).

Ilustrujmo osnovne principe zapisa realnih brojeva na dekadnom zapisu nenegativnih realnih brojeva (kao što smo rekli, u računaru se ovakav zapis interno ne koristi, već se koristi binarni zapis). Zamislimo da imamo tri dekadne cifre koje možemo iskoristiti za zapis. Dakle, imamo 1000 brojeva koji se mogu zapisati. Prva mogućnost je da se određeni broj cifara upotrebi za zapis celog dela, a određeni broj cifara za zapis decimala i takav način zapisa predstavlja zapis u fiksnom zarezu. Na primer, ako se jedna cifra upotrebi za decimale moći ćemo zapisivati vrednosti 00,0, 00,1, ..., 99,8 i 99,9. Ako se se za decimale odvoje dva mesta, onda možemo zapisivati vrednosti 0,00, 0,01, ..., 9,98 i 9,99. Ovim smo dobili bolju preciznost (svaki broj je zapisan sa dve, umesto sa jednom decimalom), ali smo to platili mnogo užim rasponom brojeva koje možemo zapisati (umesto širine oko 100, dobili smo raspono širine oko 10). U računaru se fiksni zarez često ostvaruje binarno (određeni broj bitova kodira ceo, a određeni broj bitova kodira razlomljeni deo), pri čemu se uvek jedan bit ostavlja za predstavljanje znaka broja čime se omogućava zapis i pozitivnih i negativnih vrednosti.

Umesto fiksnog zareza, u računarima se mnogo češće koristi pokretni zarez. U takvom zapisu, naše tri dekadne cifre podelićemo tako da dve odlaze za zapis mantise, a jednu za zapis eksponenta (pa će zapis biti oblika $m_1m_2e_3$). Ako su prve dve cifre m_1 i m_2, tumačićemo da je mantisa $0,m_1m_2$. Ako je treća cifra u zapisu e_3, tumačićemo da eksponent $e = e_3 - 4$ (ovim postižemo da vrednosti eksponenta mogu da budu između -4 i 5 tj. da mogu da budu i pozitivne i negativne). Pogledajmo neke zapise koje na taj način dobijamo.

zapis	vrednost	zapis	vrednost	...	zapis	vrednost	zapis	vrednost
010	$0{,}01 \cdot 10^{-4} = 0{,}000001$	011	$0{,}01 \cdot 10^{-3} = 0{,}00001$	...	018	$0{,}01 \cdot 10^{4} = 100{,}0$	019	$0{,}01 \cdot 10^{5} = 1\,000{,}0$
020	$0{,}02 \cdot 10^{-4} = 0{,}000002$	021	$0{,}02 \cdot 10^{-3} = 0{,}00002$	...	028	$0{,}02 \cdot 10^{4} = 200{,}0$	029	$0{,}02 \cdot 10^{5} = 2\,000{,}0$
980	$0{,}98 \cdot 10^{-4} = 0{,}000098$	981	$0{,}98 \cdot 10^{-3} = 0{,}00098$	...	988	$0{,}98 \cdot 10^{4} = 9800{,}0$	989	$0{,}98 \cdot 10^{5} = 98\,000{,}0$
990	$0{,}99 \cdot 10^{-4} = 0{,}000099$	991	$0{,}99 \cdot 10^{-3} = 0{,}00099$	...	998	$0{,}99 \cdot 10^{4} = 9900{,}0$	999	$0{,}99 \cdot 10^{5} = 99\,000{,}0$

Raspon je prilično širok (od $0,000001$ do $99\,000,0$ tj. od oko 10^{-6} do oko 10^5) i mnogo širi nego što je to bilo kod fiksnog zareza, ali gustina zapisa je neravnomerno raspoređena, što nije bio slučaj kod fiksnog zareza. Kod malih brojeva preciznost zapisa je mnogo veća nego kod velikih. Na primer, kod malih brojeva možemo zapisivati veliki broj decimala, ali nijedan broj između 98 000 i 99 000 nije moguće zapisati (brojevi iz tog raspona bi se morali zaokružiti na neki od ova dva broja). Ovo nije preveliki problem, jer nam obično kod velikih brojeva preciznost nije toliko bitna koliko kod malih (matematički gledano, relativna greška koja nastaje usled zaokruživanja se ne menja puno kroz ceo raspon). Dve važne komponente koje karakterišu svaki zapis u pokretnom zarezu su raspon brojeva koji se mogu zapisati (u našem primeru to je bilo od oko 10^{-6} do oko 10^5) i on je skoro potpuno određen širinom eksponentna, kao i broj dekadnih značajnih cifara (u našem primer, imamo dve dekadne značajne cifre) i on je skoro potpuno određeno širinom mantise.

2.3 Zapis teksta

Međunarodna organizacija za standardizaciju, *ISO* (engl. *International Standard Organization*) definiše tekst (ili dokument) kao *„informaciju namenjenu ljudskom sporazumevanju koja može biti prikazana u dvodimenzionalnom obliku. Tekst se sastoji od grafičkih elemenata kao što su karakteri, geometrijski ili fotografski elementi ili njihove kombinacije, koji čine sadržaj dokumenta“*. Iako se tekst obično zamišlja kao dvodimenzioni objekat, u računarima se tekst predstavlja kao jednodimenzioni (linearni) niz karaktera koji pripadaju određenom unapred fiksiranom skupu karaktera. U zapisu teksta, koriste se specijalni karakteri koji označavaju prelazak u novi red, tabulator, kraj teksta i slično.

Osnovna ideja koja omogućava zapis teksta u računarima je da se svakom karakteru pridruži određen (neoznačeni) ceo broj (a koji se interno u računarima zapisuje binarno) i to na unapred dogovoreni način. Ovi brojevi se nazivaju *kodovima karaktera* (engl. *character codes*). Tehnička ograničenja ranih računara kao i neravnomeran razvoj računarstva između različitih zemalja, doveli su do toga da postoji više različitih standardnih tabela koje dodeljuju numeričke kodove karakterima. U zavisnosti od broja bitova potrebnih za kodiranje karaktera, razlikuju se 7-bitni kodovi, 8-bitni kodovi, 16-bitni kodovi, 32-bitni kodovi, kao i kodiranja promenljive dužine koja različitim karakterima dodeljuju kodove različite dužine. Tabele koje sadrže karaktere i njima pridružene kodove obično se nazivaju *kodne strane* (engl. *code page*).

Postoji veoma jasna razlika između karaktera i njihove grafičke reprezentacije. Elementi pisanog teksta koji najčešće predstavljaju grafičke reprezentacije pojedinih karaktera nazivaju se *glifovi* (engl. *glyph*), a skupovi glifova nazivaju se *fontovi* (engl. *font*). Korespondencija između karaktera i glifova ne mora biti jednoznačna. Naime, softver koji prikazuje tekst može više karaktera predstaviti jednim glifom (to su takozvane *ligature*, kao na primer glif za karaktere „f“ i „i“: fi), dok jedan isti karakter može biti predstavljen različitim glifovima u zavisnosti od svoje pozicije u reči. Takođe, moguće je da određeni fontovi ne sadrže glifove za određene karaktere i u tom slučaju se tekst ne prikazuje na željeni način, bez obzira što je ispravno kodiran. Fontovi koji

se obično instaliraju uz operativni sistem sadrže glifove za karaktere koji su popisani na takozvanoj *WGL4* listi (*Windows Glyph List 4*) koja sadrži uglavnom karaktere korišćene u evropskim jezicima, dok je za ispravan prikaz, na primer, kineskih karaktera, potrebno instalirati dodatne fontove. Specijalnim karakterima se najčešće ne pridružuju zasebni grafički likovi.

Englesko govorno područje. Tokom razvoja računarstva, broj karaktera koje je bilo poželjno kodirati je postajao sve veći. Pošto je računarstvo u ranim fazama bilo razvijano uglavnom u zemljama engleskog govornog područja, bilo je potrebno predstaviti sledeće karaktere:

- Mala slova engleskog alfabeta: `a, b, ... , z`
- Velika slova engleskog alfabeta: `A, B, ... , Z`
- Cifre `0, 1, ..., 9`
- Interpunkcijske znake: `, . : ; + * - _ ( ) [ ] { }` ...
- Specijalne znake: kraj reda, tabulator, ...

Standardne tabele kodova ovih karaktera su se pojavile još tokom 1960-ih godina. Najrasprostranjenije od njih su:

- *EBCDIC* - IBM-ov standard, korišćen uglavnom na mejnfrejm računarima, pogodan za bušene kartice.
- *ASCII* - standard iz koga se razvila većina danas korišćenih standarda za zapis karaktera.

ASCII. *ASCII (American Standard Code for Information Interchange)* je standard uspostavljen 1968. godine od strane *Američkog nacionalnog instituta za standarde*, (engl. *American National Standard Institute*) koji definiše sedmobitan zapis koda svakog karaktera što daje mogućnost zapisivanja ukupno 128 različitih karaktera, pri čemu nekoliko kodova ima dozvoljeno slobodno korišćenje. *ISO* takođe delimično definiše ASCII tablicu kao deo svog standarda *ISO 646 (US)*.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	NUL	STX	SOT	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Slika 2.2: ASCII tablica

Osnovne osobine ASCII standarda su:

- Prva 32 karaktera – od $(00)_{16}$ do $(1F)_{16}$ – su specijalni kontrolni karakteri.
- Ukupno 95 karaktera ima pridružene grafičke likove (engl. printable characters).
- Cifre 0-9 predstavljene su kodovima $(30)_{16}$ do $(39)_{16}$, tako da se njihov ASCII zapis jednostavno dobija dodavanjem prefiksa 011 na njihov binarni zapis.
- Kôdovi velikih i malih slova se razlikuju u samo jednom bitu u binarnoj reprezentaciji. Na primer, A se kodira brojem $(41)_{16}$ odnosno $(100\,0001)_2$, dok se a kodira brojem $(61)_{16}$ odnosno $(110\,0001)_2$. Ovo omogućava da se konverzija veličine slova u oba smera može vršiti efikasno.
- Slova su poređana u *kolacionu sekvencu*, u skladu sa engleskim alfabetom.

Različiti operativni sistemi predviđaju različito kodiranje oznake za prelazak u novi red. Tako operativni sistem Windows podrazumeva da se prelazak u novi red kodira sa dva kontrolna karaktera i to CR (carriage return) predstavljen kodom $(0D)_{16}$ i LF (line feed) predstavljen kodom $(0A)_{16}$, operativni sistem Unix i njegovi derivati (pre svega Linux) podrazumevaju da se koristi samo karakter LF, dok MacOS podrazumeva korišćenje samo karaktera CR.

Nacionalne varijante ASCII tablice i ISO 646. Tokom 1980-ih, *Jugoslovenski zavod za standarde* definisao je standard *YU-ASCII (YUSCII, JUS I.B1.002, JUS I.B1.003)* kao deo standarda ISO 646, tako što su kodovi koji imaju slobodno korišćenje (a koji u ASCII tabeli uobičajeno kodiraju zagrade i određene interpunkcijske znakove) dodeljeni našim dijakriticima:

YUSCII	ASCII	kôd	YUSCII	ASCII	kôd
Ž	@	$(40)_{16}$	ž	`	$(60)_{16}$
Š	[	$(5B)_{16}$	š	{	$(7B)_{16}$
Đ	\	$(5C)_{16}$	đ	|	$(7C)_{16}$
Ć	]	$(5D)_{16}$	ć	}	$(7D)_{16}$
Č	^	$(5E)_{16}$	č	~	$(7E)_{16}$

Osnovne mane YUSCII kodiranja su to što ne poštuje abecedni poredak, kao i to da su neke zagrade i važni interpunkcijski znaci izostavljeni.

8-bitna proširenja ASCII tabele. Podaci se u računaru obično zapisuju bajt po bajt. S obzirom na to da je ASCII sedmobitni standard, ASCII karakteri se zapisuju tako što se njihov sedmobitni kôd proširi vodećom nulom. Ovo znači da jednobajtni zapisi u kojima je vodeća cifra 1, tj. raspon od $(80)_{16}$ do $(FF)_{16}$ nisu iskorišćeni. Međutim, ni ovih dodatnih 128 kodova nije dovoljno da se kodiraju svi karakteri koji su potrebni za zapis tekstova na svim jezicima (ne samo na engleskom). Zbog toga je, umesto jedinstvene tabele koja bi proširivala ASCII na 256 karaktera, standardizovano nekoliko takvih tabela, pri čemu svaka od tabela sadrži karaktere potrebne za zapis određenog jezika odnosno određene grupe jezika. Praktičan problem je što postoji dvostruka standardizacija ovako kreiranih kodnih strana i to od strane ISO (International Standard Organization) i od strane značajnih korporacija, pre svega kompanije *Microsoft*.

ISO je definisao familiju 8-bitnih kodnih strana koje nose zajedničku oznaku *ISO/IEC 8859* (kodovi od $(00)_{16}$ do $(1F)_{16}$, $(7F)_{16}$ i od $(80)_{16}$ do $(9F)_{16}$ ostali su nedefinisani ovim standardom, iako se često u praksi popunjavaju određenim kontrolnim karakterima):

ISO-8859-1	Latin 1	većina zapadnoevropskih jezika
ISO-8859-2	Latin 2	centralno i istočnoevropski jezici
ISO-8859-3	Latin 3	južnoevropski jezici
ISO-8859-4	Latin 4	severnoevropski jezici
ISO-8859-5	Latin/Cyrillic	ćirilica većine slovenskih jezika
ISO-8859-6	Latin/Arabic	najčešće korišćeni arapski
ISO-8859-7	Latin/Greek	moderni grčki alfabet
ISO-8859-8	Latin/Hebrew	moderni hebrejski alfabet

Kompanija Microsoft definisala je familju 8-bitnih strana koje se označavaju kao *Windows-125x* (ove strane se nekada nazivaju i *ANSI*). Za srpski jezik, značajne su kodne strane:

Windows-1250	centralnoevropski i istočnoevropski jezici
Windows-1251	ćirilica većine slovenskih jezika
Windows-1252	(često se neispravno naziva i ANSI) zapadnoevropski jezici

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8																
9																
A	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Đ	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	

Slika 2.3: ISO-8859-1 tablica

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
8																
9																
A	NBSP	Ą	˘	Ł	¤	Ľ	Ś	§	¨	Š	Ş	Ť	Ź	SHY	Ž	Ż
B	°	ą	˛	ł	´	ľ	ś	ˇ	¸	š	ş	ť	ź	˝	ž	ż
C	Ŕ	Á	Â	Ă	Ä	Ĺ	Ć	Ç	Č	É	Ę	Ë	Ě	Í	Î	Ď
D	Đ	Ń	Ň	Ó	Ô	Ő	Ö	×	Ř	Ů	Ú	Ű	Ü	Ý	Ţ	ß
E	ŕ	á	â	ă	ä	ĺ	ć	ç	č	é	ę	ë	ě	í	î	ď
F	đ	ń	ň	ó	ô	ő	ö	÷	ř	ů	ú	ű	ü	ý	ţ	

Slika 2.4: ISO-8859-2 tablica

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
8	€		‚		„	…	†	‡		‰	Š	‹	Ś	Ť	Ž	Ź
9		‘	’	“	”	•	–	—		™	š	›	ś	ť	ž	ź
A		ˇ	˘	Ł	¤	Ą	¦	§	¨	©	Ş	«	¬		®	Ż
B	°	±	˛	ł	´	µ	¶	·	¸	ą	ş	»	Ľ	˝	ľ	ż
C	Ŕ	Á	Â	Ă	Ä	Ĺ	Ć	Ç	Č	É	Ę	Ë	Ě	Í	Î	Ď
D	Đ	Ń	Ň	Ó	Ô	Ő	Ö	×	Ř	Ů	Ú	Ű	Ü	Ý	Ţ	ß
E	ŕ	á	â	ă	ä	ĺ	ć	ç	č	é	ę	ë	ě	í	î	ď
F	đ	ń	ň	ó	ô	ő	ö	÷	ř	ů	ú	ű	ü	ý	ţ	

Slika 2.5: Windows-1250 tablica

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
8																
9																
A	NBSP	Ё	Ђ	Ѓ	Є	S	I	Ї	J	Љ	Њ	Ћ	Ќ	SHY	Ў	Џ
B	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
C	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
D	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
E	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
F	№	ё	ђ	ѓ	є	s	i	ї	j	љ	њ	ћ	ќ	§	ў	

Slika 2.6: ISO-8859-5 tablica

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
8	Ђ		‚		„	…	†	‡		‰	Љ	‹	Њ	Ќ	Ћ	Џ
9		‘	’	“	”	•	–	—		™	љ	›	њ	ќ	ћ	џ
A		Ў	ў	J	¤	Ґ	¦	§	Ё	©	Є	«	¬		®	Ї
B	°	±	I	i	ґ	µ	¶	·	ё	№	є	»	j	S	s	ї
C	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
D	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
E	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
F	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	

Slika 2.7: Windows-1251 tablica

Unicode. Iako navedene kodne strane omogućavaju kodiranje tekstova koji nisu na engleskom jeziku, nije moguće, na primer, u istom tekstu koristiti i ćirilicu i latinicu. Takođe, za azijske jezike nije dovoljno 256 mesta za zapis svih karaktera. Pošto je kapacitet računara vremenom rastao, postepeno se krenulo sa standardizacijom skupova karaktera koji karaktere kodiraju sa više od jednog bajta. Kasnih 1980-ih, dve velike organizacije započele su standardizaciju tzv. univerzalnog skupa karaktera (engl. Universal Character Set — UCS). To su bili ISO, kroz standard *10646* i projekat *Unicode*, organizovan i finansiran uglavnom od strane američkih kompanija koje su se bavile proizvodnjom višejezičkog softvera (Xerox Parc, Apple, Sun Microsystems, Microsoft, ...).

ISO 10646 zamišljen je kao četvorobajtni standard. Prvih 65536 karaktera koristi se kao osnovni višejezički skup karaktera, dok je preostali prostor ostavljen kao proširenje za drevne jezike, naučnu notaciju i slično.

Unicode je za cilj imao da bude:

- univerzalan (UNIversal) — sadrži sve savremene jezike sa pismom;
- jedinstven (UNIque) — bez dupliranja karaktera - kodiraju se pisma, a ne jezici;
- uniforman (UNIform) — svaki karakter sa istim brojem bitova.

Početna verzija Unicode standarda svakom karakteru dodeljuje dvobajtni kôd (tako da kôd svakog karaktera sadrži tačno 4 heksadekadne cifre). Dakle, moguće je dodeliti kodove za ukupno $2^{16} = 65536$ različitih karaktera. S vremenom se shvatilo da dva bajta neće biti dovoljno za zapis svih karaktera koji se koriste na planeti, pa je odlučeno da se skup karaktera proširi i Unicode danas dodeljuje kodove od $(000000)_{16}$ do $(10FFFF)_{16}$ podeljenih u 17 tzv. ravni, pri čemu svaka ravan sadrži 65536 karaktera. U najčešćoj upotrebi je *osnovna višejezička ravan* (engl. *basic multilingual plane*) koja sadrži većinu danas korišćenih karaktera (uključujući i CJK — Chinese, Japanese, Korean — karaktere koji se najčešće koriste) čiji su kodovi između $(0000)_{16}$ i $(FFFF)_{16}$.

Vremenom su se pomenuta dva projekta UCS i Unicode združila i danas postoji izuzetno preklapanje između ova dva standarda.

U sledećoj tabeli je naveden raspored određenih grupa karaktera u osnovnoj višejezičkoj ravni:

0020-007E	ASCII printable
00A0-00FF	Latin-1
0100-017F	Latin extended A (osnovno proširenje latinice, sadrži sve naše dijakritike)
0180-027F	Latin extended B
...	
0370-03FF	grčki alfabet
0400-04FF	ćirilica
...	
2000-2FFF	specijalni karakteri
3000-3FFF	CJK (Chinese-Japanese-Korean) simboli
...	

Unicode standard u suštini predstavlja veliku tabelu koja svakom karakteru dodeljuje broj. Standardi koji opisuju kako se niske karaktera prevode u nizove bajtova se definišu dodatno.

UCS-2. ISO definiše UCS-2 standard koji svaki Unicode karakter osnovne višejezičke ravni jednostavno zapisuje sa odgovarajuća dva bajta.

UTF-8. Latinični tekstovi kodirani korišćenjem UCS-2 standarda sadrže veliki broj nula. Ne samo što te nule zauzimaju dosta prostora, već zbog njih softver koji je razvijen za rad sa dokumentima u ASCII formatu ne može da radi bez izmena nad dokumentima kodiranim korišćenjem UCS-2 standarda. *Unicode Transformation Format (UTF-8)* je algoritam koji svakom dvobajtnom Unicode karakteru dodeljuje određeni niz bajtova čija dužina varira od 1 do najviše 3. UTF je ASCII kompatibilan, što znači da se ASCII karakteri zapisuju pomoću jednog bajta, na standardni način. Konverzija se vrši na osnovu sledećih pravila:

raspon	binarno zapisan Unicode kôd	binarno zapisan UTF-8 kôd
0000-007F	00000000 0xxxxxxx	0xxxxxxx
0080-07FF	00000yyy yyxxxxxx	110yyyyy 10xxxxxx
0800-FFFF	zzzzyyyy yyxxxxxx	1110zzzz 10yyyyyy 10xxxxxx

Na primer, karakter A koji se nalazi u ASCII tabeli ima Unicode kôd $(0041)_{16} = (0000\,0000\ 0100\,0001)_2$, pa se na osnovu prvog reda prethodne tabele u UTF-8 kodiranju zapisuje kao $(01000001)_2 = (41)_{16}$. Karakter Š ima Unicode kôd $(0160)_{16} = (0000\,0001\ 0110\,0000)_2$. Na njega se primenjuje drugi red prethodne tabele i dobija se da je njegov UTF-8 zapis $(1100\,0101\ 1010\,0000)_2$ tj. $(C5A0)_{16}$. Opisani konverzioni algoritam omogućava da se čitanje samo početka jednog bajta odredi da li je u pitanju karakter zapisan korišćenjem jednog, dva ili tri bajta.

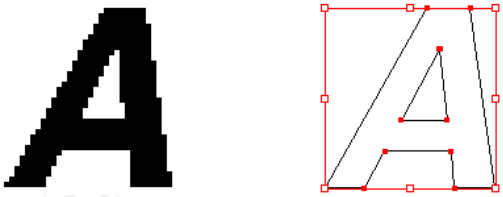
2.4 Zapis multimedijalnih sadržaja

Računari imaju sve veću ulogu u većini oblasti svakodnevnog života. Od mašina koje su pre svega služile za izvođenje vojnih i naučnih izračunavanja, računari su postali i sredstvo za kućnu zabavu (gledanje filmova, slušanje muzike), izvor informacija (internet) i nezaobilazno sredstvo u komunikaciji (elektronska pošta (engl. e-mail), ćaskanje (engl. chat, instant messaging), video konferencije, telefoniranje korišćenjem interneta (VoIP), ...). Nagli razvoj i proširivanje osnovne namene računara je posledica široke raspoloživosti velike količine multimedijalnih informacija (slika, zvuka, filmova, ...) koje su zapisane u digitalnom formatu. Sve ovo je posledica tehnološkog napretka koji je omogućio jednostavno i jeftino digitalizovanje signala, skladištenje velike količine digitalno zapisanih informacija, kao i njihov brz prenos i obradu.

2.4.1 Zapis slika

Slike se u računaru zapisuju koristeći *vektorski* zapis, *rasterski* zapis ili *kombinovani zapis*.

U vektorskom obliku, zapis slike sastoji se od konačnog broja geometrijskih figura (tačaka, linija, krivih, poligona), pri čemu se svaka figura predstavlja koncizno svojim koordinatama ili jednačinom u Dekartovoj ravni. Slike koje računari generišu često koriste vektorsku grafiku. Vektorski zapisane slike često zauzimaju manje prostora, dozvoljavaju uvećavanje (engl. zooming) bez gubitaka na kvalitetu prikaza i mogu se lakše preuređivati, s obzirom na to da se objekti mogu nezavisno jedan od drugoga pomerati, menjati, dodavati i uklanjati.



Slika 2.8: Primer slike u rasterskoj (levo) i vektorskoj (desno) grafici

U rasterskom zapisu, slika je predstavljena pravougaonom matricom komponenti koje se nazivaju pikseli (engl. pixel - PICture ELement). Svaki piksel je individualan i opisan jednom bojom. Raster nastaje kao rezultat digitalizacije slike. Rasterska grafika se još naziva i bitmapirana grafika. Uređaji za prikaz (monitori, projektori), kao i uređaji za digitalno snimanje slika (fotoaparati, skeneri) koriste rasterski zapis.

Modeli boja. Za predstavljanje crno-belih slika, dovoljno je boju predstaviti intenzitetom svetlosti. Različite količine svetlosti se diskretizuju u konačan broj nivoa osvetljenja čime se dobija određeni broj nijansi sive boje. Ovakav model boja se naziva *grayscale*. Ukoliko se za zapis informacije o količini svetlosti koristi 1 bajt, ukupan broj nijansi sive boje je 256. U slučaju da se slika predstavlja sa ukupno dve boje (na primer, crna i bela, kao kod skeniranog teksta nekog dokumenta) koristi se model pod nazivom *Duotone* i boja se tada predstavlja jednim bitom.

U RGB modelu boja, kombinovanjem crvene (R), zelene (G) i plave (B) komponente svetlosti reprezentuju se ostale boje. Tako se, na primer, kombinovanjem crvene i zelene boje reprezentuje žuta boja. Bela boja se reprezentuje maksimalnim vrednosti sve tri osnovne komponente, dok se crna boja reprezentuje minimalnim vrednostima osnovnih komponenti. U ovom modelu, zapis boje čini informacija o intenzitetu crvene, plave i zelene komponente. RGB model boja se koristi kod uređaja koji boje prikazuju kombinovanjem svetlosti (monitori, projektori, ...). Ukoliko se za informaciju o svakoj komponenti koristi po jedan bajt, ukupan broj bajtova za zapis informacije o boji je 3, te je moguće koristiti $2^{24} = 16777216$ različitih boja. Ovaj model se često naziva i *TrueColor* model boja.

Za razliku od aditivnog RGB modela boja, kod kojeg se bela boja dobija kombinovanjem svetlosti tri osnovne komponente, u štampi se koristi subtraktivni CMY (Cyan-Magenta-Yellow) model boje kod kojeg

se boje dobijaju kombinovanjem obojenih pigmenata na belom papiru. Kako se potpuno crna teško dobija mešanjem drugih pigmenata, a i kako je njena upotreba obično daleko najčešća, obično se prilikom štampanja uz CMY pigmente koristi i crni pigment čime se dobija model CMYK.

Za obradu slika pogodni su HSL ili HSV (poznat i kao HSB) model boja. U ovom modelu, svaka boja se reprezentuje Hue (H) komponentom (koja predstavlja ton boje), Saturation (S) komponentom (koja predstavlja zasićenost boje odnosno njenu „jarkost“) i Lightness (L), Value (V) ili Brightness (B) komponentom (koja predstavlja osvetljenost).

Formati zapisa rasterskih slika. Rasterske slike su reprezentovane matricom piksela, pri čemu se za svaki piksel čuva informacija o njegovoj boji. Dimenzije ove matrice predstavljaju tzv. *apsolutnu rezoluciju* slike. Apsolutna rezolucija i model boja koji se koristi određuju broj bajtova potrebnih za zapis slike. Na primer, ukoliko je apsolutna rezolucija slike 800×600 piksela, pri čemu se koristi RGB model boje sa 3 bajta po pikselu, potrebno je ukupno $800 \cdot 600 \cdot 3B = 1440000B \approx 1.373MB$ za zapis slike. Da bi se smanjila količina informacija potrebnih za zapis slike, koriste se tehnike kompresije i to *(i)* kompresije bez gubitka (engl. lossless), i *(ii)* kompresije sa gubitkom (engl. lossy). Najčešće korišćeni formati u kojima se koristi tehnike kompresije bez gubitka danas su GIF i PNG (koji se koriste za zapis dijagrama i sličnih računarski generisanih slika), dok je najčešće korišćeni format sa gubitkom JPEG (koji se obično koristi za fotografije).

2.4.2 Zapis zvuka

Zvučni talas predstavlja oscilaciju pritiska koja se prenosi kroz vazduh ili neki drugi medijum (tečnost, čvrsto telo). Digitalizacija zvuka se vrši merenjem i zapisivanjem vazdušnog pritiska u kratkim vremenskim intervalima. Osnovni parametri koji opisuju zvučni signal su njegova amplituda (koja odgovara „glasnoći“) i frekvencija (koja odgovara „visini“). Pošto ljudsko uho obično čuje raspon frekvencija od 20Hz do 20kHz, dovoljno je koristiti frekvenciju odabiranja 40kHz, tj. dovoljno je izvršiti odabiranje oko 40 000 puta u sekundi. Na primer, AudioCD standard koji se koristi prilikom snimanja običnih audio kompakt diskova, propisuje frekvenciju odabiranja 44.1kHz. Za postizanje još većeg kvaliteta, neki standardi (miniDV, DVD, digital TV) propisuju odabiranje na frekvenciji 48kHz. Ukoliko se snima ili prenosi samo ljudski govor (na primer, u mobilnoj telefoniji), frekvencije odabiranja mogu biti i znatno manje. Drugi važan parametar digitalizacije je broj bitova kojim se zapisuje svaki pojedinačni odbirak. Najčešće se koristi 2 bajta po odbirku (16 bitova), čime se dobija mogućnost zapisa $2^{16} = 65536$ različitih nivoa amplitude.

Da bi se dobio prostorni osećaj zvuka, primenjuje se tehnika višekanalnog snimanja zvuka. U ovom slučaju, svaki kanal se nezavisno snima posebnim mikrofonom i reprodukuje na posebnom zvučniku. *Stereo* zvuk podrazumeva snimanje zvuka sa dva kanala. *Surround* sistemi podrazumevaju snimanje sa više od dva kanala (od 3 pa čak i do 10), pri čemu se često jedan poseban kanal izdvaja za specijalno snimanje niskofrekvencijskih komponenti zvuka (tzv. bas). Najpoznatiji takvi sistemi su 5+1 gde se koristi 5 regularnih i jedan bas kanal.

Kao i slika, nekomprimovan zvuk zauzima puno prostora. Na primer, jedan minut stereo zvuka snimljenog u AudioCD formatu zauzima $2 \cdot 44100 \frac{sample}{sec} \cdot 60sec \cdot 2 \frac{B}{sample} = 10584000B \approx 10.1MB$. Zbog toga se koriste tehnike kompresije, od kojeg je danas najkorišćenija tehnika kompresije sa gubitkom MP3 (MPEG-1 Audio-Layer 3). MP3 kompresija se zasniva na tzv. psiho-akustici koja proučava koje je komponente moguće ukloniti iz zvučnog signala, a da pritom ljudsko uho ne oseti gubitak kvaliteta signala.

Pitanja i zadaci za vežbu

Pitanje 2.1. *Kako su u digitalnom računaru zapisani svi podaci? Koliko se cifara obično koristi za njihov zapis?*

Pitanje 2.2. *Koje su osnovne prednosti digitalnog zapisa podataka u odnosu na analogni? Koji su osnovni problemi u korišćenju digitalnog zapisa podataka?*

Pitanje 2.3. *Šta je Hornerova shema? Opisati Hornerova shemu pseudo-kodom.*

Pitanje 2.4. *Koje su prednosti zapisa u potpunom komplementu u odnosu na zapis u obliku označene apsolutne vrednosti?*

Pitanje 2.5. *Šta je to IEEE 754?*

Pitanje 2.6. *Šta je to glif, a šta font? Da li je jednoznačna veza između karaktera i glifova? Navesti primere.*

Pitanje 2.7. *Koliko bitova koristi ASCII standard? Šta čini prva 32 karaktera ASCII tabele? Kako se određuje binarni zapis karaktera koji predstavljaju cifre?*

Pitanje 2.8. *Navesti barem dve jednobajtne kodne strane koje sadrže ćirilične karaktere.*

Pitanje 2.9. *Nabrojati bar tri kodne sheme u kojima može da se zapiše reč* računarstvo.

Pitanje 2.10. *Koliko bitova koristi ASCII tabela karaktera, koliko YUSCII tabela, koliko ISO-8859-1, a koliko osnovna Unicode ravan? Koliko različitih karaktera ima u ovim tabelama?*

Pitanje 2.11. *Koja kodiranja teksta je moguće koristiti ukoliko se u okviru istog dokumenta želi zapisivanje teksta koji sadrži jedan pasus na srpskom (pisan latinicom), jedan pasus na nemačkom i jedan pasus na ruskom (pisan ćirilicom)?*

Pitanje 2.12. *U čemu je razlika između Unicode i UTF-8 kodiranja?*

Pitanje 2.13. *Prilikom prikazivanja filma, neki program prikazuje titlove tipa* `"raèunari æe biti..."`. *Objasniti u čemu je problem.*

Pitanje 2.14. *Šta je to piksel? Šta je to sempl?*

Zadatak 2.1. *Prevesti naredne brojeve u dekadni brojevni sistem:*
(a) $(10111001)_2$ *(b)* $(3C4)_{16}$ *(c)* $(734)_8$
Zadatak uraditi klasičnim postupkom, a zatim i korišćenjem Hornerove sheme. √

Zadatak 2.2. *Zapisati dekadni broj* 254 *u osnovama* 2, 8 *i* 16. √

Zadatak 2.3. *(a)*
Registar ima sadržaj
10101011010010001111010101010110011010111010101011100010100100 11.
Zapisati ovaj broj u heksadekadnom sistemu.

(b) Registar ima sadržaj $A3BF461C89BE23D7$. *Zapisati ovaj sadržaj u binarnom sistemu.*
√

Zadatak 2.4. *Na Vebu se boje obično predstavljaju šestocifrenim heksadekadnim kodovima u RGB sistemu: prve dve cifre odgovaraju količini crvene boje, naredne dve količini zelene i poslednje dve količini plave. Koja je količina RGB komponenti (na skali od 0-255) za boju* `#35A7DC`*? Kojim se kodom predstavlja čista žuta boja ako se zna da se dobija mešanjem crvene i zelene? Kako izgledaju kodovi za nijanse sive boje?* √

Zadatak 2.5. *U registru se zapisuju neoznačeni brojevi. Koji raspon brojeva može da se zapiše ukoliko je širina registra u bitovima:*
(a) 4 *(b)* 8 *(c)* 16 *(d)* 24 *(e)* 32 √

Zadatak 2.6. *Ukoliko se koristi binarni zapis neoznačenih brojeva širine* 8 *bitova, zapisati brojeve:*
(a) 12 *(b)* 123 *(c)* 255 *(d)* 300 √

Zadatak 2.7. *U registru se zapisuju brojevi u potpunom komplementu. Koji raspon brojeva može da se zapiše ukoliko je širina registra u bitovima:*
(a) 4 *(b)* 8 *(c)* 16 *(d)* 24 *(e)* 32 √

Zadatak 2.8. *Odrediti zapis narednih brojeva u binarnom potpunom komplementu širine* 8 *bitova:*
(a) 12 *(b)* −123 *(c)* 0 *(d)* −18 *(e)* −128 *(f)* 200 √

Zadatak 2.9. *Odrediti zapis brojeva -5 i 5 u potpunom komplementu dužine 6 bitova. Odrediti, takođe u potpunom komplementu dužine 6 bitova, zapis zbira i proizvoda ova dva broja.* √

Zadatak 2.10. *Ukoliko se zna da je korišćen binarni potpuni komplement širine* 8 *bitova, koji su brojevi zapisani?*
(a) 11011010 *(b)* 01010011 *(c)* 10000000 *(d)* 11111111
(e) 01111111 *(f)* 00000000
Šta predstavljaju dati zapisi ukoliko se zna da je korišćen zapis neoznačenih brojeva? √

Zadatak 2.11. *Odrediti broj bitova neophodan za kodiranje 30 različitih karaktera.* √

Zadatak 2.12. *Znajući da je dekadni kôd za karakter A 65, navesti kodirani zapis reči* `FAKULTET` *ASCII kodovima u heksadekadnom zapisu. Dekodirati sledeću reč zapisanu u ASCII kodu heksadekadno: 44 49 53 4B 52 45 54 4E 45.* √

Zadatak 2.13. *Korišćenjem ASCII tablice odrediti kodove kojima se zapisuje tekst:* `"Programiranje 1"`. *Kôdove zapisati heksadekadno, oktalno, dekadno i binarno. Šta je sa kodiranjem teksta* `Matematički fakultet`*?* √

Zadatak 2.14. *Za reči* `računarstvo`, `informatika`, *navesti da li ih je moguće kodirati narednim metodima i, ako jeste, koliko bajtova zauzimaju:*
(a) ASCII (b) Windows-1250 (c) ISO-8859-5
(d) ISO-8859-2 (e) Unicode (UCS-2) (f) UTF-8 √

Zadatak 2.15. *Odrediti (heksadekadno predstavljene) kodove kojima se zapisuje tekst* `kružić` *u UCS-2 i UTF-8 kodiranjima. Rezultat proveriti korišćenjem HEX editora.* √

Zadatak 2.16. *HEX editori su programi koji omogućavaju direktno pregledanje, kreiranje i ažuriranje bajtova koji sačinjavaju sadržaja datoteka. Korišćenjem HEX editora pregledati sadržaj nekoliko datoteka različite vrste (tekstualnih, izvršivih programa, slika, zvučnih zapisa, video zapisa, . . .).*

Zadatak 2.17. *Uz pomoć omiljenog editora teksta (ili nekog naprednijeg, ukoliko editor nema tražene mogućnosti) kreirati datoteku koja sadrži listu imena 10 vaših najomiljenijih filmova (pisano latinicom uz ispravno korišćenje dijakritika). Datoteka treba da bude kodirana kodiranjem:*
(a) Windows-1250 (b) ISO-8859-2 (c) Unicode (UCS-2) (d) UTF-8
Otvoriti zatim datoteku iz nekog pregledača Veba i proučiti šta se dešava kada se menja kodiranje koje pregledač koristi prilikom tumačenja datoteke (obično meni `View->Character encoding`*). Objasniti i unapred pokušati predvideti ishod (uz pomoć odgovarajućih tabela koje prikazuju kodne rasporede).*

Zadatak 2.18. *Za datu datoteku kodiranu UTF-8 kodiranjem, korišćenjem editora teksta ili nekog od specijalizovanih alata (na primer,* `iconv`*) rekodirati ovu datoteku u ISO-8859-2. Eksperimentisati i sa drugim kodiranjima.*

Zadatak 2.19. *Korišćenjem nekog naprednijeg grafičkog programa (na primer, GIMP ili Adobe Photoshop) videti kako se boja* `#B58A34` *predstavlja u CMY i HSB modelima.*

Glava 3

Algoritmi i izračunljivost

Neformalno govoreći, *algoritam*[1] je precizan opis postupka za rešavanje nekog problema u konačnom broju koraka. Algoritmi postoje u svim ljudskim delatnostima. U matematici, na primer, postoje algoritmi za sabiranje i za množenje prirodnih brojeva. U računarstvu, postoje algoritmi za određivanje najmanjeg elementa niza, uređivanje elemenata po veličini i slično. Da bi moglo da se diskutuje o tome šta se može a šta ne može izračunati algoritamski, potrebno je najpre precizno definisati pojam algoritma.

3.1 Formalizacije pojma algoritma

Precizni postupci za rešavanje matematičkih problema postojali su u vreme starogrčkih matematičara (na primer, Euklidov algoritam za određivanje najvećeg zajedničkog delioca dva broja), pa i pre toga. Ipak, sve do početka dvadesetog veka nije se uviđala potreba za preciznim definisanjem pojma algoritma. Tada je, u jeku reforme i novog utemeljivanja matematike, postavljeno pitanje da li postoji algoritam kojim se (pojednostavljeno rečeno) mogu dokazati sve matematičke teoreme[2]. Da bi se ovaj problem uopšte razmatrao, bilo je neophodno najpre definisati (matematički precizno) šta je to algoritam. U tome, bilo je dovoljno razmatrati algoritme za izračunavanje funkcija čiji su i argumenti i rezultujuće vrednosti prirodni brojevi (a drugi matematički i nematematički algoritmi se, digitalizacijom, mogu svesti na taj slučaj). Formalno zasnovan pojam algoritma omogućio je da kasnije budu identifikovani problemi za koje postoje algoritmi koji ih rešavaju, kao i problemi za koje ne postoje algoritmi koji ih rešavaju.

Pitanjem formalizacije pojma algoritma 1920-ih i 1930-ih godina nezavisno se bavilo nekoliko istaknutih matematičara i uvedeno je nekoliko raznorodnih formalizama, tj. nekoliko raznorodnih sistema izračunavanja. Najznačajniji među njima su:

- *Tjuringove mašine* (Tjuring),
- *rekurzivne funkcije* (Gedel[3] i Klini[4]),
- λ*-račun* (Čerč[5]),
- *Postove mašine* (Post[6]),
- *Markovljevi algoritmi* (Markov[7]),
- *mašine sa beskonačno mnogo registara* (engl. *Unlimited Register Machines* — URM)[8].

[1] Reč „algoritam“ ima koren u imenu persijskog astronoma i matematičara Al-Horezmija (engl. Muhammad ibn Musa al-Khwarizmi). On je 825. godine napisao knjigu, u međuvremenu nesačuvanu u originalu, verovatno pod naslovom „O računanju sa indijskim brojevima“. Ona je u dvanaestom veku prevedena na latinski, bez naslova, ali se na nju obično pozivalo njenim početnim rečima „Algoritmi de numero Indorum“, što je trebalo da znači „Al-Horezmi o indijskim brojevima“ (pri čemu je ime autora latinizovano u „Algoritmi“). Međutim, većina čitalaca je reč „Algoritmi“ shvatala kao množinu od nove, nepoznate reči „algoritam“ koja se vremenom odomaćila sa značenjem „metod za izračunavanje“.

[2] Ovaj problem, poznat pod imenom „Entscheidungsproblem“, postavio je David Hilbert 1928. godine. Poznato je da je još Gotfrid Lajbnic u XVII veku, nakon što je napravio mašinu za računanje, verovao da će biti moguće napraviti mašinski postupak koji će, manipulisanjem simbolima, biti u stanju da dâ odgovor na sva matematička pitanja. Ipak, 1930-ih, rezultatima Čerča, Tjuringa i Gedela pokazano je da ovakav postupak ne može da postoji.

[3] Kurt Gödel (1906-1978), austrijsko-američki matematičar.

[4] Stephen Kleene (1909-1994), američki matematičar.

[5] Alonzo Church (1903-1995), američki matematičar.

[6] Emil L. Post (1897-1954), američki matematičar.

[7] Andrej Andrejevič Markov (1856-1922), ruski matematičar.

[8] Postoji više srodnih formalizama koji se nazivaju URM. Prvi opisi mogu da se nađu u radovima Šeperdsona i Sturdžisa (engl. Sheperdson and Sturgis), dok je opis koji će biti dat u nastavku preuzet od Katlanda (engl. Nigel Cutland).

Navedeni sistemi nastali su pre savremenih računara i većina njih podrazumeva određene apstraktne mašine. I u to vreme bilo je jasno da je opis postupka dovoljno precizan ukoliko je moguće njime instruisati neku mašinu koja će taj postupak izvršiti. U tom duhu, i *savremeni programski jezici* predstavljaju precizne opise algoritama i moguće ih je pridružiti gore navedenoj listi. Značajna razlika je u tome što nabrojani formalizmi teže da budu što jednostavniji tj. da koriste što manji broj operacija i što jednostavnije modele mašina, dok savremeni programski jezici teže da budu što udobniji za programiranje te uključuju veliki broj operacija (koje, sa teorijskog stanovišta, nisu neophodne jer se mogu definisati preko malog broja osnovnih operacija).

Blok dijagrami (kaže se i *algoritamske šeme*, *dijagrami toka*, tj. *tokovnici*) mogu se smatrati poluformalnim načinom opisa algoritama. Oni neće biti razmatrani u teorijskom kontekstu ali će biti korišćeni u rešenjima nekih zadataka, radi lakšeg razumevanja.

Ako se neka funkcija može izračunati u nekom od navedenih formalizama (tj. ako se u tom formalizmu za sve moguće argumente mogu izračunati vrednosti funkcije), onda kažemo da je ona *izračunljiva* u tom formalizmu. Za sistem izračunavanja koji je dovoljno bogat da može da simulira Tjuringovu mašinu i tako da izvrši sva izračunavanja koja može da izvrši Tjuringova mašina kaže se da je *Tjuring potpun* (engl. *Turing complete*). Za sistem izračunavanja koji izračunava identičnu klasu funkcija kao Tjuringova mašina kažemo da je *Tjuring ekvivalentan* (engl. *Turing equivalent*).

Iako su navedni formalizmi međusobno veoma različiti, može se dokazati da su klase izračunljivih funkcija identične za sve njih, tj. svi oni formalizuju isti pojam algoritma i izračunljivosti. Drugim rečima, svi navedeni formalizmi su Tjuring ekvivalentni. Zbog toga se, umesto pojmova poput, na primer, Tjuring-izračunljiva ili URM-izračunljiva funkcija (i sličnih) može koristiti samo termin *izračunljiva funkcija*.

Svi navedeni formalizmi za izračunavanje, podrazumevaju u nekom smislu, pojednostavljeno rečeno, beskonačnu raspoloživu memoriju. Zbog toga savremeni računari (koji raspolažu konačnom memorijom) opremljeni savremenim programskim jezicima nisu Tjuring potpuni, u strogom smislu. S druge starne, svako izračunavanje koje se može opisati na nekom modernom programskom jeziku, može se opisati i kao program za Tjuringovu mašinu ili neki drugi ekvivalentan formalizam.

3.2 Čerč-Tjuringova teza

Veoma važno pitanje je koliko formalne definicije algoritama uspevaju da pokriju naš intuitivni pojam algoritma, tj. da li je zaista moguće efektivno izvršiti sva izračunavanja definisana nekom od formalizacija izračunljivosti i, obratno, da li sva izračunavanja koja intuitivno umemo da izvršimo zaista mogu da budu opisana korišćenjem bilo kog od precizno definisanih formalizama izračunavanja. Intuitivno je jasno da je odgovor na prvo pitanje potvrdan (jer čovek može da simulira rad jednostavnih mašina za izračunavanje), ali oko drugog pitanja postoji doza rezerve. *Čerč-Tjuringova teza*[9] tvrdi da je odgovor na oba navedena pitanja potvrdan.

Čerč-Tjuringova teza: *Klasa intuitivno izračunljivih funkcija identična je sa klasom formalno izračunljivih funkcija.*

Ovo tvrđenje je hipoteza, a ne teorema i ne može biti formalno dokazano. Naime, ono govori o intuitivnom pojmu algoritma, čija svojstva ne mogu biti formalno, matematički ispitana. Ipak, u korist ove teze govori činjenica da je dokazano da su sve navedene formalne definicije algoritama međusobno ekvivalentne, kao i da do sada nije pronađen nijedan primer intuitivno, efektivno izvodivog postupka koji nije moguće formalizovati u okviru nabrojanih formalnih sistema izračunavanja.

3.3 UR mašine

U daljem tekstu, pojam izračunljivosti biće uveden i izučavan na bazi UR *mašina (*URM*)*. Iako su po skupu izračunljivih funkcija nabrojani formalizacije pojma algoritma jednake, oni se međusobno razlikuju po svom duhu, a UR mašina je verovatno najbliža savremenom stilu programiranja: UR mašina ima ograničen (mali) skup instrukcija, programe i memoriju.

Kao i drugi formalizmi izračunavanja, UR mašine su apstrakcije koje formalizuju pojam algoritma, predstavljajući matematičku idealizaciju računara. UR mašine su dizajnirane tako da koriste samo mali broj operacija. Očigledno je da nije neophodno da formalizam za izračunavanje kao osnovnu operaciju ima, na primer, stepenovanje prirodnih brojeva, jer se ono može svesti na množenje. Slično, nije neophodno da formalizam za izračunavanje kao primitivnu operaciju ima množenje prirodnih brojeva, jer se ono može svesti na sabiranje. Čak ni sabiranje nije neophodno, jer se može svesti na operaciju pronalaženja sledbenika (uvećavanja za vrednost

[9] Ovu tezu, svaki za svoj formalizam, formulisali su nezavisno Čerč i Tjuring.

1). Zaista, definicija prirodnih brojeva podrazumeva postojanje nule i operacije sledbenika[10] i sve operacije nad prirodnim brojevima se svode na te dve elementarne. Pored njih, UR mašine koriste još samo dve instrukcije. Sva ta četiri tipa instrukcija brojeve čitaju i upisuju ih na polja ili *registre* beskonačne trake koja služi kao prostor za čuvanje podataka, tj. kao memorija mašine. Registri trake označeni su sa $R_1, R_2, R_3, \ldots$. Svaki od njih u svakom trenutku sadrži neki prirodan broj. Stanje registara u nekom trenutku zovemo *konfiguracija*. Sadržaj k-tog registra (registra R_k) označava se sa r_k, kao što je to ilustrovano na sledećoj slici:

R_1	R_2	R_3	$\ldots$
r_1	r_2	r_3	$\ldots$

Spisak i kratak opis URM instrukcija (naredbi) dati su u tabeli 3.1.[11] U tabeli, na primer, $R_m \leftarrow 0$ označava da se vrednost 0 upisuje u registar R_m, a na primer $r_m := r_m + 1$ označava da se sadržaj registra R_m (vrednost r_m) uvećava za jedan i upisuje u registar R_m.

oznaka	naziv	efekat
$Z(m)$	nula–instrukcija	$R_m \leftarrow 0$ (tj. $r_m := 0$)
$S(m)$	instrukcija sledbenik	$R_m \leftarrow r_m + 1$ (tj. $r_m := r_m + 1$)
$T(m,n)$	instrukcija prenosa	$R_n \leftarrow r_m$ (tj. $r_n := r_m$)
$J(m,n,p)$	instrukcija skoka	ako je $r_m = r_n$, idi na p-tu; inače idi na sledeću instrukciju

Tabela 3.1: Tabela URM instrukcija

URM program P je konačan numerisan niz URM instrukcija. Instrukcije se izršavaju redom (počevši od prve), osim u slučaju instrukcije skoka. Izvršavanje programa se zaustavlja onda kada ne postoji instrukcija koju treba izvršiti (kada se dođe do kraja programa ili kada se naiđe na skok na instrukciju koja ne postoji u numerisanom nizu instrukcija).

Početnu konfiguraciju čini niz prirodnih brojeva $a_1, a_2, \ldots$ koji su upisani u registre $R_1, R_2, \ldots$. Ako je funkcija koju treba izračunati $f(x_1, x_2, \ldots, x_n)$, onda se podrazumeva da su vrednosti $x_1, x_2, \ldots, x_n$ redom smeštene u prvih n registara.[12] Podrazumeva se i da, na kraju rada programa, rezultat treba da bude smešten u prvi registar.

Ako URM program P za početnu konfiguraciju $a_1, a_2, \ldots, a_n$ ne staje sa radom, onda pišemo $P(a_1, a_2, \ldots, a_n) \uparrow$. Inače, ako program staje sa radom i u prvom registru je, kao rezultat, vrednost b, onda pišemo $P(a_1, a_2, \ldots, a_n) \downarrow b$.

Kažemo da URM program izračunava funkciju $f : \mathbb{N}^n \to \mathbb{N}$ ako za svaku n-torku argumenata $a_1, a_2, \ldots, a_n$ za koju je funkcija f definisana i važi $f(a_1, a_2, \ldots, a_n) = b$ istovremeno važi i $P(a_1, a_2, \ldots, a_n) \downarrow b$. Funkcija je URM-*izračunljiva* ako postoji URM program koji je izračunava.

Primer 3.1. Neka je funkcija f definisana na sledeći način: $f(x, y) = x + y$. Vrednost funkcije f može se izračunati za sve vrednosti argumenata x i y UR mašinom. Ideja algoritma za izračunavanje vrednosti $x + y$ je da se vrednosti x doda vrednost 1, y puta, jer važi:

$$x + y = x + \underbrace{1 + 1 + \cdots + 1}_{y}$$

Odgovarajući URM program podrazumeva sledeću početnu konfiguraciju:

R_1	R_2	R_3	$\ldots$
x	y	$\ldots$	$\ldots$

i sledeću konfiguraciju u toku rada programa:

R_1	R_2	R_3	$\ldots$
$x + k$	y	k	$\ldots$

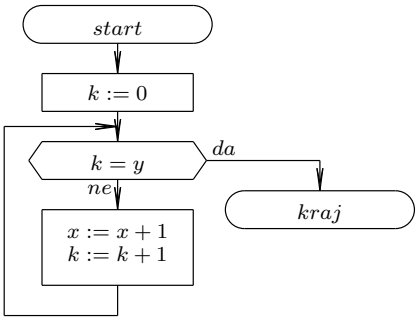
gde je $k \in \{0, 1, \ldots, y\}$.

Algoritam se može zapisati u vidu dijagrama toka i u vidu URM programa kao u nastavku:

[10] Skup prirodnih brojeva se definiše induktivno, kao najmanji skup koji sadrži nulu i zatvoren je u odnosu na operaciju pronalaženja sledbenika.

[11] Oznake URM instrukcija potiču od naziva ovih instrukcija na engleskom jeziku (*zero instruction, succesor instruction, transfer instruction* i *jump instruction*).

[12] Neki opisi UR mašina podrazumevaju da iza vrednosti x_1,x_2, …, x_n sledi niz nula, tj. da su svi registri (sem početnih koji sadrže argumente programa) inicijalizovani na nulu. U ovom tekstu se to neće podrazumevati, delom i zbog toga što promenljive u jeziku C nemaju nulu kao podrazumevanu vrednost.



1. $Z(3)$
2. $J(3,2,100)$
3. $S(1)$
4. $S(3)$
5. $J(1,1,2)$

Prekid rada programa realizovan je skokom na nepostojeću instrukciju 100[13]. Bezuslovni skok je realizovan naredbom oblika $J(1,1,\ldots)$ — poređenje registra sa samim sobom uvek garantuje jednakost te se skok vrši uvek.

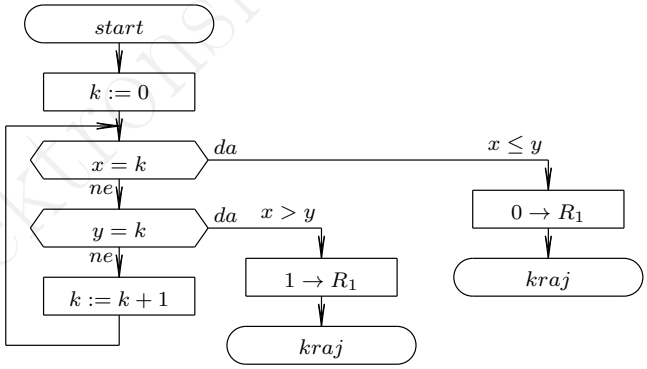
Primer 3.2. Neka je funkcija f definisana na sledeći način:

$$f(x,y) = \begin{cases} 0 & \text{, ako } x \leq y \\ 1 & \text{, inače} \end{cases}$$

URM program koji je računa koristi sledeću konfiguraciju u toku rada:

R_1	R_2	R_3	$\ldots$
x	y	k	$\ldots$

gde k dobija redom vrednosti $0, 1, 2\ldots$ sve dok ne dostigne vrednost x ili vrednost y. Prva dostignuta vrednost je broj ne veći od onog drugog. U skladu sa tim zaključkom i definicijom funkcije f, izračunata vrednost je 0 ili 1.



1. $Z(3)$ $k = 0$
2. $J(1,3,6)$ $x = k$?
3. $J(2,3,8)$ $y = k$?
4. $S(3)$ $k := k + 1$
5. $J(1,1,2)$
6. $Z(1)$ $0 \to R_1$
7. $J(1,1,100)$ kraj
8. $Z(1)$
9. $S(1)$ $1 \to R_1$

Primer 3.3. Neka je funkcija f definisana na sledeći način:

$$f(x) = [\sqrt{x}]$$

[13]Broj 100 je odabran proizvoljno kao broj sigurno veći od broja instrukcija u programu. I u programima koji slede, uniformnosti radi, za prekid programa će se takođe koristiti skok na instrukciju 100.

Izračunavanje vrednosti funkcije može se zasnivati na sledećoj osobini:

$$n = [\sqrt{x}] \Leftrightarrow n^2 \leq x < (n+1)^2$$

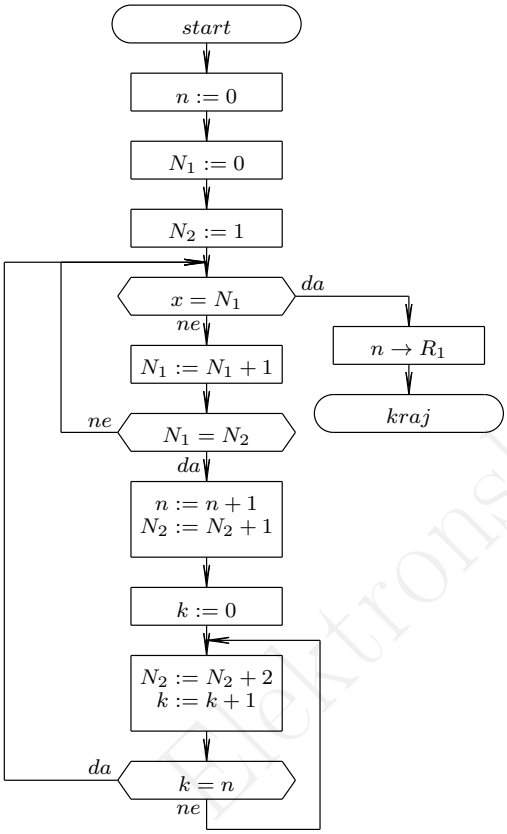
Odgovarajući URM program podrazumeva sledeću početnu konfiguraciju:

R_1	R_2	...
x	...	...

i sledeću konfiguraciju u toku svog rada:

R_1	R_2	R_3	R_4	R_5	...
x	n	$N_1 = n^2$	$N_2 = (n+1)^2$	k	...

Osnovna ideja algoritma je uvećavanje broja n za 1 i odgovarajućih vrednosti N_1 i N_2, sve dok se broj x ne nađe između njih. Nakon što se N_1 i N_2 postave na vrednosti kvadrata dva uzastopna broja n^2 i $(n+1)^2$, broj N_1 se postepeno uvećava za 1 i proverava se da li je jednak broju x. Ukoliko se ne naiđe na x, a N_1 dostigne vrednost N_2, onda se n uvećava za 1 i tada oba broja N_1 i N_2 imaju vrednost n^2 (naravno, za uvećano n). Tada je potrebno N_2 postaviti na vrednost $(n+1)^2$, tj. potrebno je uvećati N_2 za $(n+1)^2 - n^2 = 2n+1$. Ovo se postiže tako što se N_2 najpre uveća za 1, a zatim n puta uveća za 2. Nakon toga se ceo postupak ponavlja.



1.	$Z(2)$	$n := 0$
2.	$Z(3)$	$N_1 := 0$
3.	$Z(4)$	
4.	$S(4)$	$N_2 := 1$
5.	$J(1,3,17)$	$x = N_1?$
6.	$S(3)$	$N_1 := N_1 + 1$
7.	$J(3,4,9)$	$N_1 = N_2?$
8.	$J(1,1,5)$	
9.	$S(2)$	$n := n + 1$
10.	$S(4)$	$N_2 := N_2 + 1$
11.	$Z(5)$	$k := 0$
12.	$S(4)$	$N_2 := N_2 + 1$
13.	$S(4)$	$N_2 := N_2 + 1$
14.	$S(5)$	$k := k + 1$
15.	$J(5,2,5)$	$k = n?$
16.	$J(1,1,12)$	
17.	$T(2,1)$	$n \to R_1$

Primer 3.4. Neka je funkcija f definisana na sledeći način:

$$f(x) = \begin{cases} 0 & \text{, ako je } x = 0 \\ nedefinisano & \text{, inače} \end{cases}$$

Nedefinisanost funkcije f postiže se time što se program izvršava beskonačno izuzev ako je vrednost (prvog) argumenta jednaka 0:

1. $Z(2)$
2. $J(1,2,100)$
3. $J(1,1,2)$

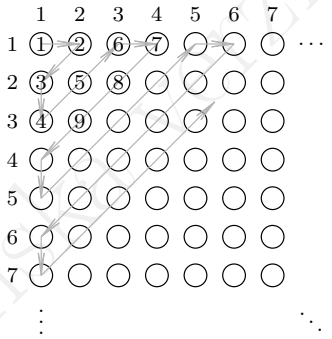
3.4 Enumeracija URM *programa*

Kardinalnost. Za dva skupa kaže se da *imaju istu kardinalnost* ako i samo ako je između njih moguće uspostaviti bijektivno preslikavanje. Za skupove koji imaju istu kardinalnost kao skup prirodnih brojeva kaže se da su *prebrojivi*. Dakle, neki skup je prebrojiv ako i samo ako je njegove elemente moguće poređati u niz (niz odgovara bijekciji sa skupom prirodnih brojeva). Za skupove koji su ili konačni ili prebrojivi, kaže se da su *najviše prebrojivi*. Georg Kantor[14] je prvi pokazao da skup realnih brojeva ima veću kardinalnost od skupa prirodnih brojeva, tj. da skup realnih brojeva nije prebrojiv.

Primer 3.5. Skupovi parnih i neparnih brojeva imaju istu kardinalnost jer je između njih moguće uspostaviti bijektivno preslikavanje $f(n) = n+1$. Ono što je na prvi pogled iznenađujuće je da su oba ova skupa prebrojiva, odnosno imaju istu kardinalnost kao i skup prirodnih brojeva, iako su njegovi pravi podskupovi (npr. $f(n) = 2 \cdot n$ uspostavlja bijekciju između skupa prirodnih i skupa parnih brojeva).

Lema 3.1. *Uređenih parova prirodnih brojeva ima prebrojivo mnogo.*

Dokaz: Razmotrimo beskonačnu tabelu elemenata koja odgovara svim uređenim parovima prirodnih brojeva. Moguće je izvršiti „cik-cak“ obilazak ove tabele, polazeći od gornjeg levog ugla kao što je to pokazano na narednoj slici.



Prilikom obilaska, svakom elementu je moguće dodeliti jedinstven redni broj, čime je uspostavljena tražena bijekcija sa skupom prirodnih brojeva. □

Korišćenjem sličnih ideja, mogu se dokazati i sledeća tvrđenja.

Lema 3.2. *Dekartov proizvod konačno mnogo prebrojivih skupova je prebrojiv.*

Lema 3.3. *Najviše prebrojiva unija prebrojivih skupova je prebrojiv skup.*

Enumeracija URM **programa.** Važno pitanje je koliko ima URM programa i koliko ima različitih URM programa (tj. onih koji izračunavaju različite funkcije). Očigledno je da ih ima beskonačno, a naredna tvrđenja govore da ih ima prebrojivo mnogo.

Lema 3.4. *Postoji prebrojivo mnogo različitih* URM *instrukcija.*

Dokaz: Instrukcija tipa Z, trivijalno, ima prebrojivo mnogo (Z instrukcije se mogu poređati u niz na sledeći način: $Z(1), Z(2), \ldots$). Slično je i sa instrukcijama tipa S. T instrukcije bijektivno odgovaraju parovima prirodnih brojeva, pa ih (na osnovu leme 3.2) ima prebrojivo mnogo. Slično, J instrukcije odgovaraju bijektivno trojkama prirodnih brojeva, pa ih (na osnovu leme 3.2) ima prebrojivo mnogo. Skup svih instrukcija je unija ova četiri prebrojiva skupa, pa je na osnovu leme 3.3 i ovaj skup prebrojiv. □

[14] Georg Cantor, 1845–1918, nemački matematičar.

Teorema 3.1. *Različitih* URM *programa ima prebrojivo mnogo*

Dokaz: Skup svih URM programa se može predstaviti kao unija skupa programa koji imaju jednu instrukciju, skupa programa koji imaju dve instrukcije i tako dalje. Svaki od ovih skupova je prebrojiv (na osnovu leme 3.2) kao konačan Dekartov stepen prebrojivog skupa instrukcija. Dakle, skup svih URM programa je prebrojiv (na osnovu leme 3.3) kao prebrojiva unija prebrojivih skupova. □

Na osnovu teoreme 3.1, moguće je uspostaviti bijekciju između skupa svih URM programa i skupa prirodnih brojeva. Drugim rečima, može se definisati pravilo koje svakom URM programu dodeljuje jedinstven prirodan broj i koje svakom prirodnom broju dodeljuje jedinstven URM program. Zbog toga, za fiksirano dodeljivanje brojeva programima, možemo da govorimo o prvom, drugom, trećem, ..., stotom URM programu itd.

3.5 Neizračunljivost i neodlučivost

Razmotrimo narednih nekoliko problema.

1. Neka su data dva konačna skupa reči. Pitanje je da li je moguće nadovezati nekoliko reči prvog skupa i, nezavisno, nekoliko reči drugog skupa tako da se dobije ista reč. Na primer, za skupove $\{a, ab, bba\}$ i $\{baa, aa, bb\}$, jedno rešenje je $bba \cdot ab \cdot bba \cdot a = bb \cdot aa \cdot bb \cdot baa$. Za skupove $\{ab, bba\}$ i $\{aa, bb\}$ rešenje ne postoji, jer se nadovezivanjem reči prvog skupa uvek dobija reč čija su poslednja dva slova različita, dok se nadovezivanjem reči drugog skupa uvek dobija reč čija su poslednja dva slova ista. Zadatak je konstruisati opšti algoritam koji za proizvoljna dva zadata skupa reči određuje da li tražena nadovezivanja postoje.[15]

2. Diofantska jednačina je jednačina oblika $p(x_1, \ldots, x_n) = 0$, gde je p polinom sa celobrojnim koeficijentima. Zadatak je konstruisati opšti algoritam kojim se određuje da li proizvoljna zadata diofantska jednačina ima racionalnih rešenja.[16]

3. Zadatak je konstruisati opšti algoritam koji proverava da li se proizvoljni zadati program P zaustavlja za date ulazne parametre.[17]

4. Zadatak je konstruisati opšti algoritam koji za proizvoljni zadati skup aksioma i zadato tvrđenje proverava da li je tvrđenje posledica aksioma.[18]

Za sva četiri navedena problema pokazano je da su *algoritamski nerešivi* ili *neodlučivi* (tj. ne postoji izračunljiva funkcija koja ih rešava). Ovo ne znači da nije moguće rešiti pojedine instance problema[19], već samo da ne postoji jedinstven, opšti postupak koji bi mogao da reši proizvoljnu instancu problema. Pored nabrojanih, ima još mnogo važnih neodlučivih problema.

U nastavku će precizno, korišćenjem formalizma UR mašina, biti opisan treći od navedenih problema, tj. *halting problem*, izuzetno važan za teorijsko računarstvo.

Problem ispitivanja zaustavljanja programa (halting problem). Pitanje zaustavljanja računarskih programa je jedno od najznačajnijih pitanja računarstva i programiranja. Često je veoma važno da li se neki program zaustavlja za neku ulaznu vrednost, da li se zaustavlja za ijednu ulaznu vrednost i slično. Za mnoge konkretne programe i za mnoge konkretne ulazne vrednosti, na ovo pitanje može se odgovoriti. No, nije očigledno da li postoji opšti postupak kojim se za proizvoljni dati program i proizvoljne vrednosti ulaznih argumenata može proveriti da li se program zaustavlja ako se pokrene sa tim argumentima.

Iako se problem ispitivanja zaustavljanja može razmatrati i za programe u savremenim programskim jezicima, u nastavku ćemo razmotriti formulaciju halting problema za URM programe:

Da li postoji URM *program koji na ulazu dobija drugi* URM *program P i neki broj x i ispituje da li se program P zaustavlja za ulazni parametar x?*

[15] Ovaj problem se naziva *Post's correspondence problem*, jer ga je postavio i rešio Emil Post 1946. godine.

[16] Ovaj problem je 10. Hilbertov problem izložen 1900. godine kao deo skupa problema koje „matematičari XIX veka ostavljaju u amanet matematičarima XX veka". Problem je rešio Matijaševič 1970-ih godina.

[17] Ovaj problem rešio je Alan Tjuring 1936. godine.

[18] Ovaj, već pomenut, problem, formulisao je David Hilbert 1928. godine, a nešto kasnije rešenje je proisteklo iz rezultata Čerča, Tjuringa i Gedela.

[19] Instanca ili *primerak problema* je jedan konkretan zadatak koji ima isti oblik kao i opšti problem. Na primer, za prvi u navedenom spisku problema, jedna instanca je zadatak ispitivanja da li je moguće nadovezati nekoliko reči skupa $\{ab, bba\}$ i, nezavisno, nekoliko reči skupa $\{aa, bb\}$ tako da se dobije ista reč.

Problem prethodne formulacije je činjenica da traženi URM program mora na ulazu da prihvati kao svoj argument drugi URM program, što je naizgled nemoguće, s obzirom na to da URM programi kao argumente mogu da imaju samo prirodne brojeve. Ipak, ovo se jednostavno razrešava zahvaljujući tome što je svakom URM programu P moguće dodeliti jedinstveni prirodan broj n koji ga identifikuje, i obratno, svakom broju n može se dodeliti program P_n (kao što je opisano u poglavlju 3.4). Imajući ovo u vidu, dolazi se do teoreme o halting problemu za URM.

Teorema 3.2 (Neodlučivost halting problema)**.** *Neka je funkcija h definisana na sledeći način:*

$$h(x,y) = \begin{cases} 1, & \text{ako se program } P_x \text{ zaustavlja za ulaz } y \\ 0, & \text{inače.} \end{cases}$$

Ne postoji program koji izračunava funkciju h, tj. ne postoji program koji za proizvoljne zadate vrednosti x i y može da proveri da li se program P_x zaustavlja za ulazni argument y.

Dokaz: Pretpostavimo da postoji program H koji izračunava funkciju h. Onda se jednostavno može konstruisati i program H' sa jednim argumentom x, koji vraća rezultat isti rezultat kao i program $H(x,x)$, tj. koji vraća rezultat 1 (tj. upisuje ga u prvi registar) ako se program P_x zaustavlja za ulaz x, a rezultat 0 ako se program P_x ne zaustavlja za ulaz x. Dalje, postoji i program Q (dobijen kombinovanjem programa H' i programa iz primera 3.4) koji za argument x vraća rezultat 0 ako se P_x ne zaustavlja za x (tj. ako je $h(x,x) = 0$), a izvršava beskonačnu petlju ako se P_x zaustavlja za x (tj. ako je $h(x,x) = 1$). Za program Q važi:

$$Q(x) \downarrow 0 \text{ ako je } P_x(x) \uparrow$$
$$Q(x) \uparrow \text{ ako je } P_x(x) \downarrow$$

Ako postoji takav program Q, onda se i on nalazi u nizu svih programa tj. postoji redni broj k koji ga jedinstveno identifikuje, pa važi:

$$P_k(x) \downarrow 0 \text{ ako je } P_x(x) \uparrow$$
$$P_k(x) \uparrow \text{ ako je } P_x(x) \downarrow$$

No, ako je x jednako upravo k, pokazuje se da je definicija ponašanja programa Q (tj. programa P_k) kontradiktorna: program Q (tj. program P_k) za ulaznu vrednost k vraća 0 ako se P_k ne zaustavlja za k, a izvršava beskonačnu petlju ako se P_k zaustavlja za k:

$$P_k(k) \downarrow 0 \text{ ako je } P_k(k) \uparrow$$
$$P_k(k) \uparrow \text{ ako je } P_k(k) \downarrow$$

Dakle, polazna pretpostavka je bila pogrešna i ne postoji program H, tj. funkcija h nije izračunljiva. Pošto funkcija h, karakteristična funkcija halting problema, nije izračunljiva, halting problem nije odlučiv. □

Funkcija koja odgovara halting problemu je jedna od najznačajnih funkcija iz skupa prirodnih brojeva u skup prirodnih brojeva koje ne mogu biti izračunate, ali takvih, neizračunljivih funkcija, ima još mnogo. Može se dokazati da funkcija jedne promenljive koje za ulazni prirodni broj vraćaju isključivo 0 ili 1 (tj. funkcija iz N u $\{0,1\}$) ima neprebrojivo mnogo, dok programa ima samo prebrojivo mnogo. Iz toga direktno sledi da ima neprebrojivo mnogo funkcija koje nisu izračunljive.

3.6 Vremenska i prostorna složenost izračunavanja

Prvo pitanje koje se postavlja kada je potrebno izračunati neku funkciju (tj. napisati neki program) je da li je ta funkcija izračunljiva (tj. da li uopšte postoji neki program koji je izračunava). Ukoliko takav program postoji, sledeće pitanje je koliko izvršavanje tog program zahteva vremena i prostora (memorije). U kontekstu URM programa, pitanje je koliko za neki URM program treba izvršiti pojedinačnih instrukcija (uključujući ponavljanja) i koliko registara se koristi. U URM programu navedenom u primeru 3.1 postoje ukupno četiri instrukcije, ali se one mogu ponavljati (za neke ulazne vrednosti). Jednostavno se može izračunati da se prva instrukcija u nizu izvršava $y+1$ puta, a preostale tri po y puta. Dakle, ukupan broj izvršenih instrukcija za ulazne vrednosti x i y je jednak $4y+1$. Ako se razmatra samo takozvani red algoritma, onda se zanemaruju konstante kojima

se množe i sabiraju vrednosti ulaznih argumenata, pa je vremenska složenost u ovom primeru linearna po drugom argumentu, argumentu y (i to se zapisuje $O(y)$). Bez obzira na vrednosti ulaznih argumenata, program koristi tri registra, pa je njegova prostorna složenost konstantna (i to se zapisuje $O(1)$). Najčešće se složenost algoritma određuje tako da ukazuje na to koliko on može utrošiti vremena i prostora u najgorem slučaju. Ponekad je moguće izračunati i prosečnu složenost algoritma — prosečnu za sve moguće vrednosti argumenata. O prostornoj i vremenskoj složenosti algoritama biće više reči u drugom tomu ove knjige.

Pitanja i zadaci za vežbu

Pitanje 3.1. *Po kome je termin* algoritam *dobio ime?*

Pitanje 3.2. *Šta je to algoritam (formalno i neformalno)? Navesti nekoliko formalizma za opisivanje algoritama. Kakva je veza između formalnog i neformalnog pojma algoritma. Šta tvrdi Čerč-Tjuringova teza? Da li se ona može dokazati?*

Pitanje 3.3. *Da li postoji algoritam koji opisuje neku funkciju iz skupa prirodnih brojeva u skup prirodnih brojeva i koji može da se isprogramira u programskom jeziku C i izvrši na savremenom računaru, a ne može na Tjuringovoj mašini?*

Pitanje 3.4. *Da li je svaka* URM *izračunljiva funkcija intuitivno izračunljiva? Da li je svaka intuitivno izračunljiva funkcija* URM *izračunljiva?*

Pitanje 3.5. *U čemu je ključna razlika između* URM *mašine i bilo kog stvarnog računara?*

Pitanje 3.6. *Opisati efekat* URM *naredbe* $J(m, n, p)$.

Pitanje 3.7. *Da li se nekim* URM *programom može izračunati hiljadita cifra broja* 2^{1000} *?*

Pitanje 3.8. *Da li postoji* URM *program koji izračunava broj* $\sqrt{2}$*? Da li postoji* URM *program koji izračunava* n*-tu decimalnu cifru broja* $\sqrt{2}$*, gde je* n *zadati prirodan broj?*

Pitanje 3.9. *Da li se nekim* URM *programom može izračunati hiljadita decimalna cifra broja* π *?*

Pitanje 3.10. *Koliko ima racionalnih brojeva? Koliko ima kompleksnih brojeva? Koliko ima različitih programa za Tjuringovu mašinu? Koliko ima različitih programa u programskom jeziku C?*

Pitanje 3.11. *Koliko elemenata ima unija konačno mnogo konačnih skupova? Koliko elemenata ima unija prebrojivo mnogo konačnih skupova? Koliko elemenata ima unija konačno mnogo prebrojivih skupova? Koliko elemenata ima unija prebrojivo mnogo prebrojivih skupova?*

Pitanje 3.12. *Koliko ima različitih* URM *programa? Kakva je kardinalnost skupa* URM *programa u odnosu na kardinalnost skupa prirodnih brojeva? Kakva je kardinalnost skupa* URM *programa u odnosu na kardinalnost skupa realnih brojeva? Kakva je kardinalnost skupa* URM *programa u odnosu na kardinalnost skupa programa na jeziku C?*

Pitanje 3.13. *Da li se svakom* URM *programu može pridružiti jedinstven prirodan broj (različit za svaki program)? Da li se svakom prirodnom broju može pridružiti jedinstven* URM *program (različit za svaki broj)?*

Pitanje 3.14. *Da li se svakom* URM *programu može pridružiti jedinstven realan broj (različit za svaki program)? Da li se svakom realnom broju može pridružiti jedinstven* URM *program (različit za svaki broj)?*

Pitanje 3.15. *Kako se naziva problem ispitivanja zaustavljanja programa? Kako glasi* halting *problem? Da li je on odlučiv ili nije? Ko je to dokazao?*

Pitanje 3.16.

1. *Da li postoji algoritam koji za drugi zadati* URM *program utvrđuje da li se zaustavlja ili ne?*
2. *Da li postoji algoritam koji za drugi zadati* URM *utvrđuje da li se zaustavlja posle 100 koraka?*
3. *Da li je moguće napisati* URM *program koji za drugi zadati* URM *program proverava da li radi beskonačno?*
4. *Da li je moguće napisati* URM *program koji za drugi zadati* URM *program proverava da li vraća vrednost 1?*
5. *Da li je moguće napisati* URM *program kojim se ispituje da li data izračunljiva funkcija (ona za koju postoji* URM *program)* f *zadovoljava da je* $f(0) = 0$*?*

6. *Da li je moguće napisati* URM *program koji za drugi zadati* URM *program ispituje da li izračunava vrednost* 2012 *i zašto?*

Pitanje 3.17. *Na primeru korena uporedite URM sa savremenim asemberlskim jezicima. Da li URM ima neke prednosti?*

Zadatak 3.1. *Napisati* URM *program koji izračunava funkciju* $f(x,y) = xy$. √

Zadatak 3.2. *Napisati* URM *program koji izračunava funkciju* $f(x) = 2^x$.

Zadatak 3.3. *Napisati* URM *program koji izračunava funkciju* $f(x,y) = x^y$.

Zadatak 3.4. *Napisati* URM *program koji izračunava funkciju:*

$$f(x,y) = \begin{cases} 1 & , \text{ako } x \geq y \\ 0 & , \text{inače} \end{cases}$$

Zadatak 3.5. *Napisati* URM *program koji izračunava funkciju*

$$f(x,y) = \begin{cases} x-y & , \text{ako } x \geq y \\ 0 & , \text{inače} \end{cases}$$

√

Zadatak 3.6. *Napisati* URM *program koji izračunava funkciju:*

$$f(x) = \begin{cases} x/3 & , \text{ako } 3|x \\ nedefinisano & , \text{inače} \end{cases}$$

Zadatak 3.7. *Napisati* URM *program koji izračunava funkciju* $f(x) = x!$.

Zadatak 3.8. *Napisati* URM *program koji izračunava funkciju* $f(x) = \left[\frac{2x}{3}\right]$.

Zadatak 3.9. *Napisati* URM *program koji broj* 1331 *smešta u prvi registar.*

Zadatak 3.10. *Napisati* URM *program koji izračunava funkciju* $f(x) = 1000 \cdot x$.

Zadatak 3.11. *Napisati* URM *program koji izračunava funkciju* $f(x,y) = 2x + y$.

Zadatak 3.12. *Napisati* URM *program koji izračunava funkciju* $f(x,y) = min(x,y)$, *odnosno:*

$$f(x,y) = \begin{cases} x & , \text{ako } x \leq y \\ y & , \text{inače} \end{cases}$$

Zadatak 3.13. *Napisati* URM *program koji izračunava funkciju* $f(x,y) = 2^{(x+y)}$

Zadatak 3.14. *Napisati* URM *program koji izračunava funkciju*

$$f(x,y) = \begin{cases} 1 & , \text{ako } x|y \\ 0 & , \text{inače} \end{cases}$$

Zadatak 3.15. *Napisati* URM *program koji izračunava funkciju*

$$f(x,y) = \begin{cases} \left[\frac{y}{x}\right] & , \text{ako } x \neq 0 \\ nedefinisano & , \text{inače} \end{cases}$$

Zadatak 3.16. *Napisati* URM *program koji izračunavaju sledeću funkciju:*

$$f(x,y) = \begin{cases} 2x & , x < y \\ x-y & , x \geq y \end{cases}$$

Zadatak 3.17. *Napisati* URM *program koji izračunava funkciju:*

$$f(x,y) = \begin{cases} x/3 & , 3|x \\ y^2 & , inace \end{cases}$$

Zadatak 3.18. *Napisati* URM *program koji izracunava funkciju* $f(x,y,z) = x + y + z$

Zadatak 3.19. *Napisati* URM *program koji izracunava funkciju* $f(x, y, z) = min(x, y, z)$.

Zadatak 3.20. *Napisati* URM *program koji izračunava funkciju*

$$f(x, y, z) = \begin{cases} \left[\frac{y}{3}\right] & , \text{ ako } 2|z \\ x + 1 & , \text{ inače} \end{cases}$$

Zadatak 3.21. *Napisati* URM *program koji izračunava funkciju*

$$f(x, y, z) = \begin{cases} 1, & \text{ako je } x + y > z \\ 2, & \text{inače} \end{cases}$$

Glava 4

Viši programski jezici

Razvoj programskih jezika, u bliskoj je vezi sa razvojem računara tj. sa razvojem hardvera. Programiranje u današnjem smislu nastalo je sa pojavom računara fon Nojmanovog tipa čiji se rad kontroliše programima koji su smešteni u memoriji, zajedno sa podacima nad kojim operišu. Na prvim računarima tog tipa moglo je da se programira samo na mašinski zavisnim programskim jezicima, a od polovine 1950-ih nastali su jezici višeg nivoa koji su drastično olakšali programiranje.

Prvi programski jezici zahtevali su od programera da bude upoznat sa najfinijim detaljima računara koji se programira. Problemi ovakvog načina programiranja su višestruki. Naime, ukoliko je želeo da programira na novom računaru, programer je morao da izuči sve detalje njegove arhitekture (na primer, skup instrukcija procesora, broj registara, organizaciju memorije). Programi napisani za jedan računar mogli su da se izvršavaju isključivo na istim takvim računarima i prenosivost programa nije bila moguća.

Viši programski jezici namenjeni su ljudima a ne mašinama i sakrivaju detalje konkretnih računara od programera. Specijalizovani programi (tzv. *jezički procesori*, *programski prevodioci*, *kompilatori* ili *interpretatori*) na osnovu specifikacije zadate na višem (apstraktnijem) nivou mogu automatski da proizvedu mašinski kôd za specifičan računar na kojem se programi izvršavaju. Ovim se omogućava prenosivost programa (pri čemu, da bi se program mogao izvršavati na nekom konkretnom računaru, potrebno je da postoji procesor višeg programskog jezika baš za taj računar). Takođe, znatno se povećava nivo apstrakcije prilikom procesa programiranja što u mnogome olakšava ovaj proces.

Razvojni ciklus programa u većini savremenih viših programskih jezika teče na sledeći način. Danas je, nakon faze planiranja, prva faza u razvoju programa njegovo *pisanje* tj. unošenje programa na višem programskom jeziku (tzv. *izvorni program* ili *izvorni kôd* – engl. *source code*), što se radi pomoću nekog editora teksta. Naredna faza je njegovo *prevođenje*, kada se na osnovu izvornog programa na višem programskom jeziku dobija prevedeni kôd na asemblerskom odnosno mašinskom jeziku (tzv. *objektni kôd* – engl. *object code*), što se radi pomoću nekog programskog prevodioca. U fazi *povezivanja* više objektnih programa povezuje se sa objektnim kodom iz standardne biblioteke u jedinstvenu celinu (tzv. izvršivi program – engl. executable program). Povezivanje vrši specijalizovan program *povezivač*, tj. *linker* (engl. *linker*) ili *uređivač veza*. Nakon povezivanja, kreiran je program u *izvršivom obliku* i on može da se *izvršava*. Nabrojane faze se obično ponavljaju, vrši se dopuna programa, ispravljanje grešaka, itd.

4.1 Kratki pregled istorije programskih jezika

Prvim višim programskim jezikom najčešće se smatra jezik *FORTRAN*, nastao u periodu 1953-1957 godine. Njegov glavni autor je Džon Bakus[1], koji je implementirao i prvi interpretator i prvi kompilator za ovaj jezik. Ime FORTRAN dolazi od *The IBM Mathematical FORmula TRANslating System*, što ukazuje na to da je osnovna motivacija bila da se u okviru naučno-tehničkih primena omogući unošenje matematičkih formula, dok je sistem taj koji bi unete matematičke formule prevodio u niz instrukcija koje računar može da izvršava. Neposredno nakon pojave Fortrana, Džon Mekarti[2] je dizajnirao programski jezik *LISP* (*LISt Processing*), zasnovan na λ-računu. LISP je uveo funkcionalno programiranje i dugo bio najpopularniji jezik u oblasti veštačke inteligencije. Krajem 1950-ih godina, nastao je i jezik COBOL (*COmmon Business-Oriented Language*), čije su osnovne primene u oblasti poslovanja. Zanimljivo je da puno starih COBOL programa i danas uspešno radi u velikim poslovnim sistemima kod nas i u svetu.

[1] John Backus (1924–2007), američki naučnik iz oblasti računarstva, dobitnik Tjuringove nagrade.

[2] John McCarthy (1927–2011), američki naučnik iz oblasti računarstva. Dobitnik Tjuringove nagrade za svoj rad na polju veštačke inteligencije.

Rani programski jezici, nastali pod uticajem asemblerskih jezika, intenzivno koriste skokove u programima (tzv. GOTO naredbu) što dovodi do programa koje je teško razumeti i održavati. Kao odgovor na softversku krizu 1970-ih godina (period kada zbog loše prakse programiranja softver nije mogao da dostigne mogućnosti hardvera), nastala je praksa *strukturnog programiranja* u kojoj se insistira na disciplinovanom pristupu programiranju, bez nekontrolisanih skokova i uz korišćenje samo malog broja naredbi za kontrolu toka programa.

Krajem 1950-ih godina započet je razvoj programskog jezika *ALGOL 60* koji je uneo mnoge koncepte prisutne skoro u svim današnjim programskim jezicima. Tokom 1970-ih pojavio se jezik *C* koji i dalje predstavlja osnovni jezik sistemskog programiranja. Tokom 1970-ih pojavio se i jezik *Pascal* koji je vrlo elegantan jezik čiji je cilj bio da ohrabri strukturno programiranje i koji se zbog ovoga (u svojim unapređenim oblicima) i danas ponegde koristi u nastavi programiranja.

Kao odgovor na još jednu softversku krizu prelazi se na korišćenje tzv. objektno-orijentisanih jezika koji olakšavaju izradu velikih programa i podelu posla u velikim programerskim timovima. Tokom 1980-ih se pojavljuje jezik *C++* koji nadograđuje jezik C objektno-orijentisanim konceptima, a tokom 1990-ih, pod uticajem interneta, i jezik *Java*, čija je jedna od osnovnih ideja prenosivost izvršivog koda između heterogenih računarskih sistema. Microsoft, je krajem 1990-ih započeo razvoj jezika *C#* koji se danas često koristi za programiranje Windows aplikacija.

Pojavom veba postaju značajni skript jezici, kao što su *PHP*, *JavaScript*, *Python*, *Ruby* itd.

4.2 Klasifikacije programskih jezika

Po načinu programiranja, programski jezici se klasifikuju u *programske paradigme*.

Najkorišćeniji programski jezici danas spadaju u grupu *imperativnih* programskih jezika. Kako u ovu grupu spada i programski jezik C (ali i programski jezik Pascal, Fortran, Basic itd.), u nastavku će biti najviše reči upravo o ovakvim jezicima. U ovim jezicima stanje programa karakterišu *promenljive* kojima se predstavljaju podaci i *naredbe* kojima se vrše određene transformacije promenljivih. Pored imperativne, značajne programske paradigme su i *objektno-orijentisana* (u nju spadaju C++, Java, C# itd.), *funkcionalna* (u nju spadaju Lisp, Haskell, ML, itd.), *logička* (u nju spada, na primer, Prolog). U savremenim jezicima mešaju se karakteristike različitih programskih paradigmi tako da je podela sve manje striktna.

Većina programskih jezika danas je *proceduralna* što znači da je zadatak programera da opiše način (proceduru) kojim se dolazi do rešenja problema. Nasuprot njima, *deklarativni* programski jezici (na primer Prolog) od programera zahtevaju da precizno opiše problem, dok se mehanizam programskog jezika onda bavi pronalaženjem rešenja problema. Deklarativni jezici nisu među dominantnim jezicima opšte namene, ali se uspešno koriste u mnogim ograničenim domenima.

4.3 Leksika, sintaksa, semantika programskih jezika

Da bi bilo moguće pisanje i prevođenje programa u odgovarajuće programe na mašinskom jeziku nekog konkretnog računara, neophodno je precizno definisati šta su ispravni programi nekog programskog jezika, kao i precizno definisati koja izračunavanja odgovaraju naredbama programskog jezika. Pitanjima ispravnosti programa bavi se *sintaksa programskih jezika* (i njena podoblast *leksika programskih jezika*). Leksika se bavi opisivanjem osnovnih gradivnih elemenata jezika, a sintaksa načinima za kombinovanje tih osnovnih elemenata u ispravne jezičke konstrukcije. Pitanjem značenja programa bavi se *semantika programskih jezika*. Leksika, sintaksa i semantika se izučavaju ne samo za programske jezike, već i za druge veštačke jezike, ali i za prirodne jezike.

Leksika. Osnovni leksički elementi prirodnih jezika su reči, pri čemu se razlikuje nekoliko različitih vrsta reči (imenice, glagoli, pridevi, ...) i reči imaju različite oblike (padeži, vremena, ...). Zadatak leksičke analize prirodnog jezika je da identifikuje reči u rečenici i svrsta ih u odgovarajuće kategorije. Slično važi i za programske jezike. Programi se računaru zadaju predstavljeni nizom karaktera. Pojedinačni karakteri se grupišu u nedeljive celine koje predstavljaju osnovne leksičke elemente, koji bi bili analogni rečima govornog jezika. Na primer, leksika jezika UR mašina razlikuje rezervisane reči (Z, S, J, T) i brojevne konstante. Razmotrimo naredni fragment koda u jeziku C:

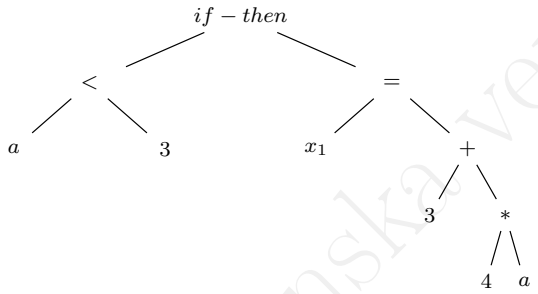
```
if (a < 3)
   x1 = 3+4*a;
```

U ovom kodu, razlikuju se sledeće *lekseme* (reči) i njima pridruženi *tokeni* (kategorije).

`if`	ključna reč
`(`	zagrada
`a`	identifikator
`<`	operator
`3`	celobrojni literal
`)`	zagrada
`x1`	identifikator
`=`	operator
`3`	celobrojni literal
`+`	operator
`4`	celobrojni literal
`*`	operator
`a`	celobrojni literal
`;`	interpunkcija

Leksikom programa obično se bavi deo programskog prevodioca koji se naziva *leksički analizator*.

Sintaksa. Sintaksa prirodnih jezika definiše načine na koji pojedinačne reči mogu da kreiraju ispravne rečenice jezika. Slično je i sa programskim jezicima, gde se umesto ispravnih rečenica razmatraju ispravni programi. Na primer, sintaksa jezika UR mašina definiše ispravne programe kao nizove instrukcija oblika: $Z(\text{broj})$, $S(\text{broj})$, $J(\text{broj}, \text{broj}, \text{broj})$ i $T(\text{broj}, \text{broj})$. Sintaksa definiše formalne relacije između elemenata jezika, time pružajući strukturne opise ispravnih niski jezika. Sintaksa se bavi samo formom i strukturom jezika bez bilo kakvih razmatranja u vezi sa njihovim značenjem. Sintaksička struktura rečenica ili programa se može predstaviti u obliku stabla. Prikazani fragment koda je u jeziku C sintaksički ispravan i njegova sintaksička struktura se može predstaviti na sledeći način:



Dakle, taj fragment koda predstavlja `if-then` naredbu (iskaz) kojoj je uslov dat izrazom poređenja vrednosti promenljive `a` i konstante `3`, a telo predstavlja naredba dodele promenljivoj `x` vrednosti izraza koji predstavlja zbir konstante `3` i proizvoda konstante `4` i vrednosti promenljive `a`.

Sintaksom programa obično se bavi deo programskog prevodioca koji se naziva *sintaksički analizator*.

Semantika. Semantika pridružuje značenje sintaksički ispravnim niskama jezika. Za prirodne jezike, semantika pridružuje ispravnim rečenicama neke specifične objekte, misli i osećanja. Za programske jezike, semantika za dati program opisuje koje je izračunavanje opisano tim programom.

Tako se, na primer, naredbi `if (a < 3) x1 = 3+4*a;` jezika C može pridružiti sledeće značenje: „Ako je tekuća vrednost promenljive `a` manja od 3, tada promenljiva `x1` treba da dobije vrednost zbira broja 3 i četvorostruke tekuće vrednosti promenljive `a`“.

Postoje sintaksički ispravne rečenice prirodnog jezika kojima nije moguće dodeliti ispravno značenje, tj. za njih nije definisana semantika, na primer: „Bezbojne zelene ideje besno spavaju“ ili „Pera je oženjeni neženja“. Slično je i sa programskim jezicima. Neki aspekti semantičke korektnosti programa se mogu proveriti tokom prevođenja programa (na primer, da su sve promenljive koje se koriste u izrazima definisane i da su odgovarajućeg tipa), dok se neki aspekti mogu proveriti tek u fazi izvršavanja programa (na primer, da ne dolazi do deljenja nulom). Na osnovu toga, razlikuje se statička i dinamička semantika. Naredni C kôd nema jasno definisano dinamičko značenje, pa u fazi izvršavanja dolazi do greške (iako se prevođenje na mašinski jezik uspešno izvršava).

```
int x = 0;
int y = 1/x;
```

Dok većina savremenih jezika ima precizno i formalno definisanu leksiku i sintaksu, formalna definicija semantike postoji samo za neke programske jezike[3]. U ostalim slučajevima, semantika programskog jezika se

[3]Leksika se obično opisuje *regularnim izrazima*, sintaksa *kontekstno slobodnim gramatikama*, dok se semantika formalno zadaje ili aksiomatski (npr. u obliku Horove logike) ili u vidu denotacione semantike ili u vidu operacione semantike.

opisuje neformalno, opisima zadatim korišćenjem prirodnog jezika. Čest je slučaj da neki aspekti semantike ostaju nedefinisani standardom jezika i prepušta se implementacijama kompilatora da samostalno odrede potpunu semantiku. Tako, na primer, programski jezik C ne definiše kojim se redom vrši izračunavanje vrednosti izraza, što u nekim slučajevima može da dovede do različitih rezultata istog programa prilikom prevođenja i izvršavanja na različitim sistemima (na primer, nije definisano da li se za izračunavanje vrednosti izraza `f() + g()` najpre poziva funkcija `f` ili funkcija `g`).

4.4 Pragmatika programskih jezika

Pragmatika jezika govori o izražajnosti jezika i o odnosu različitih načina za iskazivanje istih stvari. Pragmatika prirodnih jezika se odnosi na psihološke i sociološke aspekte kao što su korisnost, opseg primena i efekti na korisnike. Isti prirodni jezik se koristi drugačije, na primer, u pisanju tehničkih uputstava, a drugačije u pisanju pesama. Pragmatika programskih jezika uključuje pitanja kao što su lakoća programiranja, efikasnost u primenama i metodologija programiranja. Pragmatika je ključni predmet interesovanja onih koji dizajniraju i implementiraju programske jezike, ali i svih koji ih koriste. U pitanja pragmatike može da spada i izbor načina na koji napisati neki program, u zavisnosti od toga da li je, u konkretnom slučaju, važno da je program kratak ili da je jednostavan za razumevanje ili da je efikasan.

Pragmatika jezika se, između ostalog, bavi i sledećim pitanjima dizajna programskih jezika.

Promenljive i tipovi podataka. U fon Nojmanovom modelu, podaci se smeštaju u memoriju računara (najčešće predstavljeni nizom bitova). Međutim, programski jezici uobičajeno, kroz koncept *promenljivih*, nude programerima apstraktniji pogled na podatke. Promenljive omogućavaju programeru da imenuje podatke i da im pristupa na osnovu imena, a ne na osnovu memorijskih adresa (kao što je to slučaj kod asemblerskih jezika). Svakoj promenljivoj dodeljen je određen broj bajtova u memoriji (ili, eventualno, u registrima procesora) kojima se predstavljaju odgovarajući podaci. Pravila *životnog veka* (engl. *lifetime*) određuju u kom delu faze izvršavanja programa je promenljivoj dodeljen memorijski prostor (promenljivu je moguće koristiti samo tada). Nekada je moguće na različitim mestima u programu koristiti različite promenljive istog imena i pravila *dosega indentifikatora* (engl. *scope*) određuju deo programa u kome se uvedeno ime može koristiti za imenovanje određenog objekta (najčešće promenljive).

Organizovanje podataka u *tipove* omogućava da programer ne mora da razmišlja o podacima na nivou njihove interne (binarne) reprezentacije, već da o podacima može razmišljati znatno apstraktnije, dok se detalji interne reprezentacije prepuštaju jezičkom procesoru. Najčešći tipovi podataka direktno podržani programskim jezicima su celi brojevi (0, 1, 2, -1, -2, ...), brojevi u pokretnom zarezu (1.0, 3.14, ...), karakteri (`a`, `b`, `0`, `,`, `!`, ...), niske (`"zdravo"`), ... Pored ovoga, programski jezici obično nude i mogućnost korišćenja složenih tipova (na primer, nizovi, strukture tj. slogovi koji mogu da objedinjavaju nekoliko promenjivih istog ili različitog tipa).

Svaki tip podataka karakteriše:

- vrsta podataka koje opisuje (na primer, celi brojevi),
- skup operacija koje se mogu primeniti nad podacima tog tipa (na primer, sabiranje, oduzimanje, množenje, ...),
- način reprezentacije i detalji implementacije (na primer, zapis u obliku binarnog potpunog komplementa širine 8 bitova, odakle sledi opseg vrednosti od -128 do 127).

Naredni fragment C koda obezbeđuje da su promenljive `x`, `y` i `z` celobrojnog tipa (tipa `int`), a da `z` nakon izvršavanja ovog fragmenta ima vrednost jednaku zbiru promenljivih `x` i `y`.

```
int x, y;
...
int z = x + y;
```

Prilikom prevođenja i kreiranja mašinskog koda, prevodilac, vođen tipom promenljivih `x`, `y` i `z`, dodeljuje određeni broj bitova u memoriji ovim promenljivim (tj. rezerviše određene memorijske lokacije isključivo za smeštanje vrednosti ovih promenljivih) i generiše kôd (procesorske instrukcije) kojim se tokom izvršavanja programa sabiraju vrednosti smeštene na dodeljenim memorijskim lokacijama. Pri tom se vodi računa o načinu reprezentacije koji se koristi za zapis. S obzirom da je naglašeno da su promenljive `x`, `y` i `z`, celobrojnog tipa, prevodilac će najverovatnije podrazumevati zapis u potpunom komplementu i operacija `+` će se prevesti u mašinsku instrukciju kojom se sabiraju brojevi zapisani u potpunom komplementu (koja je obično direktno podržana u svakom procesoru). U zavisnosti od tipa operanada, ista operacija se ponekad prevodi na različite načine. Tako, da su

promenljive bile tipa `float`, najverovatnije bi se podrazumevao zapis u obliku pokretnog zareza i operacija `+` bi se prevela u mašinsku instrukciju kojom se sabiraju brojevi zapisani u pokretnom zarezu. Time, tipovi podataka iz jezika višeg nivoa u odgovarajućem mašinskom kodu postoje samo implicitno, najčešće samo kroz mašinske instrukcije nastale nakon prevođenja.

Tokom prevođenja programa se razrešavaju i pitanja *konverzija tipova*. Na primer, u C kodu `int x = 3.1;` promenljiva `x` je deklarisana kao celobrojna, pa se za nju odvaja određeni broj bitova u memoriji i ta memorija se inicijalizuje binarnim zapisom broja `3`. Naime, realnu vrednost `3.1` nije moguće zapisati kao celobrojnu promenljivu, pa se vrši konverzija tipa i zaokruživanje vrednosti.

Statički tipizirani jezici (uključujući C), zahtevaju da programer definiše tip svake promenljive koja se koristi u programu i da se taj tip ne menja tokom izvršavanja programa. Međutim, neki statički tipizirani programski jezici (na primer Haskell ili ML) ne zahtevaju od programera definisanje tipova, već se tipovi automatski određuju iz teksta programa. S druge strane, u *dinamički tipiziranim jezicima* ista promenljiva može da sadrži podatke različitog tipa tokom raznih faza izvršavanja programa. U nekim slučajevima dopušteno je vršiti operacije nad promenljivima različitog tipa, pri čemu se tip implicitno konvertuje. Na primer, jezik JavaScript ne zahteva definisanje tipa promenljivih i dopušta kôd poput `a = 1; b = "2"; a = a + b;`.

Kontrola toka izvršavanja. Osnovi gradivni element imperativnih programa su *naredbe*. Osnovna naredba je *naredba dodele* kojom se vrednost neke promenljive postavlja na vrednost nekog *izraza* definisanog nad konstantama i definisanim promenljivim. Na primer, `x1 = 3 + 4*a;`.

Naredbe se u programu često nižu i izvršavaju sekvencijalno, jedna nakon druge. Međutim, da bi se postigla veća izražajnost, programski jezici uvode specijalne vrste naredbi kojima se vrši kontrola toka izvršavanja programa (tzv. kontrolne strukture). Ovim se postiže da se u zavisnosti od tekućih vrednosti promenljivih neke naredbe uopšte ne izvršavaju, neke izvršavaju više puta i slično. Najčešće korišćene kontrolne strukture su granajuće naredbe (`if-then-else`), petlje (`for`, `while`, `do-while`, `repeat-until`) i naredbe skoka (`goto`). Za naredbu skoka (`goto`) je pokazano da nije neophodna, tj. svaki program se može zameniti programom koji daje iste rezultate a pri tome koristi samo sekvencijalno nizanje naredbi, naredbu izbora (`if-then-else`) i jednu vrstu petlje (na primer, `do-while`).[4] Ovaj važan teorijski rezultat ima svoju punu praktičnu primenu i u današnjem, strukturnom, programiranju naredba skoka se gotovo uopšte ne koristi jer dovodi do nerazumljivih (tzv. špageti) programa.

Potprogrami. Većina programskih jezika pruža mogućnost definisanja neke vrste *potprograma*, ali nazivi potprograma se razlikuju (najčešće se koriste termini *funkcije*, *procedure*, *sabrutine* ili *metode*). Potprogrami izoluju određena izračunavanja koja se kasnije mogu pozivati tj. koristiti na više različitih mesta u različitim kontekstima. Tako je, na primer, u jeziku C moguće definisati funkciju kojom se izračunava najveći zajednički delilac (NZD) dva broja, a kasnije tu funkciju iskoristiti na nekoliko mesta da bi se izračunao NZD različitih parova brojeva.

```
int nzd(int a, int b) { ... }
...
x = nzd(1234, 5678);
y = nzd(8765, 4321);
```

Primetimo da su potprogrami obično parametrizovani tj. mogu da primaju ulazne *parametre* (kaže se i *argumente*). Postoje različiti načini prenosa parametara u potprograme. Na primer, u nekim slučajevima (tzv. *prenos po vrednosti*) funkcija dobija svoju kopiju parametra navedenog u pozivu i sve vreme barata kopijom, ostavljajući originalni parametar nepromenjen. U nekim slučajevima (tzv. *prenos po adresi*), parametar se ne kopira već se u funkciju prenosi samo memorijska adresa na kojoj se parametar nalazi i funkcija sve vreme barata originalnim parametrom.

Potprogrami imaju i mogućnost vraćanja vrednosti izračunavanja pozivaocu. Neki jezici (npr. Pascal) suštinski razlikuju funkcije koje izračunavaju (i kaže se *vraćaju*) neku vrednost i procedure čiji je zadatak samo da proizvedu određeni sporedni efekat (npr. da ispišu nešto na ekran ili da promene vrednost promenljive).

Modularnost. Modularnost podrazumeva razbijanje većeg programa na nezavisne celine. Celine, koje sadrže definicije srodnih podataka i funkcija, obično se nazivaju *biblioteke* (engl. *library*) i smeštaju se u posebne datoteke. Ovim se postiže lakše održavanje kompleksnih sistema, kao i mogućnost višestruke upotrebe pojedinih modula u okviru različitih programa. Celine se obično zasebno prevode i kasnije povezuju u jedinstven program.

[4] Ovo tvrđenje je poznato kao *teorema o strukturnom programiranju*, Korado Bema (nem. Corrado Böhm) i Đuzepea Jakopinija (it. Giuseppe Jacopini) iz 1966.

Programski jezici obično imaju svoje *standardne biblioteke* (engl. *standard library*) koje sadrže funkcije često potrebne programeru (npr. funkciju za izračunavanje dužine niske karaktera). Često se kôd funkcija standardne biblioteke statički povezuju sa programom (tj. uključuje se u izvršivi kôd programa).

Osim standardne biblioteke, programske jezike često karakteriše i *rantajm biblioteka* (engl. *runtime library*) koja predstavlja sponu između kompilatora (tj. izvršivih programa koje on generiše) i operativnog sistema. Funkcije rantajm biblioteke se ne uključuju u izvršivi kôd programa već se dinamički pozivaju tokom izvršavanja programa (i zato, da bi program mogao da se izvrši na nekom operativnom sistemu, neophodno je da na njemu bude instalirana rantajm biblioteka za taj programski jezik).

Upravljanje memorijom. Neki programski jezici (na primer C, C++) zahtevaju od programera da eksplicitno rukuje memorijom tj. da od sistema zahteva memoriju u trenutku kada je ona potrebna i da tu memoriju eksplicitno oslobađa, tj. vraća sistemu kada ona programu više nije potrebna. Drugi programski jezici (na primer, Java, C#, Haskell, ML) oslobađaju programera ove dužnosti time što koriste tzv. *sakupljače otpada* (engl. *garbage collector*) čiji je zadatak da detektuju memoriju koju program ne koristi i da je oslobađaju. Iako je programiranje u jezicima sa sakljupljačima otpada jednostavnije, programi su obično sporiji (jer određeno vreme odlazi na rad sakupljača otpada).

U cilju obezbeđivanja pogodnog načina da se hardverom upravlja, neki programski jezici dopuštaju programeru da pristupi proizvoljnoj memorijskoj adresi (npr. u jeziku C, to se može raditi korišćenjem pokazivača), čime se dobija veća sloboda, ali i mnogo veća mogućnost pravljenja grešaka. Sa druge strane, neki programski jezici imaju za cilj skrivanje svojstava hardvera od programera, štite memoriju od direktnog pristupa programera i dopuštaju korišćenje podataka samo u okviru memorije zauzete promenljivim programa.

4.4.1 Jezički procesori

Jezički procesori (ili programski prevodioci) su programi čija je uloga da analiziraju leksičku, sintaksičku i (donekle) semantičku ispravnost programa višeg programskog jezika i da na osnovu ispravnog ulaznog programa višeg programskog jezika generišu kôd na mašinskom jeziku (koji odgovara polaznom programu, tj. vrši izračunavanje koje je opisano na višem programskom jeziku). Da bi konstrukcija jezičkih procesora uopšte bila moguća, potrebno je imati precizan opis leksike i sintakse, ali i što precizniji opis semantike višeg programskog jezika.

U zavisnosti od toga da li se ceo program analizira i transformiše u mašinski kôd pre nego što može da se izvrši, ili se analiza i izvršavanje programa obavljaju naizmenično deo po deo programa (na primer, naredba po naredba), jezički procesori se dele u dve grupe: *kompilatore* i *interpretatore*.

Kompilatori. Kompilatori (ili kompajleri) su programski prevodioci kod kojih su faza prevođenja i faza izvršavanja programa potpuno razdvojene. Nakon analize *izvornog koda* programa višeg programskog jezika, kompilatori generišu *izvršivi (mašinski) kôd* i dodatno ga optimizuju, a zatim čuvaju u vidu *izvršivih (binarnih) datoteka*. Jednom sačuvani mašinski kôd moguće je izvršavati neograničen broj puta, bez potrebe za ponovnim prevođenjem. Krajnjim korisnicima nije neophodno dostavljati izvorni kôd programa na višem programskom jeziku, već je dovoljno distribuirati izvršivi mašinski kôd[5]. Jedan od problema u radu sa kompilatorima je da se prevođenjem gubi svaka veza između izvornog i izvršivog koda programa. Svaka (i najmanja) izmena u izvornom kodu programa zahteva ponovno prevođenje programa ili njegovih delova. S druge strane, kompilirani programi su obično veoma efikasni.

Interpretatori. Interpretatori su programski prevodioci kod kojih su faza prevođenja i faza izvršavanja programa isprepletane. Interpretatori analiziraju deo po deo (najčešće naredbu po naredbu) izvornog koda programa i odmah nakon analize vrše i njegovo izvršavanje. Rezultat prevođenja se ne smešta u izvršive datoteke, već je prilikom svakog izvršavanja neophodno iznova vršiti analizu izvornog koda. Zbog ovoga, programi koji se interpretiraju se obično izvršavaju znatno sporije nego u slučaju kompilacije. S druge strane, razvojni ciklus programa je često kraći ukoliko se koriste interpretatori. Naime, prilikom malih izmena programa nije potrebno iznova vršiti analizu celokupnog koda.

Za neke programske jezike postoje i interpretatori i kompilatori. Intepretator se tada koristi u fazi razvoja programa da bi omogućio interakciju korisnika sa programom, dok se u fazi eksploatacije kompletno razvijenog i istestiranog programa koristi kompilator koji proizvodi program sa efikasnim izvršavanjem.

Danas se često primenjuje i tehnika kombinovanja kompilatora i interpretatora. Naime, kôd sa višeg programskog jezika se prvo kompilira u neki precizno definisan međujezik niskog nivoa (ovo je obično jezik neke

[5]Ipak, ne samo u akademskom okruženju, smatra se da je veoma poželjno da se uz izvršivi distribuira i izvorni kôd programa (tzv. *softver otvorenog koda, engl. open source)* da bi korisnici mogli da vrše modifikacije i prilagođavanja programa za svoje potrebe.

apstraktne virtuelne mašine), a zatim se vrši interpretacija ili kompilacija u vreme izvršavanja (engl. just-in-time compilation) ovog međujezika i njegovo izvršavanje na konkretnom računaru. Ovaj pristup primenjen je kod programskog jezika Java, kod .Net jezika (C#, VB), programskog jezika Kotlin, itd.

Pitanja i zadaci za vežbu

Pitanje 4.1. *Ukoliko je raspoloživ kôd nekog programa na nekom višem programskom jeziku (na primer, na jeziku C), da li je moguće jednoznačno konstruisati odgovarajući mašinski kôd? Ukoliko je raspoloživ mašinski kôd nekog programa, da li je moguće jednoznačno konstruisati odgovarajući kôd na jeziku C?*

Pitanje 4.2. *Za koji programski jezik su izgrađeni prvi interpretator i kompilator i ko je bio njihov autor? Koji su najkorišćeniji programski jezici 1960-ih, koji 1970-ih, koji 1980-ih i 1990-ih, a koji danas? Šta je to strukturno, a šta objektno-orijentisano programiranje?*

Pitanje 4.3. *Koje su najznačajnije programske paradigme? U koju paradigmu spada programski jezik C? Šta su to proceduralni, a šta su to deklarativni jezici?*

Pitanje 4.4. *Šta je rezultat leksičke analize programa? Šta leksički analizator dobija kao svoj ulaz, a šta vraća kao rezultat svog rada? Šta je zadatak sintaksičke analize programa? Šta sintaksički analizator dobija kao svoj ulaz, a šta vraća kao rezultat svog rada?*

Pitanje 4.5. *Navesti primer leksički ispravnog, ali sintaksički neispravnog dela programa. Navesti primer sintaksički ispravnog, ali semantički neispravnog dela programa.*

Pitanje 4.6. *Šta karakteriše svaki tip podataka? Da li su tipovi prisutni i u prevedenom, mašinskom kodu? Šta je to konverzija tipova? Šta znači da je programski jezik statički, a šta znači da je dinamički tipiziran?*

Pitanje 4.7. *Koje su osnovne naredbe u programskim jezicima? Šta je to GOTO naredba i po čemu je ona specifična?*

Pitanje 4.8. *Čemu služe potprogrami? Koji su najčešće vrste potprograma? Čemu služe parametri potprograma, a čemu služi povratna vrednost? Koji su najčešći načini prenosa parametara?*

Pitanje 4.9. *Šta podrazumeva modularnost programa? Čemu služe biblioteke? Šta je standardna, a šta rantajm biblioteka?*

Pitanje 4.10. *Šta su prednosti, a šta mane mogućnosti pristupa proizvoljnoj memoriji iz programa? Šta su sakupljači otpada?*

Pitanje 4.11. *Šta je kompilator, a šta interpretator? Koje su osnovne prednosti, a koje su mane korišćenja kompilatora u odnosu na korišćenje interpretatora?*

Pitanje 4.12. *Ako se koristi kompilator, da li je za izvršavanje programa korisniku neophodno dostaviti izvorni kôd programa? Ako se koristi interpretator, da li je za izvršavanje programa korisniku neophodno dostaviti izvorni kôd programa?*

Indeks