Danijela Simić Filip Marić Predrag Janičić Milan Banković Milena Vujošević Janičić Vesna Marinković Mladen Nikolić

Mirko Spasić Sana Stojanović Đurđević Ivana Tanasijević

Elektronska verzija (2024)

UVOD U INFORMATIKU

Beograd
2024.

Elektronska verzija (2024)

Autori:
dr Danijela Simić, docent na Matematičkom fakultetu u Beogradu
dr Filip Marić, redovni profesor na Matematičkom fakultetu u Beogradu
dr Predrag Janičić, redovni profesor na Matematičkom fakultetu u Beogradu
dr Milan Banković, docent na Matematičkom fakultetu u Beogradu
dr Milena Vujošević Janičić, vandredni profesor na Matematičkom fakultetu u Beogradu
dr Mladen Nikolić, vandredni profesor na Matematičkom fakultetu u Beogradu
dr Mirko Spasić, docent na Matematičkom fakultetu u Beogradu
dr Sana Stojanović Đurđević, docent na Matematičkom fakultetu u Beogradu
dr Ivana Tanasijević, docent na Matematičkom fakultetu u Beogradu

UVOD U INFORMATIKU

Izdavač: Matematički fakultet Univerziteta u Beogradu
Studentski trg 16, 11000 Beograd

Recenzenti:
??
??

Obrada teksta i ilustracije: *autori* (osim za slike nabrojane na kraju knjige)
Dizajn korica: *autori*
©2024. Danijela Simić, Filip Marić, Predrag Janičić, Milan Banković, Milena Vujošević Janičić, Mladen Nikolić, Mirko Spasić, Sana Stojanović Đurđević i Ivana Tanasijević
Ovo delo zaštićeno je licencom Creative Commons CC BY-NC-ND 4.0 (Attribution-NonCommercial-NoDerivatives 4.0 International License). Detalji licence mogu se videti na veb-adresi http://creativecommons.org/licenses/by-nc-nd/4.0/. Dozvoljeno je umnožavanje, distribucija i javno saopštavanje dela, pod uslovom da se navedu imena autora. Upotreba dela u komercijalne svrhe nije dozvoljena. Prerada, preoblikovanje i upotreba dela u sklopu nekog drugog nije dozvoljena.

Elektronska verzija (2024)

Sadržaj

Sadržaj **3**

1 Kratka istorija informatike i informaciono-komunikacionih tehnologija **7**
1.1 Rana istorija računarskih sistema . . . 7
1.2 Računari Fon Nojmanove arhitekture . . . 10
1.3 Oblasti savremenog računarstva . . . 13

2 Hardver i softver **15**
2.1 Hardver savremenih računara . . . 15
2.2 Softver savremenih računara . . . 19

3 Digitalizacija **27**
3.1 Analogni i digitalni podaci i digitalni računari . . . 27
3.2 Zapis brojeva . . . 28
3.3 Zapis teksta . . . 33
3.4 Zapis multimedijalnih sadržaja . . . 38
3.5 Jezici za obeležavavanje . . . 40

4 Algoritmi i izračunljivost **57**
4.1 Prirodno-jezički opisi algoritama i izračunavanja . . . 57
4.2 Programi i izračunavanja na višem programskom jeziku . . . 58
4.3 Izračunavanja i programi na mašinski zavisnim jezicima . . . 59
4.4 Zasnivanje pojma algoritma . . . 62
4.5 Zaustavljanje programa i halting problem . . . 67
4.6 Algoritmika i vremenska i prostorna složenost izračunavanja . . . 69
4.7 Pregled . . . 70

5 Sistemski softver **73**
5.1 Operativni sistem . . . 73
5.2 Operativni sistemi zasnovani na UNIX-u . . . 85
5.3 Komandno okruženje operativnog sistema UNIX . . . 85

6 Računarske mreže **105**
6.1 Uloga računarskih mreža . . . 105
6.2 Komponente računarskih mreža . . . 106
6.3 Tipovi računarskih mreža . . . 109
6.4 Slojevitnost mreže i mrežni protokoli . . . 111
6.5 Bezbednost u računarskim mrežama . . . 112

7 Programski jezici i prevodioci **115**
7.1 Klasifikacije programskih jezika . . . 116
7.2 Leksika, sintaksa, semantika i pragmatika programskih jezika . . . 128
7.3 Jezički procesori . . . 132

8 Proces razvoja softvera **145**

Elektronska verzija (2024)

8.1 Planiranje . . . 146
8.2 Metodologije razvoja softvera . . . 147
8.3 Ekspoloatacija . . . 152
8.4 Alati i tehnike korišćeni u razvoju softvera . . . 152
8.5 Novi trendovi i tehnologije . . . 153
8.6 Dodatno . . . 153

9 Baze podataka **155**
9.1 Uvod u baze podataka . . . 155
9.2 Relacione baze podataka . . . 155
9.3 NoSQL baze podataka . . . 156
9.4 Arhitektura baza podataka . . . 156
9.5 Sigurnost i backup . . . 157
9.6 Trendovi i budućnost baza podataka . . . 157
9.7 Pitanja i odgovori . . . 157

10 Matematika i informatika **159**
10.1 Primene matematike u računarstvu i informatici . . . 159
10.2 Matematički softver . . . 165

11 Veštačka inteligencija **169**
11.1 Uska i opšta veštačka inteligencija . . . 169
11.2 Filozofski i etički aspekti veštačke inteligencije . . . 170
11.3 Talasi veštačke inteligencije . . . 171
11.4 Znanje i zaključivanje . . . 171
11.5 Pretraga . . . 172
11.6 Automatsko deduktivno rasuđivanje . . . 172
11.7 Mašinsko učenje i induktivno rasuđivanje . . . 174

12 Računarska grafika **181**

13 Računari i društvo **183**
13.1 Socijalni i etički aspekti računarstva . . . 183
13.2 Zavisnost od interneta i društvene mreže . . . 188
13.3 Pravni i ekonomski aspekti računarstva . . . 189

Spisak preuzetih slika **193**

Literatura **195**

Elektronska verzija (2024)

Predgovor

Ovo je materijal za predmet Uvod u informatiku na prvoj godini smera *Informatika* na Matematičkom fakultetu Univerziteta u Beogradu) Nadamo da izbor sadržaja i način prezentovanja mogu da budu zanimljivi ne samo studentima, već i svima drugima koje interesuje ova oblast računarstva.

Bićemo zahvalni čitaocima na svim ispravkama, sugestijama i komentarima koje nam pošalju.

- Kratka istorija informatike i informaciono-komunikacionih tehnologija – Filip Marić i Predrag Janičić
- Hardver i softver - Filip Marić, Predrag Janičić i Danijela Simić
- Digitalizacija - Filip Marić, Predrag Janičić i Danijela Simić
- Algoritmi i izračunljivost - Predrag Janičić
- Sistemski softver - Milan Banković
- Računarske mreže - Mirko Spasić
- Programski jezici i prevodioci - Milena Vujošević Janičić
- Proces razvoja softvera - Danijela Simić
- Baze podataka - Ivana Tanasijević
- Matematika i informatika - Filip Marić
- Veštačka inteligencija - Mladen Nikolić i Predrag Janičić
- Računarska grafika - Vesna Marinković
- Računari i društvo - Sana Stojanović Đurđević

Danijela Simić, Filip Marić, Predrag Janičić, Milan Banković, Milena Vujošević Janičić, Mladen Nikolić, Mirko Spasić, Sana Stojanović Đurđević i Ivana Tanasijević

Beograd, mart 2024.

Elektronska verzija (2024)

Elektronska verzija (2024)

Glava 1

Kratka istorija informatike i informaciono-komunikacionih tehnologija

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* 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,

[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.

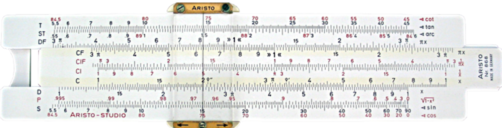
[3] Muhammad ibn Musa al-Khwarizmi (780–850), persijski matematičar.

[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*.

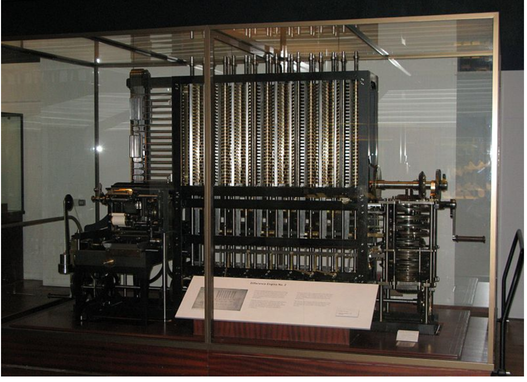
kasnije nazvane *Paskaline*, koje su služile za sabiranje i oduzimanje celih brojeva. *Gotfrid Lajbnic*[7] konstruisao je 1672. godine mašinu koja je mogla da izvršava sve četiri osnovne aritmetičke operacije (sabiranje, oduzimanje, množenje i deljenje) nad celim brojevima. Ova mašina bila je zasnovana 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.

[7] Gottfried Wilhelm Leibniz (1646–1716), nemački filozof i matematičar.
[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. Računske operacije izvršavao je hiljadu puta brže od elektromehaničkih mašina. Osnovna svrha bila mu je

[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).

Elektronsko izdanje (2021)

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.

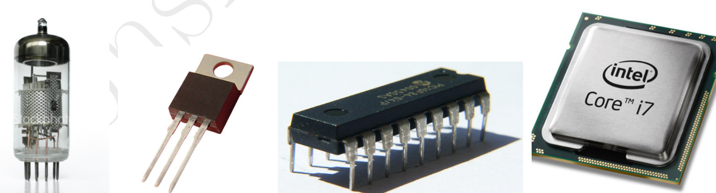
1.2 Računari Fon Nojmanove arhitekture

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.

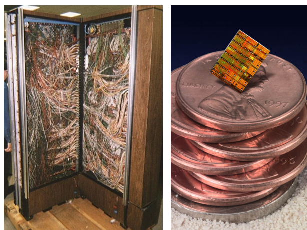
II generacija računara (od kraja 1950-ih do polovine 1960-ih) koristila je *tranzistore* umesto vakuumskih cevi. Iako je tranzistor otkriven još 1947. godine, tek sredinom pedesetih počinje da se koristi umesto vakuumskih

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

Elektronska verzija (2024)

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.

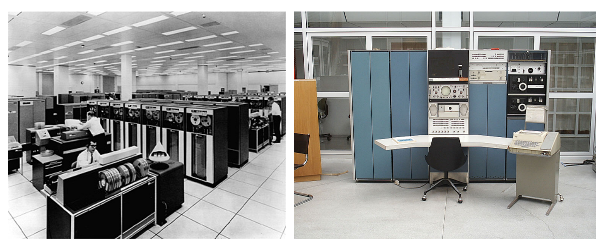


Slika 1.5: Integrisana kola dovela su do minijaturizacije i kompleksni žičani spojevi su mogli biti realizovani na izuzetno maloj površini.

Elektronska verzija (2024)

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

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.

[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 pojedinci koji na inovativan način modifikuju postojeće hardverske i softverske sisteme.

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);
- *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);

[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).

Elektronska verzija (2024)

- *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).

Elektronska verzija (2024)

Glava 2

Hardver i softver

2.1 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.

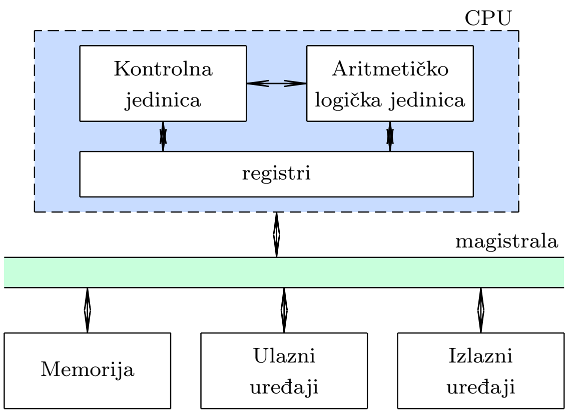
Iako na prvi pogled deluje da se jedan uobičajeni stoni računar sastoji od kućišta, monitora, tastature i miša, ova podela je veoma površna, podložna promenama (već kod prenosnih računara, stvari izgledaju znatno drugačije) i nikako ne ilustruje koncepte bitne za funkcionisanje računara. Mnogo značajnija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

Elektronska verzija (2024)



Slika 2.1: 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 računanja sve češće se koriste specijalizovani grafički procesori (engl. *Graphics Processing Unit*, *GPU*). Iako su prvobitno bili namenjeni isključivo za grafičke operacije, kao što su renderovanje 3D scena i obrada piksela, njihova visoko paralelizovana arhitektura omogućava efikasno izvršavanje različitih proračuna nad velikim blokovima podataka. Nasuprot tome, centralni procesori (engl. *Central Processing Unit*, *CPU*) su dizajnirani za efikasno izvođenje sekvencijalnih operacija i optimizovani su za brzo procesiranje jedne ili nekoliko niti (engl. *threads*). Dizajn grafičkih procesora zasnovan je na simetričnoj multiprocesorskoj arhitekturi, podeljenoj na više klastera (engl. *Streaming Multiprocessors*), gde svaki klaster sadrži manji broj jednostavnih jezgara koja dele memoriju i resurse. Glavna razlika između grafičkih i centralnih procesora leži u njihovoj unutrašnjoj arhitekturi: centralni procesori imaju nekoliko složenih jezgara optimizovanih za nisku latenciju, dok grafički procesori imaju stotine ili čak hiljade jednostavnih jezgara koja mogu paralelno izvršavati veliki broj niti. Takva arhitektura omogućava grafičkim procesorima da istovremeno obrađuju veliki broj podataka, što ih čini idealnim za zadatke kao što su grafička obrada, naučne simulacije, kriptografski proračuni, kao i za mašinsko učenje i analizu velikih podataka.

Uloga grafičkih procesora u mašinskom učenju postala je ključna zbog rastuće potrebe za izvođenjem intenzivnih numeričkih operacija u modelima dubokih neuronskih mreža. Neuronske mreže sastoje se od više slojeva sa hiljadama, a ponekad i milionima parametara, koje je potrebno paralelno ažurirati tokom obuke. Grafički procesori su idealni za ove zadatke jer omogućavaju paralelno izvršavanje velikog broja operacija sabiranja i množenja matrica, što značajno smanjuje ukupno vreme potrebno za obuku modela. Popularne softverske biblioteke, kao što su *TensorFlow* i *PyTorch*, koriste grafičke procesore za ubrzanje obuke modela, omogućavajući efikasno paralelno računanje pomoću CUDA (engl. *Compute Unified Device Architecture*) ili OpenCL (engl. *Open Computing Language*).

Kompanija NVIDIA razvila je CUDA, programski jezik koji omogućava programerima da pišu kod koji se efikasno izvršava na grafičkim procesorima. Ključna prednost CUDA jezika je njegova sposobnost da mapira zadatke na hiljade paralelnih jezgara unutar grafičkog procesora, maksimizujući iskorišćenost dostupnih resursa. Programeri koriste CUDA jezik za kreiranje malih funkcija (engl. *kernels*) koje se izvršavaju na svakom jezgru paralelno, što značajno ubrzava proračune. Upotreba CUDA jezika u kontekstu mašinskog učenja omogućila je efikasnu implementaciju algoritama kao što su konvolutivne i rekurentne neuronske mreže, čime su grafički procesori postali ključna komponenta u razvoju sistema zasnovanim na modelima veštačke inteligencije.

OpenCL je otvoreni standard za paralelno programiranje koji omogućava razvoj heterogenih aplikacija na različitim uređajima i platformama, čineći ga pogodnim za heterogeno računarstvo – programer definiše uređaje i resurse za izvršavanje zadataka, koji se mogu distribuirati na različite uređaje unutar iste mreže. Ključna prednost OpenCL-a je njegova portabilnost, jer aplikacije napisane u ovom jeziku mogu raditi na različitim platformama bez potrebe za promenama osnovnog koda.

Veštačka inteligencija postaje ključni faktor u unapređenju hardverskog dizajna, posebno u kontekstu projektovanja i optimizacije mikroprocesora i integrisanih kola [23]. U tradicionalnim alatima za automatizaciju elektronskog dizajna (eng. *Electronic Design Automation*, *EDA*), dizajniranje čipova je podrazumevalo niz ručnih koraka, uključujući raspoređivanje komponenti, povezivanje (eng. *routing*), analizu kašnjenja i optimiza-

ciju potrošnje energije. Sada, korišćenjem tehnika veštačke inteligencije kao što su neuronske mreže i algoritmi zasnovani na dubokom učenju, ovaj proces postaje znatno efikasniji i automatizovaniji.

Na primer, u sklopu optimizacije performansi, modeli veštačke inteligencije mogu da analiziraju milione mogućih konfiguracija čipa kako bi pronašli optimalnu ravnotežu između potrošnje energije i brzine rada procesora, što je posebno važno za modernu *RISC-V arhitekturu*. RISC-V je otvorena arhitektura koja omogućava fleksibilnost u kreiranju sopstvenih skupova instrukcija, a veštačka inteligencija omogućava inženjerima da brzo isprobavaju različite varijante instruktivnih setova i struktura unutar mikroarhitekture. Pored toga, metode veštačke inteligencije se primenjuju u modernim EDA alatima kao što su Cadence, Synopsys, i pomažu u proračunu kašnjenja, predikciji signala i automatskoj sintezi logičkih kola, omogućavajući dizajnerima da brzo generišu optimalne dizajne koji zadovoljavaju tražene performanse i sigurnosne standarde.

Integracija tehnologija veštačke inteligencije sa EDA alatima ne samo da smanjuje ukupno vreme potrebno za projektovanje, već i smanjuje potrošnju energije, dok istovremeno omogućava razvoj kompleksnih i specifičnih arhitektura koje objedinjavanju više funkcionalnosti u jedinstvenom dizajnu. Modeli veštačke inteligencije predviđaju najbolji način povezivanja komponenti kako bi se minimizovala kašnjenja signala i optimizovala prostorna raspodela unutar čipa, što pomaže u ispunjavanju zahtevnih kriterijuma za stalnim povećanjem performansi.

Kvantni računari predstavljaju revoluciju u svetu računarskih tehnologija, zasnivajući se na potpuno drugačijim principima u odnosu na klasične računare. Dok klasični računari koriste bitove kao osnovnu jedinicu informacija i mogu biti u stanju `0` ili `1`, kvantni računari koriste kvantne bitove ili kubitove (engl. *qubits*). Kubiti se razlikuju od klasičnih bitova po tome što mogu biti u superpoziciji stanja (`0` i `1`), što znači da mogu istovremeno postojati kao kombinacija oba stanja, sa određenim verovatnoćama koje se mogu promeniti tokom operacija zahvaljujući pojavi koja se naziva kvantna superpozicija. Ovaj princip omogućava kvantnim računarima da paralelno obrađuju veliku količinu informacija, eksponencijalno povećavajući njihov računski potencijal.

Pored superpozicije, kvantni računari koriste i kvantnu spregu (engl. *entanglement*), još jedan ključni kvantni fenomen. Kada su dva kubita upletena, stanje jednog kubita direktno zavisi od stanja drugog, bez obzira na fizičku udaljenost između njih. Ove osobine omogućavaju kvantnim računarima da rešavaju probleme koje klasični računari, čak i superkompjuteri, ne mogu rešiti u razumnom vremenskom roku.

Superkompjuteri, koji koriste klasične arhitekture sa hiljadama višejezgarnih CPU i GPU jedinica, oslanjaju se na paralelizam za ubrzanje sekvencijalnih algoritama. Iako su superkompjuteri kao što su Summit i Fugaku sposobni da izvršavaju kvadrilione operacija u sekundi, kvantni računari imaju potencijal da prevaziđu ovu sposobnost za specifične zadatke. Na primer, Šorov algoritam iz kriptografije može faktorisati velike brojeve eksponencijalno brže od najboljih klasičnih algoritama, što znači da bi dovoljno razvijen kvantni računar mogao da razbije današnje standarde kriptografije za nekoliko minuta, dok bi klasičnom računaru trebalo milijarde godina. Dok su klasični superkompjuteri ograničeni Morovim zakonom (ograničenje u broju tranzistora koji se mogu smestiti na jedan čip), kvantni računari nemaju takva ograničenja, što omogućava teoretski neograničeno povećanje performansi sa dodavanjem više kubita.

Međutim, kvantni računari su još uvek u ranoj fazi razvoja. Jedan od najvećih izazova je očuvanje kvantne koherentnosti, odnosno stabilnosti kvantnih stanja tokom vremena, jer su kubiti veoma osetljivi na spoljašnje smetnje. I najmanje promene u okruženju mogu destabilizovati kvantne operacije. Trenutno najmoćniji kvantni procesori imaju desetine do stotine kubita, ali se očekuje da će budući sistemi sadržati hiljade ili čak milione kubita, što će omogućiti izvođenje kompleksnih kvantnih simulacija i proračuna.

Iako današnji kvantni računari ne mogu zameniti superkompjutere za sve vrste problema, oni već nadmašuju klasične računare u specifičnim zadacima. Na primer, kvantne simulacije molekula i materijala mogu precizno modelovati kvantna svojstva hemijskih spojeva, što nije moguće ni sa najmoćnijim klasičnim računarima. Ove simulacije imaju potencijal da ubrzaju razvoj novih lekova, materijala i optimizaciju hemijskih procesa. Kvantni računari se sve više koriste i za optimizacione probleme, poput pronalaženja optimalnih ruta u velikim transportnim mrežama, rešavanja problema pakovanja i modelovanja finansijskih tržišta.

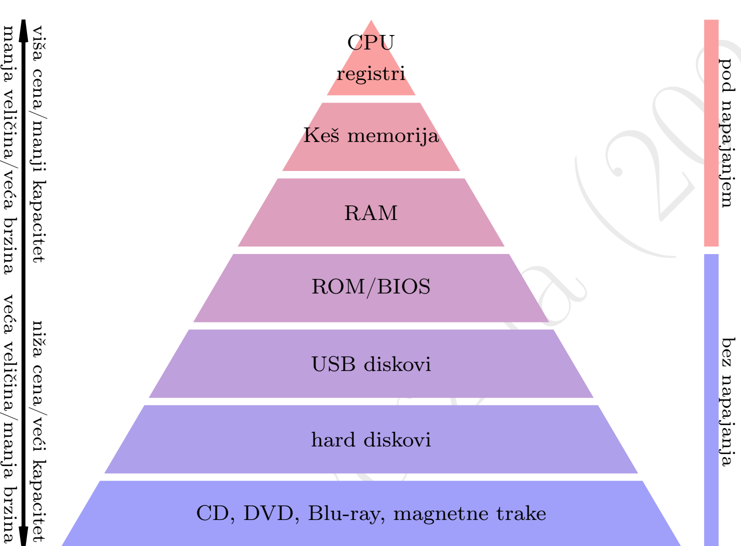
U budućnosti, kvantni računari imaju potencijal da promene kompletno računarsko polje, omogućavajući rešenja za trenutno nerešive probleme u kriptografiji, optimizaciji i simulacijama fizičkih i bioloških sistema. Kombinacija kvantnih i klasičnih računara, poznata kao kvantno–klasični hibridni sistemi, mogla bi postati standard u industriji, gde klasični računari obrađuju sekvencijalne delove algoritma, dok kvantni računari rešavaju najkompleksnije proračune. Ova kombinacija mogla bi da otključa mogućnosti koje danas smatramo naučnom fantastikom, pružajući potpuno novi nivo računarske moći i inovacija.

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.



Slika 2.2: Memorijska hijerarhija

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 podatak 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 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.

2.2 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.

2.2.1 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

Elektronska verzija (2024)

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.

2.2.2 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 2.2.1 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.

Elektronska verzija (2024)

- 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.

Elektronska verzija (2023)

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.

2.2.3 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*[1] (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

[1] 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.

Elektronska verzija (2024)

funkcionalnost operativnog sistema (preko sistemskih poziva), a zatim operativni sistem koristi funkcionalnost samog hardvera.

Pitanja i zadaci za vežbu

Pitanje 2.1. *Nabrojati osnovne periode u razvoju računara i navesti njihove osnovne karakteristike i predstavnike.*

Pitanje 2.2. *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 2.3. *Kojoj spravi koja se koristi u današnjem svetu najviše odgovaraju Paskalove i Lajbnicove sprave?*

Pitanje 2.4. *Kakva je veza između tkačkih razboja i računara s početka XIX veka?*

Pitanje 2.5. *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 2.6. *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 2.7. *Kada su nastali prvi elektronski računari? Nabrojati nekoliko najznačajnijih.*

Pitanje 2.8. *Na koji način je programiran računar ENIAC, a na koji računar EDVAC?*

Pitanje 2.9. *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 2.10. *Šta su to računari sa skladištenim programom? Šta je to hardver a šta softver?*

Pitanje 2.11. *Šta su procesorske instrukcije? Navesti nekoliko primera.*

Pitanje 2.12. *Koji su uobičajeni delovi procesora? Da li se u okviru samog procesora nalazi određena količina memorije za smeštanje podataka? Kako se ona naziva?*

Pitanje 2.13. *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 2.14. *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 2.15. *Koja serija Intelovih procesora je bila dominantna u PC računarima 1980-ih i 1990-ih godina?*

Pitanje 2.16. *Šta je to tehnološka konvergencija? Šta su to tableti, a šta „pametni telefoni“?*

Pitanje 2.17. *Koje su osnovne komponente savremenog računara? Šta je memorijska hijerarhija? Zašto se uvodi keš-memorija? Koje su danas najkorišćenije spoljne memorije?*

Pitanje 2.18. *U koju grupu jezika spadaju mašinski jezici i asemblerski jezici?*

Pitanje 2.19. *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 2.20. *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 2.1. *Na opisanom asemblerskom jeziku opisati izračunavanje vrednosti izraza* `x := x*y + y + 3`*. Generisati i mašinski kôd za napisani program.*

Zadatak 2.2. *Na opisanom asemblerskom jeziku opisati izračunavanje:*

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

Zadatak 2.3. *Na opisanom asemblerskom jeziku opisati izračunavanje:*

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

Zadatak 2.4. *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 2.21. *Koji su osnovni razlozi slojevite organizacije softvera? Šta je sistemski, a šta aplikativni softver?*

Pitanje 2.22. *Koji su osnovni zadaci operativnog sistema? Šta su sistemski pozivi? Koji su operativni sistemi danas najkorišćeniji?*

Elektronska verzija (202

Glava 3

Digitalizacija

Današnji računari su *digitalni*. To znači da su svi podaci koji su u njima zapisani – zapisani kao nizovi brojeva. Zapisivanje tekstova, slika, zvuka i filmova u 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.

3.1 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.

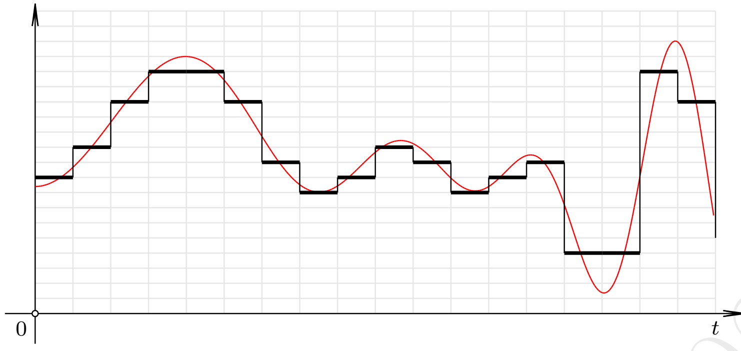
Osnovna prednost analogne tehnologije je da je ona obično veoma jednostavna ukoliko se zadovoljimo relativno niskim kvalitetom (još su drevni narodi mogli da naprave nekakav zapis zvuka uz pomoć jednostavne igle prikačene na trepereću membranu).

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

Elektronska verzija (2024)

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 3.1: 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.

3.2 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.

3.2.1 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 · b^i
```

ili, malo modifikovano:

```
x := a_0
za svako i od 1 do n
    x := x + a_i · 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 · b
    x := x + a_i · 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.

Elektronska verzija (2024)

```
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.

Elektronska verzija (2024)

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

3.2.2 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

Elektronska verzija (2024)

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.

3.2.3 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).

Elektronska verzija (2024)

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 $m_1 m_2 e_3$). Ako su prve dve cifre m_1 i m_2, tumačićemo da je mantisa $0, m_1 m_2$. 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.

3.3 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

Elektronska verzija (2021)

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 3.2: 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.

Elektronska verzija (2024)

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:

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

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	Ð	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 3.3: ISO-8859-1 tablica

Elektronska verzija (2024)

	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 3.4: 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 3.5: 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	Ѕ	І	Ї	Ј	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	ѕ	і	ї	ј	љ	њ	ћ	ќ	§	ў	

Slika 3.6: 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	Ј	¤	Ґ	¦	§	Ё	©	Є	«	¬		®	Ї
B	°	±	І	і	ґ	µ	¶	·	ё	№	є	»	ј	Ѕ	ѕ	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 3.7: 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

Elektronska verzija (2024)

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.

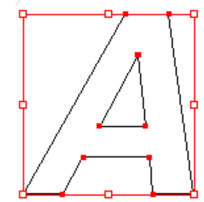
3.4 Zapis multimedijalnih sadržaja

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.

3.4.1 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 3.8: 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

Elektronska verzija (2024)

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).

3.4.2 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.

3.4.3 Zapis video sadržaja

Video zapis u računaru predstavlja kompleksan proces koji uključuje efikasno povezivanje slike i zvuka, uz primenu različitih tehnika kompresije i sinhronizacije. Video se sastoji od niza pojedinačnih *frejmova*, pri čemu je svaki frejm digitalna slika koja se kombinuje sa odgovarajućim segmentom zvučnog zapisa. *Koderi i dekoderi* (eng. codecs) igraju ključnu ulogu u ovom procesu. Koder je odgovoran za kombinovanje slike i zvuka u jedinstven *tok* (eng. stream) tokom snimanja, pri čemu se informacije komprimuju da bi se smanjila veličina fajla. Naime, i u zapisu videa, kao i zvuka javlja se redundacija podataka. Redundancija se odnosi na višak podataka koji nisu neophodni za reprodukciju, ali zauzimaju prostor i usporavaju prenos, te se uklanjanjem redundancije značajno povećava efikasnost.

Koderi (npr. *H.264, H.265, AV1*) su algoritmi koji vrše kompresiju video i audio podataka tokom snimanja. Koristi se tehnike poput vremenskog (eng. inter–frame) i prostornog (eng. intra–frame) kodiranja kako bi se smanjila redundancija. U toku ovog procesa posmatraju se uzastopni frejmovi i analiziraju sličnosti između njih. Koderi koriste različite algoritme, poput diskretne kosinusne transformacije (eng. Discrete Cosine Transform, DCT) i kompenzacije pokreta (eng. Motion Compensation) za smanjenje veličine podataka. Glavna ideja iza DCT je da se podaci rastave na različite frekvencije, pri čemu je većina u niskim frekvencijama. Niske frekvencije se mogu zadržati, dok se visoke frekvencije, koje predstavljaju manje značajne detalje, mogu delimično ili

Elektronska verzija (2024)

potpuno ukloniti, čime se smanjuje veličina podataka bez značajnog gubitka vizuelnog kvaliteta. Kod tehnike kompenzacije pokreta glavna ideja je da se pokretni elementi u video zapisu efikasno predstave pomoću modela predviđanja. Umesto da se kompletan frejm kodira iznova, kodira se samo razlika između trenutnog i prethodnog frejma, koristeći vektore pokreta za opisivanje kretanja objekata. *H.265*, poznat i kao *HEVC*, koristi naprednije tehnike predviđanja i particionisanja kako bi omogućio efikasniju kompresiju u odnosu na stariji *H.264*, dok *AV1*, kao projekat otvorenog koda, donosi još bolju kompresiju.

Prilikom reprodukcije, dekoder razdvaja podatke i rekonstruiše originalne frejmove slike zajedno sa odgovarajućim audio segmentima. Koriste se sofisticirani algoritmi za dekompresiju i rekonstrukciju podataka, odvajanjem frejmova slike od odgovarajućih audio segmenata i pritom osiguravajući da se slika i zvuk *sinhronizovano* i precizno prikažu korisniku. Sinhronizacija između zvuka i slike postiže se pomoću *vremenskih oznaka* (eng. timestamp) koji osiguravaju da se svaki deo zvučnog zapisa reprodukuje u tačno definisanom trenutku tokom prikazivanja određenog frejma. Ovaj proces je ključan za postizanje prirodnog osećaja pokreta i zvuka, jer svaki nesklad može izazvati percepcijski disonant kod korisnika. Dekoderi koriste tehnike kao što su kompenzacija kretanja zasnova na blokovima (eng. Block-based Motion Compensation) i inverzna kosinusna transformacija (eng. Inverse Discrete Cosine Transform, IDCT) kako bi povratili originalni kvalitet slike i zvuka.

Moderni video formati kao što su *MP4*, *MKV*, ili *MOV* koriste tzv. kontejnere koji integrišu video i audio tokove, ali i druge podatke, kao što su prevodi (eng. title) ili metapodaci. Kontejneri služe kao *omoti* koji omogućavaju simultano čitanje i reprodukciju svih ovih tokova na uređajima za reprodukciju, osiguravajući sinhronizaciju između zvučnih i vizuelnih elemenata. Na ovaj način, tokom reprodukcije, dekoder u realnom vremenu analizira, odvaja i dekodira frejmove videa i odgovarajuće segmente zvučnog zapisa, koristeći sofisticirane algoritme za predviđanje i rekonstrukciju kako bi održao preciznu sinhronizaciju, čak i u slučajevima gubitka podataka ili varijabilnih brzina prenosa. *MP4* je široko podržan format i često se koristi za uživo puštanje sadržaja zbog balansa između kvaliteta i veličine fajla. *MKV* pruža veću fleksibilnost i može integrisati više tokova (npr. više audio zapisa ili titlova), dok *MOV* format, razvijen od strane kompanije Apple, nudi visok nivo kvaliteta i često se koristi u profesionalnim okruženjima za uređivanje videa.

Jedan od ključnih izazova u zapisu videa je balans između kvaliteta i efikasnosti. Pojmovi kao što su *HD*, *Full HD*, i *4K* označavaju različite rezolucije video zapisa koje direktno utiču na kvalitet slike. *HD (High Definition)* se odnosi na rezoluciju od 1280×720 piksela, dok *Full HD* ima rezoluciju od 1920×1080 piksela, što se često označava kao 1080*p* na platformama poput YouTube servisa. Ovo *p* označava progresivno skeniranje, što znači da se svaki frejm prikazuje u celosti, što doprinosi boljem kvalitetu slike. U manjim formatima ili u slučaju upletenog (eng. interlaced) skeniranja, frejmovi se ne prikazuju svi odjednom, već se deli na polja, gde se prvo prikazuje polovina slike, a zatim druga polovina, što može dovesti do smanjenja kvaliteta slike ili povećanja *treperenja*. Više rezolucije, kao što su *4K* (3840×2160 piksela), pružaju još veću oštrinu i detalje, ali zahtevaju i veći protok podataka i veću memoriju za skladištenje.

Brzina bita (brzina protoka, eng. bitrate), izražen u megabitima po sekundi (Mbps), označava količinu podataka koja se prenosi ili obrađuje u jedinici vremena tokom reprodukcije videa. Viša brzina bita rezultira boljim kvalitetom slike, ali takođe zahteva veći kapacitet za skladištenje i širu mrežnu propusnost, dok niži brzina bita može uzrokovati gubitak detalja i pojavu vizuelnih nepravilnosti u slici. Brzina bita je u korelaciji sa rezolucijom i korišćenim kodekom – veće rezolucije, kao što su *4K*, obično zahtevaju veću brzinu bita za očuvanje kvaliteta. Na primer, za *Full HD* (1080*p*) video često se preporučuje brzina bita između 8-12 Mbps, dok za *4K* rezoluciju je potrebno da brzina bita bude između 35-45 Mbps kako bi se očuvao visok nivo detalja i kvaliteta slike. Takođe, moderniji kodeci, kao što su *H.265* i *AV1*, omogućavaju bolju kompresiju i kvalitet slike pri nižoj brzini bita u odnosu na starije kodeke, kao što je *H.264*. Na primer, *H.264* za *Full HD* (1080p) kvalitet može zahtevati brzinu bita od 8-12 Mbps, dok *H.265* može postići sličan kvalitet pri 4-6 Mbps, a *AV1* može još više smanjiti potrebnu brzinu bita, omogućavajući sličan kvalitet slike pri brzina bita od 3-5 Mbps za *Full HD* (1080p), što ga čini pogodnim za reprodukovanje sadržaja preko mreža sa ograničenom propusnošću.

Kompresija sa gubicima (eng. lossy compression) koristi psihovizuelne modele kako bi eliminisala informacije koje ljudsko oko manje primećuje, čime se drastično smanjuje veličina fajla. Na primer, visoke frekvencije boja i svetlosti koje oko teško registruje često se odstranjuju iz zapisa, što omogućava efikasniju upotrebu memorije. Pored toga, adaptivne metode kodiranja kao što je kvantizacija omogućavaju dinamičko prilagođavanje nivoa detalja u zavisnosti od kompleksnosti scene, čime se optimizuje kompresija u realnom vremenu. Ove tehnike zajedno omogućavaju da moderni video zapisi zadrže visok nivo vizuelnog kvaliteta uz relativno mali prostor za skladištenje, što je od ključnog značaja za primenu u prenosu podataka i skladištenju na savremenim uređajima.

3.5 Jezici za obeležavavanje

Postoje dva osnovna pristupa za kreiranje multimedijalnih dokumenata: pristup "Ono što vidiš je ono što dobijaš"(WYSIWYG) i pristup pomoću jezika za označavanje. Metoda WYSIWYG omogućava korisnicima da odmah vide kako će finalni dokument izgledati, koristeći grafičke alate za uređivanje. Ovaj pristup je intuitivan

i jednostavan za korišćenje, što ga čini pogodnim za korisnike koji preferiraju vizuelne povratne informacije dok kreiraju sadržaj. Međutim, WYSIWYG uređivači ponekad mogu ograničiti nivo preciznosti i mogućnost prilagođavanja, posebno kada su u pitanju složene potrebe za formatiranjem.

Nasuprot tome, jezici za označavanje nude drugačiji način kreiranja dokumenata, naglašavajući strukturu i preciznost. Jezik za označavanje je sistem za anotiranje dokumenta na način koji razlikuje anotacije od stvarnog sadržaja. Te anotacije, ili 'oznake', pružaju strukturu i instrukcije za formatiranje neophodne za efektivno predstavljanje informacija. Jezici za označavanje omogućavaju autorima da sistematski organizuju sadržaj, odrede njegov vizuelni prikaz, pa čak i dodaju metapodatke za obradu. Oni su osnovni u kreiranju i predstavljanju različitih oblika sadržaja, od jednostavnih beleški i članaka do složenih naučnih dokumenata, knjiga i web stranica.

Ključna prednost upotrebe jezika za označavanje je nivo kontrole koji pružaju nad strukturom i prezentacijom dokumenta. Za razliku od WYSIWYG uređivača dokumenata, jezici za označavanje omogućavaju detaljno prilagodjavanje, što ih čini idealnim za složene dokumente gde je konzistentno formatiranje od suštinskog značaja. Korišćenjem ova dva pristupa, autori mogu odabrati onaj koji najbolje odgovara njihovim potrebama — WYSIWYG za jednostavnost i trenutni prikaz ili jezike za označavanje za fleksibilnost, preciznost i doslednost.

Jezici za označavanje imaju svoje korene u izdavačkoj industriji, gde su urednici koristili posebne notacije za označavanje instrukcija za formatiranje i slaganje teksta. Ovi koncepti su adaptirani u digitalne formate, počevši od SGML-a (Standardni Generalizovani Jezik za Označavanje), koji je služio kao meta-jezik za definisanje drugih jezika za označavanje. Rane verzije HTML-a su bile definisane unutar okvira SGML-a, čineći HTML jednom od prvih široko korišćenih primena SGML-a. Jezici za označavanje definisani unutar SGML-a često se nazivaju SGML aplikacijama, što naglašava fleksibilnost i proširivost koje je SGML pružio kao osnovu za razvoj jezika za označavanje. Ova evolucija je utrla put modernim jezicima kao što su HTML za web sadržaj i LaTeX za akademsko pisanje.

3.5.0.1 Opšta struktura jezika za označavanje

Jezici za označavanje razdvajaju logičku strukturu dokumenta od njegove vizuelne prezentacije. Logička struktura podrazumeva organizovanje dokumenta u manje jedinice, poput poglavlja i paragrafa, kao i označavanje naglašenih delova teksta, kao što su citati i definicije. Vizuelna prezentacija, s druge strane, uključuje definisanje stila, poput tipova fonta, veličina, razmaka između redova i boja koje se koriste u različitim delovima dokumenta.

Primer: Opšta struktura jezika za označavanje Evo primera koji pokazuje jednostavan jezik za označavanje za organizovanje kolekcije recepata:

```
<!DOCTYPE kolekcija SYSTEM "recipe-collection.dtd">
<kolekcija>
  <recept author="Marko Peric" title="Cokoladna torta">
    <sastojci>
      <sastojak>2 šolje brašna</sastojak>
      <sastojak>1 šolja šećera</sastojak>
      <sastojak>1/2 šolje kakao praha</sastojak>
    </sastojci>
    <instrukcije>
      <korak>Ugrejte rernu na 350°F (175°C).</korak>
      <korak>Pomešajte sve suve sastojke.</korak>
      <korak>Pecite 30 minuta.</korak>
    </instrukcije>
  </recepti>
</kolekcija>
```

U ovom primeru su prikazani osnovni koncepti jezika za označavanje: elementi, oznake (tagovi) i atributi. Elementi, kao što su <recipe> i <ingredients>, predstavljaju logičke delove dokumenta, dok atributi, kao što su author i title, pružaju dodatne informacije o elementima.

Definicija tipa dokumenta (DOCTYPE) Jedan važan aspekt jezika za označavanje je definicija tipa dokumenta, koja osigurava da dokument poštuje određenu strukturu i pravila. Svaki jezik za označavanje koji se bazira na SGML-u (Standardni Generalizovani Jezik za Označavanje) koristi DOCTYPE deklaraciju da bi specificirao tip dokumenta koji se kreira. DOCTYPE deklaracija pruža informacije o Definiciji tipa dokumenta (DTD), koja definiše dozvoljene elemente, njihove odnose i atribute.

Na primer:

Elektronska verzija (2024)

```
<!DOCTYPE collection SYSTEM "recipe-collection.dtd">
```

Ova deklaracija ukazuje na to da je tip dokumenta kolekcija, a njegova struktura je definisana fajlom `recipe-collection.dtd`. `DTD` specificira koje oznake (tagovi) su dozvoljene, njihova značenja i kako mogu biti ugnježdene jedna u drugu.

Ispod je detaljniji primer DTD pravila za definisanje strukture dokumenta:

```
<!ELEMENT kolekcija - - (recept+)>
<!ELEMENT recept - - (sastojci, instrukcije)>
<!ELEMENT sastojci - O (sastojak+)>
<!ELEMENT instrukcije - O (korak+)>
<!ELEMENT sastojak - O (#PCDATA)>
<!ELEMENT korak - O (#PCDATA)>
```

U ovom primeru, `<ELEMENT>` se koristi za definisanje elemenata kao što su kolekcija, recept, sastojci i instrukcije. Simbol + označava da se element mora pojaviti bar jednom, dok `#PCDATA` označava parsirane karaktere, što znači da je tekst dozvoljen unutar tog elementa.

Jezici za označavanje koji koriste SGML, kao što je HTML, definišu sopstveni skup oznaka, njihovo značenje i moguće odnose među njima. Ova specifikacija tipa dokumenta određuje sintaksu i strukturu dokumenta, osiguravajući da se dokument kreira na osnovu unapred definisanih pravila.

3.5.0.2 Pregled različitih jezika za označavanje

Uobičajeni jezici za označavanje: Najšire korišćeni jezici za označavanje su HTML, LaTeX, Markdown i XML. Svaki od njih služi različitoj svrsi u zavisnosti od konteksta u kojem se koriste.

HTML (HyperText Markup Language): Pretežno se koristi za strukturiranje web stranica i web aplikacija. Definiše osnovne elemente web stranice, kao što su naslovi, paragrafi, linkovi, slike i drugo. HTML je osnova za web sadržaj i često se kombinuje sa CSS-om i JavaScript-om kako bi se kreirale vizuelno privlačne i interaktivne stranice.

LaTeX: Sistem za pripremu dokumenata koji se najčešće koristi za slaganje složenih dokumenata, naročito u akademskim i naučnim radovima. Odlikuje se izvanrednim formatiranjem matematičkih formula, naučnih notacija i bibliografija. Za razliku od programa za obradu teksta, LaTeX osigurava dosledno formatiranje, što je ključno za tehničke radove i disertacije.

Markdown: Lagan jezik za označavanje koji je jednostavan za pisanje i čitanje. Markdown se često koristi za kreiranje dokumentacije, README fajlova i blog postova. Njegova jednostavna sintaksa omogućava korisnicima brzo formatiranje teksta pomoću plain-text editora, čineći ga pristupačnim i tehničkim i netehničkim korisnicima.

XML (eXtensible Markup Language): Dizajniran za skladištenje i prenos podataka, XML se fokusira na strukturu i semantiku podataka, a ne na njihovu prezentaciju. XML se koristi u različitim aplikacijama, uključujući web servise, konfiguracione fajlove i razmenu podataka između sistema.

3.5.0.3 HTML (HyperText Markup Language)

Kratka istorija. Poreklo HTML-a datira iz ranih dana interneta 1991. godine. Razvoj HTML-a napredovao je prilično haotično (različite verzije HTML-a su razvijane nekoordinisano od strane različitih pojedinaca i grupa, što je dovelo do nedoslednosti). Od verzije 3.2, upravljanje HTML-om je preuzeo World Wide Web Konzorcijum (W3C) (http://www.w3c.org/). Trenutna verzija standarda je HTML5 (HTML 5.2).

HTML (HyperText Markup Language) je standardni jezik za kreiranje i strukturiranje web stranica. Na početku HTML dokumenta nalazi se DOCTYPE deklaracija koja označava verziju standarda koja se koristi; za HTML5, to se piše kao <!DOCTYPE html>. HTML dokument se sastoji od elemenata kao što su ceo dokument, paragrafi, tabele, slike i drugi.

Osnovni elementi. Elementi se označavaju pomoću tagova, koji se nazivaju i oznake ili markeri, i koji označavaju početak i kraj elementa. Većina elemenata ima početni tag u obliku <ime-elementa> i završni tag u obliku </ime-elementa>, koji obuhvataju sadržaj elementa.

Neki elementi nemaju sadržaj, i za njih se koriste samougašeni tagovi, kao što je `<ime-elementa/>`.

Elementi takođe mogu imati atribute koji pružaju dodatne informacije o njima. Atributi se navode unutar početnog taga u obliku ime-attributa="vrednost-attributa".

Primer: Kreiranje jednostavne web stranice HTML se koristi za kreiranje osnovnih delova web stranice. Evo osnovnog primera HTML dokumenta:

Elektronska verzija (2024)

```
<!DOCTYPE html>
<html>
  <head><title>Moja prva stranica</title></head>
  <body>
    <h1>Dobrodošli!</h1>
    <p>Ovo je primer HTML označavanja.</p>
  </body>
</html>
```

Kompletna web stranica u HTML-u je predstavljena jednim html elementom. Sadržaj html elementa sastoji se od dva glavna dela head i body. Sekcija <head> pruža važne informacije i linkove koji utiču na ceo dokument, dok sekcija <body> sadrži sadržaj koji korisnici vide na stranici.

head: Sekcija head sadrži meta-informacije o dokumentu, kao što su naslov, skup karaktera i linkovi ka spoljnim resursima poput CSS-a za stilizovanje ili JavaScript-a za funkcionalnost.

```
<head>
  <title>My First Page</title>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="styles.css">
    <link href="https://fonts.googleapis.com/css2?family=Roboto&display=swap" rel="stylesheet">
  </head>
```

<title>: Određuje naslov web stranice, koji se prikazuje na kartici pregledača i koriste ga pretraživači. Precizan i relevantan naslov doprinosi boljem rangiranju stranice u rezultatima pretrage.

Meta tagovi, <meta> se koriste u sekciji <head> HTML dokumenta za pružanje metapodataka – informacija o web stranici, kao što su kodiranje karaktera, opis, ključne reči i postavke prikaza (eng. viewport). Oni igraju ključnu ulogu u SEO optimizaciji, mobilnoj prilagodljivosti i pravilnom prikazu specijalnih karaktera. Još jedan primer meta taga je <meta name="viewportčontent="width=device-width, initial-scale=1.0», koji se koristi za kontrolu izgleda na mobilnim pregledačima i osigurava da stranica bude responzivna na različitim uređajima.

Tag <link> u HTML-u koristi se za povezivanje sa spoljnim resursima, kao što su stilovi ili fontovi. Najčešće se postavlja u sekciju <head> HTML dokumenta. Tag <link> je ključan za uključivanje stilova (npr. CSS fajlova), ikonica i drugih resursa koji unapređuju izgled i funkcionalnost web stranice. To je samougašeni tag i ne zahteva završni </link> tag. U gornjem primeru dat je link ka eksternom CSS fajlu koji definise stil stranice, a u drugom je uključen Google font Roboto koji će biti korišćen na stranici. Na taj način osiguravamo da se stranica prikazuje pravilno i zadržava isti izgled, čak i ako font Roboto nije instaliran na datom računaru, što omogućava dosledan prikaz bez obzira na uređaj sa kog se stranica pregleda.

body: Sekcija body sadrži sav sadržaj koji se prikazuje na stranici, kao što su tekst, slike, tabele i linkovi. U narednom delu biće prikazani najčešće korišćeni tagovi u okviru sekcije body.

Elementi za naslove. HTML obezbeđuje šest nivoa elemenata za naslove, od <h1> do <h6>. Ovi elementi se koriste za definisanje naslova različite važnosti.

- <h1> se koristi za najvažniji naslov, obično glavni naslov stranice.
- <h2> do <h6> se koriste za podnaslove, pri čemu je <h6> najmanje važan.

Evo primera kako se ovi tagovi za naslove mogu koristiti:

```
<body>
  <h1>Glavni naslov stranice</h1>
  <h2>Naslov sekcije</h2>
  <h3>Naslov podsekcije</h3>
  <h4>Manji naslov podsekcije</h4>
  <h5>Još manji naslov</h5>
  <h6>Najmanje važan naslov</h6>
</body>
```

Ovi naslovi pomažu u organizaciji sadržaja web stranice, čineći strukturu stranice jasnijom za korisnike. Takođe su važni za pretraživače i pristupačnost, jer pružaju jasnu hijerarhiju sadržaja.

Element paragrafa. Element <p> koristi se za definisanje paragrafa u HTML dokumentu. Paragrafi su blokovi teksta odvojeni od ostalog sadržaja, što olakšava čitanje i organizaciju informacija na stranici.

Evo primera kako se <p> tag može koristiti:

Elektronska verzija (2024)

```
<body> <p>Ovo je prvi paragraf na web stranici. On pruža uvod
u sadržaj koji je predstavljen.</p> <p>Ovo je još jedan paragraf, koji se
može koristiti za detaljnije objašnjenje, davanje primera ili pružanje
dodatnih informacija.</p> </body>
```

Element `<p>` pomaže u razdvajanju teksta na delove, poboljšavajući čitljivost i olakšava korisnicima da lakše prate informacije. Često se paragrafi obeležavaju različitim stilovima korišćenjem css-a, što dodatno povećava čitljivost.

Elementi liste. HTML nudi različite vrste elemenata za organizovanje sadržaja u liste:

- `<ul>` **(Neuređena lista)**: Ovaj element se koristi za kreiranje lista gde redosled stavki nije bitan. Stavke su obično označene tačkama.
- `<ol>` **(Uređena lista)**: Ovaj element se koristi za kreiranje lista gde je redosled stavki važan. Stavke su numerisane brojevima, slovima, rimskim brojevima i slično.
- `<dl>` **(Lista opisa)**: Ovaj element se koristi za kreiranje lista termina i njihovih opisa.

Evo primera kako se ovi elementi lista mogu koristiti:

```
<h2>Primer neuređene liste</h2>
<ul>
  <li>Stavka 1</li>
  <li>Stavka 2</li>
  <li>Stavka 3</li>
</ul>

<h2>Primer uređene liste</h2>
<ol>
  <li>Prvi korak</li>
  <li>Drugi korak</li>
  <li>Treći korak</li>
</ol>

<h2>Primer liste opisa</h2>
<dl>
  <dt>HTML</dt>
  <dd>Jezik za označavanje za kreiranje web stranica.</dd>
  <dt>CSS</dt>
  <dd>Jezik stilskih tablica koji se koristi za opisivanje izgleda i formatiranja HTML dokumenta.</d
</dl>
```

Elementi `<ul>` i `<ol>` pomažu u organizaciji sadržaja u obliku tačkica ili numerisanih lista, čineći sekvence ili kolekcije stavki lakšim za praćenje. Element `<dl>` se koristi za davanje definicija ili objašnjenja, što ga čini korisnim za bilo koju situaciju gde treba povezati termine sa opisima.

Tekstualni elementi. HTML nudi razne tekstualne elemente za naglašavanje ili promenu prikaza teksta:

- `<i>`: Koristi se za italic stil teksta, često za strane reči ili tehničke izraze.
- `<b>`: Koristi se za podebljanje teksta, obično radi isticanja ključnih reči ili važnih fraza.
- `<u>`: Podvlači tekst. Iako se ranije često koristio u HTML-u, danas se koristi ređe jer CSS pruža bolje opcije za stilizovanje.
- `<em>`: Naglašava tekst, obično prikazan u italic stilu. Takođe ukazuje na naglasak ili važnost za čitače ekrana.
- `strong>`: Označava jako naglašavanje, obično prikazano podebljano. Koristi se za označavanje važnosti i dostupno je čitačima ekrana.
- `<small>`: Koristi se za prikaz manjeg teksta, često za napomene ili sitno štampane informacije.
- `<br>`: Ubacuje prelazak u novi red. Za razliku od tagova za paragraf, koristi se za početak novog reda bez kreiranja novog bloka teksta.

Elektronska verzija (2024)

- Komentari (<!- komentar ->): Koriste se za dodavanje beleški unutar HTML koda koje nisu prikazane u pregledaču. Komentari su korisni za dokumentaciju koda ili privremeno onemogućavanje delova HTML-a.

Evo primera kako se ovi tagovi mogu koristiti:

```
<p>Ovo je paragraf sa <i>italic</i>, <b>podebljanim</b> i
<u>podvučenim</u> tekstom.</p> <p><em>Naglašen tekst</em> i <strong>snažan
tekst</strong> su važni za pristupačnost i prenošenje značenja.</p>
<p><small>Ovo je napomena u malom tekstu.</small></p> <p>Ovo je red
teksta.<br>Ovo je sledeći red, napravljen korišćenjem preloma reda.</p> <!--
Ovo je komentar koji neće biti vidljiv na prikazanoj stranici -->
```

različitih nivoa naglaska ili važnosti, poboljšavajući vizuelnu strukturu i pristupačnost web stranice.

Linkovi. Linkovi ili hiperveze koriste se za povezivanje dva resursa na internetu. Hiperveza je element na web stranici koji korisnici mogu aktivirati, obično klikom, što pretraživaču nalaže da učita novu stranicu ili resurs. Linkovi su opisani pomoću <a> elementa, a sadržaj ovog elementa predstavlja klikabilno područje koje aktivira link.

Atribut `href` koristi se za određivanje URL-a resursa koji treba da se prikaže kada se link aktivira.

```
<a href="http://www.matf.bg.ac.rs">Matematički fakultet,
Beograd</a>
```

Podrazumevano, linkovi se otvaraju u istoj kartici, ali može se koristiti atribut `target="_blank"` kako bi se otvorio link u novoj kartici.

```
<a href="http://www.matf.bg.ac.rs"
target="_blank">Matematički fakultet, Beograd (otvori u novoj kartici)</a>
```

Atribut `href` može sadržati apsolutnu ili relativnu adresu. **Apsolutni URL** uključuje kompletnu URL adresu, počevši od protokola kao što je `http://` ili `https://`.

```
<a href="http://www.example.com">Poseti Example</a>
```

Relativni URL se koristi za linkove unutar istog sajta i ne uključuju protokol niti naziv domena.

```
<a href="/about.html">O nama</a>
```

se poziva na putanje relativne u odnosu na trenutnu stranicu ili osnovni URL.

Moguće je i povezivanje na specifičan deo web stranice pomoću **fragment identifikatora**. Ovi identifikatori se definišu korišćenjem atributa `id` na elementu.

```
<h2 id="kontakt">Kontakt</h2>
...
<a href="#kontakt">Idi na sekciju Kontakt</a>
```

sa id-jem kontakt.

Hiperveze su osnovna funkcionalnost interneta, omogućavajući korisnicima da lako prelaze između povezanih resursa.

Tabele. Tabele se u HTML-u koriste za prikaz podataka u strukturiranom tabelarnom formatu. Sastoje se od redova i kolona, što olakšava organizovano prikazivanje informacija. Tabela se definiše pomoću elementa <table>, dok se redovi definišu pomoću elementa <tr>, a ćelije se predstavljaju pomoću <td> (standardna ćelija) za obične ćelije ili <th> (zaglavlje) za ćelije zaglavlja.

```
<table border="1">
  <tr>
    <th>Stavka</th>
    <th>Količina</th>
  </tr>
  <tr>
    <td>Jabuke</td>
    <td>10</td>
  </tr>
  <tr>
```

Elektronska verzija (2024)

```
    <td>Narandže</td>
    <td>15</td>
  </tr>
</table>
```

definiše red, `<th>` se koristi za ćelije zaglavlja (koje su podrazumevano prikazane podebljano i centrirano), a `<td>` se koristi za standardne ćelije podataka.

Atribut tabele `border` može se koristiti za dodavanje ivice tabeli. U gornjem primeru, `border="1"` dodaje ivicu širine 1 piksel tabeli.

Za spajanje više kolona ili redova, možete koristiti atribute `colspan` i `rowspan`, respektivno.

```
    <table border="1">
    <tr>
      <th colspan="2">Zalihe voća</th>
    </tr>
    <tr>
      <th>Stavka</th>
      <th>Količina</th>
    </tr>
    <tr>
      <td>Jabuke</td>
      <td>10</td>
    </tr>
  </table>
```

U ovom primeru, atribut `colspan="2"` koristi se da zaglavna ćelija obuhvati dve kolone.

```
<table border="1">
  <tr>
    <th rowspan="2">Kategorija</th>
    <th>Stavka</th>
    <th>Količina</th>
  </tr>
  <tr>
    <td>Jabuke</td>
    <td>10</td>
  </tr>
  <tr>
    <td>Voće</td>
    <td>Narandže</td>
    <td>15</td>
  </tr>
</table>
```

Ovde, `rowspan="2"` koristi se da zaglavlje "Kategorija"obuhvati dva reda.

Element `<caption>` koristi se za dodavanje naslova tabeli, koji se podrazumevano prikazuje iznad nje.

```
<table border="1">
  <caption>Zalihe voća</caption>
  <tr>
    <th>Stavka</th>
    <th>Količina</th>
  </tr>
  <tr>
    <td>Jabuke</td>
    <td>10</td>
  </tr>
</table>
```

Elektronska verzija (2024)

Ovaj naslov pruža tabeli objašnjenje, pomažući korisnicima da lakše razumeju šta predstavljaju podaci.

Slike. Web stranice takođe mogu uključivati multimedijalni sadržaj kao što su slike, audio i video klipovi. Za umetanje slika koristi se `<img>` element, koji je prazan tag (tj. nema završni tag).

Atribut `src` predstavlja URL slike i obavezan je. Preporučuje se korišćenje relativnih putanja za slike kada je to moguće.

Atribut `alt` pruža alternativni tekst koji će se prikazati ako pregledač ne može da prikaže sliku. Ovaj atribut je važan za pristupačnost, jer omogućava čitačima ekrana da opišu sadržaj slike korisnicima sa oštećenjem vida.

```
<img src="images/competition.jpg" alt="Učesnici na matematičkom takmičenju" title="Matematičko takı
```

Atribut title pruža prikaz teksta koji se pojavljuje kada korisnik pređe mišem preko slike. Atributi `width` i `height` definišu dimenzije slike, koje mogu biti postavljene u pikselima ili kao procenat u odnosu na kontejner elementa. Preporučuje se da ove vrednosti odgovaraju stvarnim dimenzijama slike kako bi se izbegli problemi sa skaliranjem.

Spoljni sadržaj. Osim slika, HTML omogućava ugrađivanje drugih tipova multimedijalnog sadržaja, kao što su YouTube video zapisi ili Google Mape, pomoću `<iframe>` taga.

```
<iframe width="560" height="315" src="https://www.youtube.com/embed/dQw4w9WgXcQ" title="YouTube video p
```

omogućavajući korisnicima da reprodukuju video bez napuštanja sajta.

Primer ugrađivanja Google Mapa:

```
<iframe src="https://www.google.com/maps/embed?pb=!1m18!1m12!1m3!1d3151.8354345093644!2d144.9556513158
```

Ovaj primer ugrađuje interaktivnu Google Mapu, omogućavajući korisnicima da direktno na stranici pregledaju lokacije.

Opšti elementi. HTML pruža dva opšta elementa koja se često koriste za grupisanje sadržaja bez dodavanja posebnog značenja: `<div>` i `<span>`.

Element `<div>` je blokovski kontejner koji se koristi za grupisanje većih delova sadržaja. Često se koristi za obuhvatanje sekcija web stranice, kao što su navigacioni meniji, bočne trake ili čitavi segmenti stranice. Posebno je koristan kada se kombinuje sa globalnim atributima id i class kako bi se stilizovale ili manipulisale te sekcije pomoću CSS-a ili JavaScript-a.

```
    <div id="galerija">
  <h2>Galerija slika</h2>
  <img src="images/photo1.jpg" alt="Fotografija 1" />
  <img src="images/photo2.jpg" alt="Fotografija 2" />
  <img src="images/photo3.jpg" alt="Fotografija 3" />
</div>
```

U ovom primeru, `<div>` element se koristi za grupisanje skupa slika i njihovog naslova, omogućavajući lako stilizovanje i kontrolu galerije kao celine.

Element `<span>` je jednolinijski kontejner koji se koristi za grupisanje manjih delova teksta ili drugih jednolinijiskih elemenata. Često se koristi kada je potrebno stilizovati ili dodati drugačije ponašanje određenom delu teksta bez stvaranja novog bloka.

```
<p>Glavne komponente računara su <span class="highlight">procesor</span>, <span class="highlight">r
```

Ovde se `<span>` koristi za isticanje pojedinačnih komponenti unutar paragrafa. Atribut `class="highlight"` može se koristiti za primenu specifičnog stila (npr. promenu boje teksta) na ove termine pomoću CSS-a.

HTML je ključan za kreiranje strukture web stranica. U kombinaciji sa CSS-om za stilizovanje (iako CSS nije jezik za označavanje, već jezik stilskih tablica koji se koristi za primenu stilova, kao što su boje, fontovi i raspored, kako bi se poboljšao izgled tog sadržaja) i JavaScript-om za interaktivnost, HTML služi kao osnova za sav web razvoj.

Elektronska verzija (2024)

3.5.0.4 LaTeX

Kratka istorija. Poreklo LaTeX-a datira iz ranih 1980-ih. Razvio ga je Lesli Lamport (Leslie Lamport) na osnovu sistema za slaganje teksta TeX Donalda Knuta (Donald Knuth), s ciljem da obezbedi jednostavniji način za kreiranje kvalitetno formatiranih dokumenata. LaTeX je brzo stekao popularnost u akademskim krugovima, posebno u matematici, računarstvu i inženjerstvu, zbog svoje sposobnosti da obradi složene sadržaje kao što su matematičke formule i bibliografije. Danas je LaTeX de facto standard za akademsko izdavaštvo u mnogim naučnim oblastima, zahvaljujući preciznosti u formatiranju i snažnim mogućnostima za upravljanje referencama i jednačinama.

Kako se .tex dokument prevodi Da bi se .tex dokument preveo u čitljiv format (kao što je PDF), potrebno je koristiti LaTeX kompajler. Najčešće korišćeni alati za prevođenje su: - **pdflatex**: Direktno prevodi .tex fajl u PDF format. Na primer, u terminalu možete pokrenuti:

```
pdflatex naziv_fajla.tex
```

Ova komanda će generisati PDF fajl iz vašeg LaTeX dokumenta. - **xelatex** ili **lualatex**: Ovi kompajleri omogućavaju korišćenje Unicode karaktera i jednostavan rad sa različitim fontovima. Pokreću se slično kao pdflatex.

Takođe postoje integrisana razvojna okruženja kao što su **TeXShop**, **TeXworks**, ili **Overleaf** (online alat), koji omogućavaju jednostavno uređivanje i kompajliranje LaTeX dokumenata sa samo nekoliko klikova.

Osnovni elementi. LaTeX se koristi za generisanje visokokvalitetnih dokumenata koristeći jednostavnu sintaksu za definisanje elemenata. Dokument započinje deklaracijom tipa dokumenta kao što je `\documentclass` koja definiše osnovne karakteristike dokumenta, npr. članak, knjigu ili prezentaciju.

```
\documentclass{article}
  \title{Moj prvi LaTeX dokument}
  \author{Danijela Simic}
  \date{\today}

\begin{document}

  \maketitle

  Dobrodošli u svet LaTeX-a!
\end{document}
```

U ovom primeru kreiramo osnovni dokument sa naslovom, autorom i datumom, koji će biti automatski generisan.

- `\documentclass{article}`: Ova komanda definiše tip dokumenta. Umesto `article`, mogu se koristiti i drugi tipovi kao što su `report` (izveštaj), `book` (knjiga), ili `beamer` (prezentacija). Mogu se dodati i opcije poput veličine fonta (npr. `12pt`, `11pt`, `10pt`) ili formata (npr. `twocolumn`, `landscape`). Primer:

```
\documentclass[12pt, a4paper]{article}
```

Ovde definišemo veličinu fonta od 12pt na A4 papiru.

- ** **: Postavlja naslov dokumenta. - ** **: Definiše autora dokumenta. - ** **: Postavlja datum dokumenta. Komanda `7. novembar 2024.` automatski koristi današnji datum, ali možete navesti i specifičan datum, npr. . - **

Elektronska verzija (2024)

1. januar 2024

: Ova komanda generiše stranicu sa naslovom, autorom i datumom na osnovu prethodno definisanih vrednosti. - **documentclass: Kao što je već pomenuto, ova komanda definiše tip dokumenta, što utiče na osnovni izgled i ponašanje dokumenta. - **Veličina fonta i format stranice**: Možete odrediti veličinu fonta (npr. `10pt`, `12pt`) i format stranice (npr. `a4paper`, `letterpaper`) unutar `documentclass` opcija. Primer:

```
\documentclass[12pt, a4paper]{article}
```

Ovde se podešava veličina fonta i format papira. - **Paketi**: Nakon `documentclass` linije, možete koristiti `usepackage` komande da uključite različite pakete koji proširuju funkcionalnost LaTeX-a. Na primer:

```
\usepackage{graphicx}  % Za rad sa slikama
\usepackage{amsmath}   % Za napredne matematičke formule
```

- **Podešavanje fontova i stilova**: Paket `fontspec` (u XeLaTeX i LuaLaTeX) omogućava korišćenje različitih fontova, dok osnovni LaTeX koristi standardne fontove.

Kada su sva podešavanja obavljena, započinje glavni deo dokumenta sa `begindocument`. Sve što se nalazi unutar `begin document` i `enddocument` predstavlja sadržaj dokumenta, uključujući tekst, slike, matematičke formule, tabele i ostalo.

Struktura dokumenta. Dokumenti u LaTeX-u su strukturirani pomoću različitih nivoa naslova: `section`, `subsection`, `subsubsection`, itd. Ovo omogućava hijerarhijsko organizovanje sadržaja, što olakšava preglednost i navigaciju.

```
\section{Uvod}
Ovo je uvodni deo dokumenta.

\subsection{Motivacija}
Ovde objašnjavamo zašto koristimo LaTeX.
```

Liste. LaTeX podržava neuređene i uređene liste koje olakšavaju organizaciju sadržaja.

```
\begin{itemize}
    \item Prva stavka.
    \item Druga stavka.
\end{itemize}

\begin{enumerate}
    \item Prva stavka.
    \item Druga stavka.
\end{enumerate}
```

Elektronska verzija (2024)

Tabele. Kreiranje tabela u LaTeX-u omogućava organizovano predstavljanje podataka.

```
\begin{table}[h]
  \centering
  \begin{tabular}{|c|c|}
    \hline
    Stavka & Količina \\
    \hline
    Jabuke & 10 \\
    Narandže & 15 \\
    \hline
  \end{tabular}
  \caption{Zalihe voća}
  \label{tab:zalihe}
\end{table}
```

Slike. Slike se u LaTeX-u uključuju pomoću paketa `graphicx` i naredbe `\includegraphics`. Preporučuje se da slike budu u PDF, PNG ili JPEG formatu.

```
\usepackage{graphicx}
\begin{figure}[h]
  \centering
  \includegraphics[width=0.5\textwidth]{slika.png}
  \caption{Primer slike}
  \label{fig:primer_slika}
\end{figure}
```

Tekstualni elementi. LaTeX omogućava formatiranje teksta korišćenjem naredbi kao što su za podebljavanje, za kurziv, i za podvlačenje. Komentari se mogu dodati korišćenjem znaka procenta (%), koji označava da ostatak linije neće biti procesuiran.

```
\textbf{Ovo je podebljani tekst}, dok je \textit{ovo kurziv}.
% Ovo je komentar u LaTeX kodu
```

LaTeX je moćan alat za stvaranje tehničkih i naučnih dokumenata, nudeći visoku preciznost i fleksibilnost pri formatiranju matematičkih formula, tabela, slika i složenih struktura teksta. U kombinaciji sa BibTeX-om za citiranje i različitim paketima za napredne funkcionalnosti, LaTeX ostaje ključni alat za akademsko i naučno pisanje.

Matematičke formule. Jedna od glavnih prednosti LaTeX-a je jednostavno i efektivno pisanje matematičkih formula. Matematički izrazi mogu biti ubačeni unutar teksta koristeći dvostruke zagrade `$ ... $` ili kao zasebni blok koristeći

...

.

Primeri matematičkih formula:

- Jednostavna inline formula: `$ a^2 + b^2 = c^2 $`
- Zasebna formula:

  ```
  \[
    E = mc^2
  \]
  ```

- Složena formula sa integracijom:

  ```
  \begin{equation}
    \int_a^b x^2 \, dx = \frac{b^3 - a^3}{3}
  \end{equation}
  ```

Elektronska verzija (2024)

- Matrica reda 2×2:

```
\[
  \begin{bmatrix}
    a & b \\
    c & d
  \end{bmatrix}
\]
```

- Formula za sumu:

```
\[
  \sum_{i=1}^n i = \frac{n(n+1)}{2}
\]
```

- Diferencijalna jednačina:

```
\begin{equation}
  \frac{d^2y}{dx^2} + 3\frac{dy}{dx} + 2y = 0
\end{equation}
```

- Integral sa trigonometrijom:

```
\begin{equation}
  \int_0^{\pi} \sin(x) \, dx = 2
\end{equation}
```

- Razlomak:

```
\[
  \frac{a+b}{c+d}
\]
```

- Složena suma:

```
\[
  \sum_{i=1}^n \frac{1}{i^2}
\]
```

- Proizvod:

```
\[
  \prod_{i=1}^n i = n!
\]
```

- Limiti:

```
\begin{equation}
  \lim_{x \to 0} \frac{\sin x}{x} = 1
\end{equation}
```

Elektronska verzija (2024)

Kreiranje sadržaja. Da bi se kreirao sadržaj u LaTeX dokumentu, koristi se komanda `\tableofcontents`. Ova komanda se postavlja na mesto u dokumentu gde želite da se sadržaj prikaže, obično neposredno nakon komande `\maketitle`. Sadržaj se automatski generiše na osnovu naslova kao što su `\section`, `\subsection`, i `\subsubsection` koji su definisani u dokumentu.

```
\documentclass{article}
\begin{document}
  \title{Moj LaTeX dokument}
  \author{Danijela Simić}
  \date{\today}
  \maketitle
  \tableofcontents

  \section{Uvod}
  Ovo je uvodni deo dokumenta.

  \subsection{Motivacija}
  Ovde objašnjavamo zašto koristimo LaTeX.
\end{document}
```

Važno je napomenuti da je potrebno dva puta prevesti (kompajlirati) dokument da bi se sadržaj ispravno generisao. Prvi put LaTeX kreira pomoćni fajl sa informacijama o stranicama naslova, a drugi put koristi taj fajl da bi ispravno prikazao sadržaj u PDF-u. Dakle, u terminalu treba pokrenuti komandu pdflatex naziv_fajla.tex dva puta zaredom.

3.5.0.5 Markdown (Jezik za označavanje)

Kratka istorija. Markdown je razvijen 2004. godine od strane Johna Grubera, u saradnji sa Aaronom Swartzom, kao lagani jezik za označavanje koji je dizajniran za brzo formatiranje teksta. Cilj Markdown-a bio je da bude čitljiv i lako razumljiv čak i u izvornom obliku, kao i da se jednostavno konvertuje u HTML. Danas se Markdown koristi širom interneta, posebno na platformama kao što su GitHub, Reddit, i mnogi blogovi, zbog svoje jednostavnosti i fleksibilnosti.

Kako se koristi Markdown. Markdown omogućava brzo kreiranje formatiranih dokumenata pomoću jednostavne sintakse koja koristi razne simbole za kreiranje elemenata kao što su naslovi, liste, linkovi, slike i kod blokovi.

Osnovni elementi. **Naslovi** Naslovi se kreiraju pomoću znaka `#`, pri čemu broj znakova označava nivo naslova (od 1 do 6):

```
# Naslov 1
## Naslov 2
### Naslov 3
#### Naslov 4
##### Naslov 5
###### Naslov 6
```

Paragrafi Paragrafi se formiraju jednostavnim pisanjem teksta, a za razdvajanje paragrafa koristi se jedan ili više praznih redova.

Podebljani i italic tekst - *Italic* tekst se označava stavljanjem teksta između jedne zvezdice (`*`) ili donje crte (`_`):

```
*italic* ili _italic_
```

- **Podebljani** tekst se označava sa dve zvezdice (`**`) ili dve donje crte (`__`):

```
**podebljani** ili __podebljani__
```

- Kombinacija podebljanog i italics moguća je kombinovanjem zvezdica ili donjih crta:

Elektronska verzija (2024)

```
**_italic i podebljani_**
```

Liste Markdown podržava dve vrste listi: neuređene i uređene.
- **Neuređene liste** kreiraju se pomoću zvezdica (*), znakova plus (+) ili crtica (-):

```
* Stavka 1
* Stavka 2
  * Podstavka 2.1
  * Podstavka 2.2
```

- **Uređene liste** kreiraju se brojevima praćenim tačkom:

```
1. Prva stavka
2. Druga stavka
   1. Podstavka 2.1
   2. Podstavka 2.2
```

Linkovi Linkovi se kreiraju korišćenjem uglastih zagrada za tekst linka, a zatim obličnih zagrada za URL:

```
[Posetite W3C](http://www.w3c.org/)
```

Slike Sintaksa za slike je slična sintaksi za linkove, ali sa uzvičnikom (!) na početku:

```
![Alternativni tekst](url_slike)
```

Kod blokovi Markdown omogućava umetanje koda unutar linije koristeći znakove backtick (`).

```
Ovo je ‘inline kod‘ primer.
```

Za višeredne blokove koda koriste se tri backtick znaka:

```
‘‘‘
function pozdrav() {
    console.log("Zdravo, svetu!");
}
‘‘‘
```

Takođe, mogu se dodati jezici nakon ‘‘‘ za naglašavanje sintakse, npr.:

```
‘‘‘python
print("Hello, World!")
‘‘‘
```

Citatni blokovi Citatni blokovi se kreiraju korišćenjem znaka > ispred teksta:

```
> Ovo je citatni blok.
```

Horizontalna linija Horizontalne linije se kreiraju koristeći tri ili više zvezdica (***), crtica (--) ili donje crte (___):

```
---
```

Linkovi ka naslovima u dokumentu Markdown omogućava kreiranje linkova ka određenim delovima dokumenta koristeći id-ove naslova. Na primer:

```
[Idi na Naslov 2](#naslov-2)
```

Zaključak. Markdown je jednostavan, lagan jezik za označavanje koji omogućava brzo kreiranje formatiranih dokumenata koji su čitljivi i u izvornom obliku. Zahvaljujući svojoj jednostavnosti i fleksibilnosti, Markdown je postao veoma popularan, posebno među programerima i autorima tehničkih dokumenata.

Elektronska verzija (2024)

3.5.0.6 XML (eXtensible Markup Language)

XML je svestrani jezik za označavanje koji se primarno koristi za skladištenje i prenos podataka. Za razliku od HTML-a, koji je dizajniran za prikazivanje podataka, XML se fokusira na definisanje struktura podataka i semantike. Omogućava programerima da kreiraju prilagođene oznake (tagove) za opisivanje značenja podataka, pružajući fleksibilan i lako čitljiv format. XML se široko koristi u web servisima, razmeni podataka i konfiguracionim fajlovima.

Primer XML-a

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="fiction">
    <title lang="en">The Great Gatsby</title>
    <author>F. Scott Fitzgerald</author>
    <year>1925</year>
    <price>10.99</price>
  </book>
  <book category="non-fiction">
    <title lang="en">Sapiens: A Brief History of Humankind</title>
    <author>Yuval Noah Harari</author>
    <year>2011</year>
    <price>14.99</price>
  </book>
</bookstore>
```

U ovom primeru, XML se koristi za opisivanje inventara knjižare. Svaki element <book> sadrži ugnježdene oznake koje pružaju detalje o pojedinačnim knjigama, kao što su naslov, autor, godina i cena. Atribut `category` pruža dodatne informacije o vrsti knjige, dok atribut `lang` označava jezik naslova knjige.

XML se obično koristi u sledećim scenarijima:

- **Web servisi**: Kao format za razmenu podataka između različitih sistema, kao što je SOAP (Simple Object Access Protocol).
- **Konfiguracioni fajlovi**: Mnoge aplikacije koriste XML za svoje konfiguracione postavke, obezbeđujući doslednu strukturu koja je jednostavna za čitanje i izmene.
- **Reprezentacija podataka**: XML se koristi za predstavljanje složenih struktura podataka na način koji je čitljiv i za ljude i za računare, što ga čini idealnim za interoperabilnost između različitih aplikacija i sistema.

Elektronska verzija (2024)

Pitanja i zadaci za vežbu

Pitanje 3.1. *Kako su u digitalnom računaru zapisani svi podaci? Koliko se cifara obično koristi za njihov zapis?*

Pitanje 3.2. *Koje su osnovne prednosti digitalnog zapisa podataka u odnosu na analogni? Koji su osnovni problemi u korišćenju digitalnog zapisa podataka?*

Pitanje 3.3. *Šta je Hornerova shema? Opisati Hornerova shemu pseudo-kodom.*

Pitanje 3.4. *Koje su prednosti zapisa u potpunom komplementu u odnosu na zapis u obliku označene apsolutne vrednosti?*

Pitanje 3.5. *Šta je to IEEE 754?*

Pitanje 3.6. *Šta je to glif, a šta font? Da li je jednoznačna veza između karaktera i glifova? Navesti primere.*

Pitanje 3.7. *Koliko bitova koristi ASCII standard? Šta čini prva 32 karaktera ASCII tabele? Kako se određuje binarni zapis karaktera koji predstavljaju cifre?*

Pitanje 3.8. *Navesti barem dve jednobajtne kodne strane koje sadrže ćirilične karaktere.*

Pitanje 3.9. *Nabrojati bar tri kodne sheme u kojima može da se zapiše reč* računarstvo.

Pitanje 3.10. *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 3.11. *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 3.12. *U čemu je razlika između Unicode i UTF-8 kodiranja?*

Pitanje 3.13. *Prilikom prikazivanja filma, neki program prikazuje titlove tipa* `"raèunari æe biti..."`. *Objasniti u čemu je problem.*

Pitanje 3.14. *Šta je to piksel? Šta je to sempl?*

Zadatak 3.1. *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 3.2. *Zapisati dekadni broj* 254 *u osnovama* 2, 8 *i* 16. √

Zadatak 3.3. *(a)*
Registar ima sadržaj

1010101101001000111101010101011001101011101010101110001010010011.

Zapisati ovaj broj u heksadekadnom sistemu.

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

Zadatak 3.4. *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 3.5. *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 3.6. *Ukoliko se koristi binarni zapis neoznačenih brojeva širine* 8 *bitova, zapisati brojeve:*
(a) 12 *(b)* 123 *(c)* 255 *(d)* 300 √

Zadatak 3.7. *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 3.8. *Odrediti zapis narednih brojeva u binarnom potpunom komplementu širine* 8 *bitova:*
(a) 12 *(b)* -123 *(c)* 0 *(d)* -18 *(e)* -128 *(f)* 200 √

Zadatak 3.9. *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 3.10. *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 3.11. *Odrediti broj bitova neophodan za kodiranje 30 različitih karaktera.* √

Zadatak 3.12. *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 3.13. *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 3.14. *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 3.15. *Odrediti (heksadekadno predstavljene) kodove kojima se zapisuje tekst* `kružić` *u UCS-2 i UTF-8 kodiranjima. Rezultat proveriti korišćenjem HEX editora.* √

Zadatak 3.16. *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 3.17. *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 3.18. *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 3.19. *Korišćenjem nekog naprednijeg grafičkog programa (na primer, GIMP ili Adobe Photoshop) videti kako se boja* `#B58A34` *predstavlja u CMY i HSB modelima.*

Elektronska v

Glava 4

Algoritmi i izračunljivost

Računari mogu da obave razne zadatke, mnoge – na zadivljujući način. Zato je interesantno a i veoma važno pitanje šta sve računari *mogu* a i pitanje šta računari *ne mogu* da urade. Da bismo mogli da odgovorimo na ta pitanja, moraćemo da odgovorimo na pitanje šta je to *algoritam*[1]. Neformalno govoreći, algoritam je precizan opis postupka za rešavanje nekog problema u konačnom broju koraka. Svaki računarski *program* je konkretna implementacija nekog algoritma u nekom konkretnom programskom jeziku, na primer, u jeziku Java, C ili C++ i koju računar može da izvrši. Danas granica između algoritama i programa nije kruta jer postoje, na primer, sistemi u kojima opisi algoritama u vidu dijagrama mogu da se neposredno izvršavaju (na primer, Blockly/Scratch dijagrami).

Algoritmi se primenjuju svakodnevno i na svakom koraku. Postoje, na primer, algoritmi za sabiranje i za množenje prirodnih brojeva, za određivanje najmanjeg elementa niza, za uređivanje elemenata po veličini i slično. U okviru razmatranja pojma algoritma i programa, razmatraju se samo postupci kojima se od nekih podataka dobijaju nekakvi rezultati, tj. novi podaci (ne razmatraju se postupci u kojima je potrebno obavljati neku fizičku radnju i slično). Pošto se svi podaci digitalizacijom mogu zapisati brojevima (možda uz neki gubitak), dovoljno je razmatrati algoritme za izračunavanje funkcija čiji su i argumenti i rezultujuće vrednosti prirodni brojevi. Pitanja šta može a šta ne može da uradi računar svode se na pitanja šta se može a šta ne može izračunati algoritamski.

Pitanje šta može da se izračuna algoritamski i od strane mašine, postavljala su se i pre pojave prvih računara. Ljudi su hiljadama godina pravili sprave koje su pomagale u računanju. Jedan od najvećih koraka napravio je Gotfrid Lajbnic još u XVII veku kada je napravio mehaničku mašinu za računanje koja je podržavala sve četiri osnovne računske operacije. Lajbnic je verovao da će biti moguće napraviti mašinski postupak koji će, manipulisanjem simbolima, biti u stanju da dâje odgovor na sva matematička pitanja. Koristeći osnovne operacije izračunavanja, mogu se opisati postupci za rešavanje složenijih matematičkih problema. Takvi postupci postojali su još u vreme starogrčkih matematičara (na primer, Euklidov algoritam za određivanje najvećeg zajedničkog delioca dva broja), pa i pre toga.

4.1 Prirodno-jezički opisi algoritama i izračunavanja

Svaki algoritam sačinjen je od operacija koje treba izvršiti da bi se rešio neki zadatak. To je još uvek daleko od prave definicije. Krenimo sa prirodno-jezičkim opisima algoritama i izračunavanja.

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 promenljivoj y treba da bude dodeljena vrednost $2x + 3$, onda se to može opisati na sledeći način (gde simbol `*` označava množenje, a `+` sabiranje):

[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".

Elektronska verzija (2024)

```
dodeli promenljivoj y vrednost 2*x + 3
```

Kao naredni primer, razmotrimo postupak ili algoritam za maksimuma dva data broja. Pretpostavimo da promenljive x i y sadrže dve date brojevne vrednosti, a da promenljiva m treba da dobije vrednost veće od njih. Nakon sledećeg postupka promenljiva m ima vrednost većeg od zadata dva broja:

```
ako je x >= y onda
   dodeli promenljivoj m vrednost x
inače
   dodeli promenljivoj m vrednost y
```

Kao malo komplikovaniji primer razmotrimo stepenovanje. Na primer, n-ti stepen broja x (tj. vrednost x^n) moguće je izračunati uzastopnom primenom množenja: ako se krene od vrednosti 1 i ona se 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 dodatna 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:

```
dodeli promenljivoj s vrednost 1
dodeli promenljivoj i vrednost 0
dok je i < n radi sledeće:
   dodeli promenljivoj s vrednost s*x
   dodeli promenljivoj i vrednost i+1
```

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

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

4.2 Programi i izračunavanja na višem programskom jeziku

Na prvim elektronskim 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.

Algoritmi i izračunavanja koja su u prethodnom delu opisana na prirodnom jeziku mogu da se pretoče u programe – konkretne implementacije – na višem programskom jeziku kao što je, na primer, C ili C++:

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

Primetimo da u navedenom kodu, simbol `=` ne označava jednakost nego operaciju dodele. U drugim jezicima ovo izračunavanje opisuje se na slične načine. Naglasimo da navedeno izračunavanje mora da bude deo neke malo šire celine kako bi moglo da se izvrši (na primer, mora da se navede kojoj vrsti brojeva pripadaju `x` i `y`, promenljiva `x` na neki načina mora da dobije svoju vrednost, itd). I u naredna dva primera implementacije na jeziku C++ slične su prirodno-jezičkim opisima:

```
if (x >= y) {
   m = x;
}
else {
   m = y;
}
```

```
s = 1;
i = 0;
while (i < n) {
   s = s*x;
   i = i+1;
}
```

4.3 Izračunavanja i programi na mašinski zavisnim jezicima

U prethodnom delu izračunavanja opisana na prirodnom jeziku i na jeziku C/C++ jasna su čoveku, čak i onom koji nije nikad programirao. Međutim, takvi opisi nisu razumljivi računaru. Računar može da „razume" i izvrši samo programe napisane na mašinskom jeziku. Kao što je rečeno, na prvim elektronskim računarima moglo je da se programira samo na mašinskom jeziku.

Procesor je centralni deo računara i on podržava izvesne naredbe, tj. *primitivne instrukcije* koje su hardverski implementirane i koje su sve veoma jednostavne. Postoje, na primer, instrukcije za sabiranje dva broja, za množenje dva broja, instrukcije za poređenje dva broja, instrukcije za konjunkcija bitova, instrukcija skoka na neki korak u programu i slično.

Svaki program na mašinskom jeziku sastavljen je od niza procesorskih primitivnih instrukcija. Naredbe mogu da se izvršavaju redom, jedna za drugom, a kada je potrebno da se neke naredbe ponove veći broj puta ili da se određene naredbe preskoče, koriste se *naredbe skoka*. Procesor ima određeni broj *registara*, memorijskih jedinica u koje može neposredno da upisuje i iz kojih može neposredno da čita podatke. Sve instrukcije na mašinskom jeziku zapisuju se u vidu brojeva, koji su u računaru zapisani u binarnom sistemu (tj. u obliku niza nula i jedinica). Da bi mogao da se izvrši na računaru, program na mašinskom jeziku mora da budu smešten u memoriju, u vidu niza brojeva koji odgovaraju nizu naredbi.

Svi zadaci koje računari izvršavaju svode se na ovakve programe, tj. na nizove primitivnih instrukcija procesora. Kompleksni algoritmi implementirani na ovakvom jeziku mogu se sastojati od ogromnog broja primitivnih instrukcija, te je programiranje na mašinskom jeziku veoma zahtevno i naporno.

Procesori različitih računara mogu da se razlikuju (na primer, po tome koliko registara u procesoru imaju, koje instrukcije može da izvrši njihova aritmetičko-logička jedinica, itd). Razvoj najvećeg broja procesora usmeren je tako da većina savremenih procesora ima veoma slične skupove instrukcija, a isti mašinski programi mogu se koristiti na čitavim familijama procesora.

4.3.1 Programi na asembleru

Asemblerski jezik je jezik koji je veoma blizak mašinskom jeziku računara, ali se, umesto brojevnog zapisa za instrukcija koriste (mnemotehničke, lako pamtljive) simboličke oznake instrukcija (tj. programi se zapisuju u vidu teksta). Da bi ovako napisan program mogao da se izvršava, neophodno je izvršiti njegovo prevođenje na mašinski jezik (tj. zapisati instrukcije u vidu brojeva, i to binarnom azbukom) i uneti na odgovarajuće mesto u memoriji. Ovo prevođenje je trivijalno i jednoznačno i vrše ga programi koji se nazivaju *asembleri*. Korišćenjem asemblera olakšava se programiranje, ali skoro sva zahtevnost programiranja na mašinskom jeziku prisutna je i dalje.

Pretpostavimo da naš hipotetički procesor sadrži, između ostalog, 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.

Zarad specifikovanja instrukciju na koju se vrši skok, koriste se *labele* – označena mesta u programu. Pretpostavimo da naš procesor može da izvršava, između ostalog, sledeće dve vrste skokova (bezuslovne i uslovne):

Elektronska verzija (2024)

- 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`.

Opišimo za ovakav procesor izračunavanje vrednosti $2x+3$ (podsetimo se da izračunavanje mora da se razloži na primitivne operacije). 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 maksimuma 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).

Elektronsko izdanje (2024)

Procesori skoro uvek podržavaju instrukcije kojima se izračunavaju zbir i proizvod dva cela broja, ali stepenovanje obično nije podržano kao primitivna instrukcija. Izračunavanje vrednosti x^n 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 koji implementira algoritam za stepenovanje opisan u prethodnom delu (i koji odgovara opisu iz poglavlja i) 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
```

4.3.2 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.

Elektronska verzija

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 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 hipotetičkim instrukcijama navedenim ovde za ilustraciju.

4.4 Zasnivanje pojma algoritma

Kroz nekoliko navedenih primera ilustrovali smo pojam algoritma, ali da bi se pitanje šta se uopšte može izračunati algoritamski i mnoga slična mogla precizno razmatrati, neophodno je najpre definisati (matematički rigorozno) šta je to algoritam. Iako su razni algoritmi razvijani vekovima, do početka dvadesetog veka retko se postavljalo pitanje kako formalno zasnovati pojam algoritma i koji su sve problemi mašinski, algoritamski rešivi. Prve precizne definicija algoritma ipak su prethodile prvim računarima i date su početkom dvadesetog veka, u jeku reforme i novog utemeljivanja matematike. Jedno od fundamentalnih pitanja postavljenih tada bilo je pitanje da li postoji algoritam kojim se (pojednostavljeno rečeno) mogu dokazati sve matematičke teoreme.

Algoritam možemo opisati kao bilo koji postupak koji može da se izvrši na računaru, preciznije na njegovom procesoru (ne razmatramo postupke koji uključuju periferije: emitovanje zvuka ili slike, štampanje i slično). Time dolazimo do pitanja šta očekujemo od jednog računara ili procesora da može da izvrši kao svoje osnovne operacije. Međutim, nisu svi procesori isti, pa se ovaj pristup komplikuje - možemo onda da razmatrmo izračunavanja koja mogu da se izvrše da proizvoljnom računaru. S jedne strane, želimo da naš pojam algoritma pokriva sve što može da se uradi na savremenom računaru, a s druge – da taj pojam bude što jednostavniji, kako bi bio podesniji za formalna razmatranja.

Očigledno je da očekujemo da sabiranje brojeva može i treba da se smatra algoritamskim postupkom i da (proizvoljni) procesor može da ga izvršava. Ne samo da je sabiranje očigledno operacija potrebna u mnogim problemima, nego je ova operacija jasno opisiva. Zaista, algoritam za sabiranje brojeva uči se još u prvom razredu osnovne škole. Dakle, od (svakog) procesora očekujemo da ume da sabira brojeve. Slično, kako je množenje brojeva važno i precizno opisivo, od (svakog) procesora očekujemo da ume da množi brojeve. Međutim, pitanje je da li od procesora očekujemo da ume da neposredno računa, na primer, determinantu matrice. Odgovor je da ne mora, pošto se to računanje može svesti na množenja i sabiranja (naravno, u tome pretpostavljamo da procesor može da se instruira i da ponavlja neke radnje pod nekim uslovima). Slično, kao što smo videli u prethodnom delu, nije neophodno da procesor kao osnovnu operaciju ima stepenovanje prirodnih brojeva, jer se ono može svesti na množenje. Zapravo, uviđamo da onda od procesora ne treba da očekujem da ume neposredno ni da množi, jer se množenje može svesti na sabiranje. Zaista, proizvod prirodnih brojeva x i y jednaj je zbiru $x + x + \ldots + x$ u kojem se vrednost x pojavljuje y puta. Ali, onda možemo ići i još dalje: nije nam potrebno ni sabiranje sve dok naš procesor ume da nekoj vrednosti dodaje vrednost 1 pod nekim uslovom. Zaista, zbir prirodnih brojeva x i y jednak je zbiru $x + 1 + \ldots + 1$ u kojem se vrednost 1 pojavljuje y puta. Takvim razmatranjem dolazimo do opisa veoma jednostavnog procesora koji će poslužiti za definisanje pojma algoritma.

4.4.1 UR mašine

UR mašina je apstraktna mašina koja ne postoji u fizičkom obliku ali predstavlja matematičku idealizaciju računara i omogućava zasnivanje pojma algoritma[2]: reći ćemo da je algoritam onaj i samo onaj postupak koji može da se opiše kao program za UR mašinu.

UR mašina raspolaže beskonačnim skupom memorijskih lokacija – *registara*, koji služe kao prostor za čuvanje (brojevnih) podataka. Registri su označeni prirodnim brojevima: $, 2, 3, \ldots$. Svaki od njih u svakom trenutku sadrži neki prirodan broj. Stanje registara u nekom trenutku zovemo *konfiguracija*. Sadržaj k-tog registra označava se sa r_k, kao što je to ilustrovano na sledećoj slici:

r_1	r_2	r_3	$\ldots$

U prethodnom delu teksta, naslutili smo da je idealizovanom računaru potreban i dovoljan mali broj operacija. Tako je dizajnirana i UR mašina – ona podržava:

- operaciju upisivanja nule u memorijsku lokaciju;
- operaciju dodavanja jedinice na sadržaj memorijske lokacije;
- kopiranje sadržaja sa jedne u drugu lokaciju;

[2]Prvi opis dat je u jednom radu Šeperdsona i Sturdžisa (engl. Sheperdson and Sturgis) iz 1963. godine.

- operaciju skoka na određeno mesto u programu ukoliko je sadržaj dve lokacije jednak.

Zapis i kratak opis URM instrukcija (naredbi) dati su u tabeli 7.1.[3] U tabeli, na primer, $r_m := 0$ označava da se vrednost 0 upisuje u m-ti registar, a $r_m := r_m + 1$ označava da se sadržaj m-tog registra uvećava za jedan.

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

Tabela 4.1: 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 od početnog redom. 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. Početne vrednosti u registrima iza njih nisu poznate unapred. 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 $P(a_1, a_2, \ldots, a_n) \uparrow$. 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 4.1. *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:

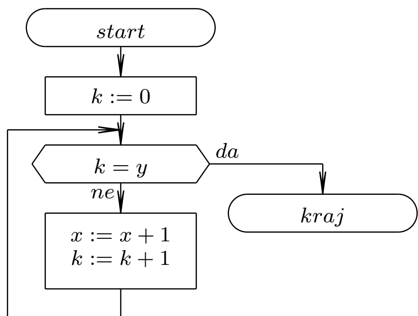
x	y	...	...

i sledeću konfiguraciju u toku rada programa:

$x+k$	y	k	...

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

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



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

Elektronska verzija (2024)

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[4]. *Bezuslovni skok je realizovan naredbom oblika* $J(1, 1, \ldots)$ *— poređenje registra sa samim sobom uvek garantuje jednakost te se skok vrši uvek.*

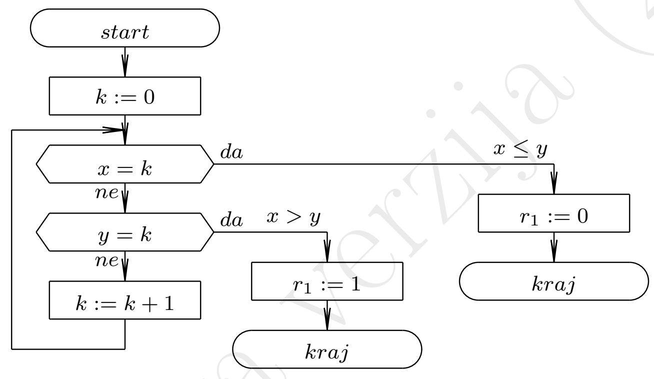
Primer 4.2. *Neka je funkcija f definisana na sledeći način:*

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

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

x	y	k	...

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.	$Z(1)$	$r_1 := 0$
7.	$J(1, 1, 100)$	*kraj*
8.	$Z(1)$	
9.	$S(1)$	$r_1 := 1$

Navedena dva primera ilustruju dva izračunavanja nad prirodnim brojevima. Možemo se sada zapitati da li se nekako mogu vršiti i izračunavanja koja se odnose na brojeve koji nisu prirodni, pa ni celi, posebno imajući u vidu da neki brojevi nemaju konačan decimalni zapis, na primer, broj $\sqrt{2}$. Iako taj broj nije ceo, možemo, na primer, zahtevati da se izračuna njegova k-ta cifra. Pokazaćemo, bez upuštanja u detalje, da to jeste izračunljivo. Koristeći osobinu $n = [\sqrt{x}] \Leftrightarrow n^2 \leq x < (n+1)^2$, može se pokazati da je izračunljiva naredna funkcija:

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

Pošto je izračunljiva funkcija f, očigledno je izračunljiva i funkcija $g(k) = [10^k \cdot \sqrt{2}]$. Poslednja cifra vrednosti $g(k)$ (što je takođe izračunljivo) jednaka je k-toj cifri broja $\sqrt{2}$. Ovo ilustruje način na koji možemo vršiti izračunavanja koja se ne odnose na prirodne brojeve.

Primer 4.3. *Neka je funkcija f definisana na sledeći način:*

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

[4]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.

Elektronska verzija (2024)

Nju izračunava naredni program (nedefinisanost funkcije f postiže se time što se program izvršava beskonačno izuzev ako je vrednost argumenta jednaka 0*):*

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

4.4.2 Enumeracija URM programa

Važno pitanje je koliko uopšte ima različitih URM programa. Očigledno je da ih ima beskonačno, ali postoje razne vrste beskonačnosti. 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*. Skup realnih brojeva nije prebrojiv, a postoje i mnogi skupovi koji imaju mnogo veće kardinalnosti.

Primer 4.4. *Skupovi parnih i neparnih brojeva imaju istu kardinalnost jer je između njih moguće uspostaviti bijektivno preslikavanje $f(n) = n + 1$. Štaviše, oba ova skupa su prebrojiva, tj. imaju istu kardinalnost kao i skup prirodnih brojeva, iako su njegovi pravi podskupovi (na primer, funkcija $f(n) = 2 \cdot n$ uspostavlja bijekciju između skupa prirodnih i skupa parnih brojeva).*

Primetimo da se u zapisu URM programa koriste samo sledeći simboli: Z, S, T, J, (,), , i deset cifara za zapis brojeva. Zato se može napisati samo konačno mnogo programa koji imaju ukupno k simbola, $k = 1, 2, 3, ...$ (zapravo, za neke dužine, kao što su 1 ili 2 taj broj je jednak nuli). Dalje, zbog toga možemo u niz poređati sve URM programe: najpre sve one dužine 1 (recimo po alfabetskom redu), pa onda dužine 2, pa dužine 3, itd. Odatle sledi naredna teorema.

Teorema 4.1. *Različitih* URM *programa ima prebrojivo mnogo*

Na osnovu navedene teoreme, 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. Za bilo koju vrednost i, smatramo da je funkcija koju izračunava program P_i algoritamski izračunljiva. I obratno, ako je neka funkcija algoritamski izračunljiva onda mora da postoji broj i, takav da program P_i izračunava tu funkciju. Na sličan način može se pokazati da i na bilo kom programskom jeziku ima prebrojivo mnogo programa.

4.4.3 Drugi pristupi zasnivanju pojma algoritma

U prethodnom tekstu, pojam izračunljivosti uveden je na bazi UR *mašina (*URM*)*. Taj formalizam je elegantan, intuitivan i blizak savremenom programiranju, ali nije jedini. Posebno tokom 1920-ih i 1930-ih godina više velikih matematičara bavilo se zasnivanjem pojma algoritma i izračunljivosti i oni su razvili više raznorodnih formalizama. Najznačajniji među njima verovatno su: *Tjuringove mašine* (Tjuring), *rekurzivne funkcije* (Gedel[5] i Klini[6]) i *λ-račun* (Čerč[7]). *Blok dijagrami* (kaže se i *algoritamske šeme*, *dijagrami toka*, tj. *tokovnici*), kao oni prikazani u primerima 4.1 i 4.2, mogu se smatrati poluformalnim načinom opisa algoritama.

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 moćan da izvrši sva izračunavanja koja može da izvrši Tjuringova mašina kaže se da je *Tjuring potpun* (engl. *Turing complete*).

Iako su pomenuti formalizmi međusobno veoma različiti, može se rigorozno 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 potpuni i ekvivalentni. Zbog toga se, umesto pojmova poput, na primer, Tjuring-izračunljiva ili URM-izračunljiva funkcija (i sličnih) može koristiti samo termin *izračunljiva funkcija*. Konačno, možemo reći da je algoritam svaki postupak kojim se izračunava neka izračunljiva funkcija.

[5]Kurt Gödel (1906-1978), austrijsko-američki matematičar.
[6]Stephen Kleene (1909-1994), američki matematičar.
[7]Alonzo Church (1903-1995), američki matematičar.

Elektronska verzija (2024)

4.4.4 Savremeni računari i izračunljivost

Sada, kada nam je poznat formalni pojam algoritma, možemo da se zapitamo da li savremeni računari mogu da izvrše svaki algoritam, što bi bilo očekivano. Savremeni računari (tj. savremeni procesori koji su centralni deo računara) očigledno mogu da izvrše sve instrukcije koje ima UR mašina, pa mogu da izračunaju sve funkcije koje mogu da izračunaju nabrojani formalizmi za izračunavanje, tj. sve izračunljive funkcije. Međutim, ne mogu da izračunaju ništa što ne mogu ti formalizmi! Štaviše, striktno govoreći, njihova moć je i manja: svi navedeni formalizmi za izračunavanje podrazumevaju, pojednostavljeno rečeno, beskonačnu raspoloživu memoriju. Zbog toga savremeni računari (koji raspolažu konačnom memorijom) nisu Tjuring potpuni, mada mogu da izrše sve URM programe koji ne zahtevaju beskonačnu memoriju.

Savremeni računar ima procesor koji sadrži ugrađene komponente za sabiranje, oduzimanje, množenje, deljenje, poređenje celih brojeva i brojeva u pokretnom zarezu. Takvim operacijama odgovaraju instrukcije procesora, a on ima još i instrukcije koje vrše skok na određenu instrukciju u programu. Dakle, procesor savremenog računara nije suštinski mnogo bogatiji od UR mašine. Zapravo, mogao bi da se napravi procesor (i računar oko njega) koji može da izvršava samo URM instrukcije. To bi bilo dovoljno da računar može da radi bilo šta što može koristeći neki savremeni procesor. Takav URM procesor bio bi sigurno znatno jeftiniji (jer je znatno jednostavniji), ali bi ipak bio nepraktičan i neefikasan jer bi se, na primer, operacije množenja velikih brojeva svodile na ogroman broj dodavanja vrednosti 1.

Sledeće važno pitanje je da li nekakvi drugačiji računari mogu da izračunaju nešto što ne može UR mašina. Na primer, kvantni računari imaju bitno drugačiju arhitekturu od sada dominantnih računara. Oni neka izračunavanja mogu da izvrše znatno brže nego obični računari, ali i dalje ne mogu da izračunaju ništa što ne mogu i obični računari. Trenutno ne postoji računar niti arhitektura za (fizički, ne apstraktni) računar koja omogućava izračunavanje koje ne omogućava UR mašina.

4.4.5 Saremeni programski jezici i izračunljivost

Na *savremenim programskim jezicima* takođe se mogu implementirati svi algoritmi, te ih je moguće pridružiti navedenoj listi pristupa zasnivanja algoritama. Zaista, za veliku većinu savremenih programskih jezika važi da su Tjuring-kompletni. 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 (a u cilju jednostavnije formalne analize), 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). [8]

4.4.6 Čerč-Tjuringova teza

Pominjane formalne definicije izračunljivosti i algoritama su ekvivalentne i to je moguće rigorozno dokazati. Veoma važno pitanje je i koliko formalne definicije algoritama uspevaju da pokriju naš intuitivni pojam algoritma, tj. da li čovek zaista može efektivno izvršiti sva izračunavanja definisana nekom od formalizacija izračunljivosti i, obratno, da li sva izračunavanja koja čovek intuitivno ume da izvrši zaista mogu da budu opisana korišćenjem bilo kog od precizno definisanih formalizama izračunavanja. Očigledno 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 (jer formalno, matematički ne mogu biti opisani ni procesi koji se odvijaju u ljudskom mozgu). U korist ove teze govori činjenica da do sada nije pronađen nijedan primer intuitivno, efektivno izvodivog postupka izračunavanja koji nije moguće formalizovati u okviru nabrojanih formalnih sistema izračunavanja. Ovim dolazimo do uverenja da je skup funkcija koje može da izračuna čovek jednak skupu funkcija koje može da izvrši bilo koji savremeni računar i jednak je skupu funkcija koje može da izvrši UR mašina (ovde pretpostavljamo da za izračunavanja nije potrebno beskonačno memorijskih

Elektronska verzija (2021)

[8] Naredba skoka u programima može da dovodi do nečitljivih i nerazumljivih (tzv. špageti) programa. Dokazano je da ona i nije neophodna i da je dovoljno da programski jezik podržava samo sekvencijalno nizanje naredbi, naredbu izbora (if-then-else) i barem jednu vrstu petlje (na primer, do-while). Ovo tvrdenje je poznato kao teorema o strukturnom programiranju, Korado Bema (nem. Corrado Böhm) i Duzepea Jakopinija (it. Giuseppe Jacopini) iz 1966. godine.

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

lokacija). U ovoj vezi, naravno, potpuno zanemarujemo pitanje dužine trajanja izračunavanja koje vrši čovek i koje vrše mašine.

4.4.7 Neizračunljivost i neodlučivost

Algoritmima se, za zadate argumente, izračunavaju neke vrednosti, tj. algortimima se opisuju izračunljive funkcije. Veliko pitanje je da li postoje funkcije nad brojevima koje su jasno definisane i totalne (definisane za sve vrednosti argumenata) a nisu izračunljive. Drugim rečima, postoje li problemi koji ne mogu da se reše algoritamski. Razmotrimo narednih nekoliko problema.

1. Već pomenuto pitanje da li postoji algoritam kojim se (pojednostavljeno rečeno) mogu dokazati sve matematičke teoreme može se takođe može svesti na izračunavanje neke funkcije. Ovo pitanje može se formulisati i kao pitanje da li postoji algoritam koji za proizvoljni zadati skup aksioma i zadato tvrđenje proverava da li je tvrđenje posledica aksioma. Ovaj problem, poznat pod imenom „Entscheidungsproblem“, postavio je David Hilbert 1928. godine. Ipak, 1930-ih, rezultatima Čerča, Tjuringa i Gedela pokazano je da ovakav postupak ne može da postoji. Međutim, pre ovog znamenitog rezultata trebalo je opisati šta se uopšte može smatrati algoritmom.

2. 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.[10]

3. 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.[11]

4. Zadatak je konstruisati opšti algoritam koji proverava da li se proizvoljni zadati program P zaustavlja za date ulazne parametre.[12]

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[13], već samo da ne postoji jedinstven, opšti postupak koji bi mogao da reši proizvoljnu instancu problema. Pored nabrojanih, ima još mnogo važnih neodlučivih problema. Poznavanje neodlučivih problema u informatici je veoma važno i može se uporediti sa poznavanjem činjenice iz fizike i mašinstva da nije moguće napraviti *perpetuum mobile* – mašinu koja proizvodi jednako ili više energije nego što je njoj predato. Kao što je besmisleno pokušavati napraviti perpetuum mobile, tako je besmisleno pokušavati napraviti program koji rešava neki od nerešivih problema.

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

4.5 Zaustavljanje programa i 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:

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

[11] 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.

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

[13] 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.

Elektronska verzija (2024)

Halting problem: *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?*

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 4.4.2). Imajući ovo u vidu, dolazi se do teoreme o halting problemu za URM.

Teorema 4.2 (Neodlučivost halting problema)**.** *Neka je funkcija h definisana na sledeći način:*

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

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 4.3) 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:

$$\begin{aligned} Q(x) \downarrow 0 \;\; &\text{ako je } \; P_x(x) \uparrow \\ Q(x) \uparrow \;\; &\text{ako je } \; P_x(x) \downarrow \end{aligned}$$

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:

$$\begin{aligned} P_k(x) \downarrow 0 \;\; &\text{ako je } \; P_x(x) \uparrow \\ P_k(x) \uparrow \;\; &\text{ako je } \; P_x(x) \downarrow \end{aligned}$$

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:

$$\begin{aligned} P_k(k) \downarrow 0 \;\; &\text{ako je } \; P_k(k) \uparrow \\ P_k(k) \uparrow \;\; &\text{ako je } \; P_k(k) \downarrow \end{aligned}$$

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. □

Neodlučivost halting problema ne odnosi se samo na URM programe, već i na bilo koji savremeni, Tjuring-kompletan programski jezik (uključujući C, C++, Java, itd).

Halting problem je neodlučiv, tj. ne postoji opšti postupak kojim se za proizvoljni zadati program može utvrditi da li se on zaustavlja za zadate vrednosti argumenata. Ipak, za mnoge konkretne programe, može se utvrditi da li se zaustavljaju ili ne. Kako ne postoji opšti postupak koji bi se primenio na sve programe, zaustavljanje svakog programa mora se ispitivati zasebno i koristeći specifičnosti tog programa.

U programima u kojima su petlje jedine naredbe koje mogu dovesti do nezaustavljanja potrebno je dokazati zaustavljanje svake pojedinačne petlje. Ovo se obično radi tako što se definiše dobro zasnovana relacija[14] takva

[14] Za relaciju $\succ$ se kaže da je *dobro zasnovana* (engl. well founded) ako ne postoji beskonačan opadajući lanac elemenata $a_1 \succ a_2 \succ \ldots$.

Elektronska verzija (2

da su susedna stanja kroz koje se prolazi tokom izvršavanja petlje međusobno u relaciji. Kod elementarnih algoritama ovo se obično radi tako što se izvrši neko preslikavanje skupa stanja u skup prirodnih brojeva i pokaže da se svako susedno stanje preslikava u manji prirodan broj.[15] Pošto je relacija $>$ na skupu prirodnih brojeva dobro zasnovana, i ovako definisana relacija na skupu stanja biće dobro zasnovana.

Primer 4.5. *Algoritam koji vrši množenje uzastopnim sabiranjem (poglavlje 4.3.1) se zaustavlja. Zaista, u svakom koraku petlje vrednost $n - i$ je prirodan broj. Ova vrednost opada kroz svaki korak petlje (jer se n ne menja, a i raste), pa u jednom trenutku mora da dostigne vrednost* 0.

Primer 4.6. *Ukoliko se ne zna gornje ograničenje za polaznu vrednost n, nije poznato da li se naredna petlja uvek zaustavlja:*

```
while (n > 1) {
   if (n % 2)
      n = 3*n + 1;
   else
      n = n/2;
}
```

Opšte uverenje je da se funkcija zaustavlja za svaku ulaznu vrednost n (to tvrdi još uvek nepotvrđena Kolacova (Collatz) hipoteza *iz 1937). Navedeni primer pokazuje kako pitanje zaustavljanja čak i za neke veoma jednostavne programe može da bude ekstremno komplikovano.*

Naravno, ukoliko je poznata širina podatka `unsigned int`*, i ukoliko se testiranjem za sve moguće ulazne vrednosti pokaže da se* `f` *zaustavlja, to bi dalo odgovor na pitanje u specijalnom slučaju.*

4.6 Algoritmika i 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). 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.

Primer 4.7. *Razmotrimo problem izračunavanja vrednosti x^n (za nenegativnu vrednost n). U poglavlju 4.2 već smo videli programski kôd koji izračunava traženu vrednost:*

```
s = 1;
i = 0;
while (i < n) {
   s = s*x;
   i = i+1;
}
```

Nije teško videti da će naredni kôd da izvrši upravo n množenja. Možemo se zapitati da li se tražena vrednost može izračunati i efikasnije. Primetimo sledeće: ako je vrednost n parna, onda je $x^n = (x^2)^{n/2}$ pa, na primer, ako je n jednako 6, vrednost u izračunavanju može da se koristi veza $x^6 = (x^2)^3$. U izračunavanju vrednosti x^2 koristi se jedno množenje, a onda u izračunavanju $(x^2)^3$ još dva, ukupno tri, što je znatno nego šest koliko bi koristio početni način. Na ovoj ideji zasniva se sledeći kôd za izračunavanja vrednosti x^n:

```
s = 1;
while (n > 0) {
   if (n % 2 == 0) {
      x = x*x;
      n = n/2;
   } else {
      s = s*x;
      n = n-1;
   }
```

[15] Smatramo da i nula pripada skupu prirodnih brojeva.

```
}
```

Može se pokazati da navedeni kôd u izračunavanju x^{16} koristi samo četiri množenja, a u izračunavanju x^{1024} samo deset. Generalno, u izračunavanju x^n koristi oko $\log_2 n$ množenja, što je za velike vrednosti n mnogo bolje nego polazni, naivni algoritam.

Navedeni primer ilustruje teme kojima se bavi oblast informatike koja se zove *algoritmika*: analizom vremenske i prostorne složenosti programa, kao i dizajnom što efikasnijih algoritama koji rešavaju neki problem.

4.7 Pregled

- Svi podaci koje se mogu obrađivati na računaru predstavljeni su brojevima. Računar svaku obradu podataka svodi na niz instrukcija koje vrše izračunavanje nad brojevima (ili vrše skok na neku instrukciju u nizu).
- Za računar je, suštinski, dovoljno da može da obavi svega nekoliko, veoma jednostavnih instrukcija.
- Neformalno, algoritam je precizan opis postupka za rešavanje nekog problema u konačnom broju koraka. Računarski program je konkretna implementacija nekog algoritma u nekom konkretnom programskom jeziku.
- Formalno, algoritam je svaki postupak koji se može opisati na mašini kao što je URM.
- Savemeni programski jezici mogu da služe i kao formalizam za opisivanje pojma algoritma.
- Bilo koji savremeni računar može da uradi ono i samo ono što može UR. Preciznije - može i manje, jer ima samo konačno mnogo memorijskih lokacija na raspolaganju.
- Bilo koji savremeni računar može da uradi ono i samo ono što može da izračuna i čovek – to govori Čerč-Tjuringova teza (koja se ne dokazuje). U ovoj vezi zanemaruje se brzina izvršavanja pojedinačnih operacija.
- Postoje totalne, precizno definisane funkcije nad brojevima koje se mogu i one koje se ne mogu izračunati algoritamski, na računaru.
- Mnogi važni problemi ne mogu se rešiti algoritamski. Jedan od njih je da ne postoji program koji za proizvoljni zadati program može da utvrdi da li taj program staje sa radom za neki zadati argument.
- Algoritmika je oblast informatike koja se bavi analizom vremenske i prostorne složenosti programa, kao i dizajnom što efikasnijih algoritama koji rešavaju neki problem.

Pitanja i zadaci za vežbu

Pitanje 4.1. *Po kome je termin* algoritam *dobio ime?*

Pitanje 4.2. *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 4.3. *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 4.4. *Da li je svaka* URM *izračunljiva funkcija intuitivno izračunljiva? Da li je svaka intuitivno izračunljiva funkcija* URM *izračunljiva?*

Pitanje 4.5. *U čemu je ključna razlika između* URM *mašine i bilo kog stvarnog računara?*

Pitanje 4.6. *Opisati efekat* URM *naredbe $J(m, n, p)$.*

Pitanje 4.7. *Da li se nekim* URM *programom može izračunati hiljadita cifra broja 2^{1000} ?*

Pitanje 4.8. *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?*

tronska verzija (2024)

Pitanje 4.9. *Da li se nekim* URM *programom može izračunati hiljadita decimalna cifra broja* π *?*

Pitanje 4.10. *Koliko ima racionalnih brojeva? Koliko ima kompleksnih brojeva? Koliko ima različitih programa za Tjuringovu mašinu? Koliko ima različitih programa u programskom jeziku C?*

Pitanje 4.11. *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 4.12. *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 4.13. *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 4.14. *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 4.15. *Kako se naziva problem ispitivanja zaustavljanja programa? Kako glasi* halting *problem? Da li je on odlučiv ili nije? Ko je to dokazao?*

Pitanje 4.16.

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

Zadatak 4.1. *Napisati* URM *program koji izračunava funkciju* $f(x, y) = xy$. √

Zadatak 4.2. *Napisati* URM *program koji izračunava funkciju* $f(x) = 2^x$.

Zadatak 4.3. *Napisati* URM *program koji izračunava funkciju* $f(x, y) = x^y$.

Zadatak 4.4. *Napisati* URM *program koji izračunava funkciju:*

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

Zadatak 4.5. *Napisati* URM *program koji izračunava funkciju*

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

√

Zadatak 4.6. *Napisati* URM *program koji izračunava funkciju:*

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

Zadatak 4.7. *Napisati* URM *program koji izračunava funkciju* $f(x) = x!$.

Zadatak 4.8. *Napisati* URM *program koji izračunava funkciju* $f(x) = \left[\frac{2x}{3}\right]$.

Zadatak 4.9. *Napisati* URM *program koji broj* 1331 *smešta u prvi registar.*

Zadatak 4.10. *Napisati* URM *program koji izračunava funkciju* $f(x) = 1000 \cdot x$.

Zadatak 4.11. *Napisati* URM *program koji izračunava funkciju* $f(x, y) = 2x + y$.

Zadatak 4.12. *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 4.13. *Napisati* URM *program koji izračunava funkciju* $f(x, y) = 2^{(x+y)}$

Zadatak 4.14. *Napisati* URM *program koji izračunava funkciju*

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

Zadatak 4.15. *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 4.16. *Napisati* URM *program koji izračunavaju sledeću funkciju:*

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

Zadatak 4.17. *Napisati* URM *program koji izračunava funkciju:*

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

Zadatak 4.18. *Napisati* URM *program koji izracunava funkciju* $f(x, y, z) = x + y + z$

Zadatak 4.19. *Napisati* URM *program koji izracunava funkciju* $f(x, y, z) = min(x, y, z)$.

Zadatak 4.20. *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 4.21. *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 5

Sistemski softver

Podsetimo se da se računarski sistem sastoji iz *hardvera* i *softvera*. Hardver čine sve fizičke komponente računara, poput procesora, operativne memorije, magistrala, ulazno-izlaznih uređaja i spoljnih memorija. *Softver* čine računarski programi koji se mogu pokretati i izvršavati na tom računaru.

Hardver i softver su od jednake važnosti za računarski sistem, jer su jedan bez drugoga potpuno neupotrebljivi. Računar je, po definiciji, mašina koja izvršava programe – to je sve što računar ume i može da radi. Zbog toga računaru odmah po uključivanju mora biti dostupan program koji bi izvršavao, u suprotnom računar ne bi mogao da radi ništa.

Softver može biti *aplikativni* i *sistemski*. Pod aplikativnim softverom podrazumevamo programe koji su razvijeni sa ciljem primene u različitim oblastima. Drugim rečima, to su oni programi koji se koriste da korisniku reše neki problem ili mu pruže neku uslugu. U aplikativni softver spada npr. kancelarijski softver (editori i procesori teksta, programi za tabelarna izračunavanja i sl.), multimedijalni softver (reprodukcija i obrada slika, zvuka i videa, vektorska grafika i sl.), komunikacioni softver (veb pregledači, programi za udaljeni pristup, mejl klijenti i sl.), matematički i naučni softver (programi za numerička i simbolička izračunavanja, statistički softver i sl.), razvojni softver (programski prevodioci, debageri, profajleri, sistemi za održavanje i kontrolu verzija softvera i drugi programerski alati).

Dakle, aplikativni softver čine svi oni programi koji su nama potrebni da bismo mogli da primenjujemo računare u različitim oblastima. Ipak, pokretanje i zaustavljanje ovakvih programa, kao i bezbedna i kontrolisana upotreba različitih računarskih resursa od strane aplikativnih programa tokom njihovog izvršavanja nije ni malo jednostavan posao. Da bi korisnik mogao da pokreće svoje aplikativne programe, upravlja njihovim radom, bezbedno pristupa svojim podacima na spoljnim memorijama, kao i drugim resursima računara, potrebni su posebni programi koji mu to omogućavaju. Skup takvih programa nazivamo *sistemskim softverom*.

Najznačajnija komponenta sistemskog softvera je *operativni sistem*. Njega čini skup programa koji obezbeđuju efikasnu i bezbednu kontrolu računarskih resursa. U ove resurse ubrajamo kako hardverske komponente (procesor, memoriju, ulazno-izlazne uređaje, spoljne memorije, komunikacioni hardver i sl.), tako i podatke sačuvane u spoljnim memorijama. Pored upravljanja ovim resursima, operativni sistem obezbeđuje mehanizme korišćenja tih resursa od strane aplikativnih programa. Najzad, operativni sistem omogućava komunikaciju korisnika sa računarom posredstvom ulazno-izlaznih uređaja. Ovo podrazumeva upotrebu podataka, kao i pokretanje i zaustavljanje aplikativnih programa.

Pored operativnog sistema, u sistemski softver spadaju i drugi programi poput antivirusnih programa, alata za održavanje, konfiguraciju i optimizaciju sistema, programi za učitavanje operativnog sistema, ugrađeni programi poput BIOS-a i različitih firmvera za uređaje i sl. Međutim, podeljena su mišljenja o tome koji od tih programa se mogu smatrati delom operativnog sistema, a koji predstavljaju izdvojene softverske komponente. Takođe, ponekad se i programerski alati poput previodica, linkera i debagera smatraju sistemskim softverom. Isto važi za različite mrežne servere, koji spadaju u komunikacioni softver, ali su obično u nadležnosti administratora sistema. Sa druge strane, pojedine komponente koje se isporučuju sa operativnim sistemom (poput editora teksta, jednostavnih igara, multimedijalnih programa i sl.) se obično smatraju delom operativnog sistema, iako ne spadaju u sistemski softver. Dakle, granica između sistemskog i aplikativnog softvera nije uvek tako oštra i postoje programi koji se mogu smatrati „sivom zonom" između ove dve vrste softvera.

5.1 Operativni sistem

Operativni sistem predstavlja skup programa koji omogućavaju efikasno i bezbedno upravljanje resursima računarskog sistema i stavljanje tih resursa na raspolaganje aplikativnim programima i, posredno, korisniku računara. Komponente operativnog sistema se mogu nalaziti u ROM memoriji (u novije vreme EEPROM) ili u

Elektronska verzija (2024)

spoljašnjim memorijama (danas tipično na hard ili SSD disku), odakle se po uključivanju računara automatski učitavaju u operativnu memoriju (RAM) i započinju sa izvršavanjem na procesoru računara. Operativni sistem dalje omogućava pokretanje aplikativnih programa od strane korisnika (podsredstvom dela operativnog sistema koji se zove *korisnički interfejs*). Ovo znači da je bez operativnog sistema nemoguće efektivno koristiti računar, što operativni sistem čini neophodnom softverskom komponentom svakog računara.

U neka ranija vremena, računari su imali fiksirani, fabrički ugrađeni operativni sistem koji se nalazio u ROM memoriji računara. Sa druge strane, moderni računari obično imaju operativni sistem koji je izmenjiv, jer se nalazi u spoljašnjoj memoriji čiji se sadržaj može menjati. Otuda na većini savremenih računara korisnik ima mogućnost izbora operativnog sistema koji će koristiti, kao i mogućnost nadogradnje verzije operativnog sistema i pratećeg softvera.

U najpoznatije operativne sisteme za desktop i laptop računare danas spadaju operativni sistemi iz serije *Windows*, razvijeni od strane kompanije Majkrosoft (engl. *Microsoft*). U pitanju je vlasnički softver za čiju se upotrebu plaća licenca, uz različita ograničenja koja su tom licencom nametnuta. Alternativno, korisnicima su na raspolaganju operativni sistemi koji spadaju u kategoriju *slobodnog softvera*, što znači da se mogu potpuno slobodno koristiti, distribuirati i modifikovati gotovo bez ikakvih ograničenja (a uz to su u većini slučajeva i potpuno besplatni). U ovakve operativne sisteme uglavnom spadaju sistemi zasnovani na operativnom sistemu *UNIX* (poput sistema *GNU/Linux* i *BSD UNIX*). Pored toga, postoji i aktivan razvoj slobodnih operativnih sistema kompatibilnih sa *Windows* operativnim sistemom (poput *ReactOS* sistema). Sa druge strane, i među UNIX-zasnovanim operativnim sistemima postoje oni sa ne-slobodnom licencom – najpoznatiji primer je *macOS* kompanije Epl (engl. *Apple*).

Kada su u pitanju mobilni uređaji (poput pametnih telefona i tableta), na tržištu dominiraju komercijalni proizvodi poput sistema *Android* kompanije Gugl (engl. *Google*) i *iOS* operativnog sistema kompanije Epl. Takođe postoji i veliki broj *GNU/Linux* distribucija prilagođenih za izvršavanje na ovakvim uređajima, ali je njihov udeo na tržištu veoma ograničen, pre svega zbog neadekvatne podrške proizvođača hardvera za ovakve sisteme.

5.1.1 Struktura operativnog sistema

Operativni sistem je veoma složen softver koji se sastoji iz više celina. Glavni deo operativnog sistema naziva se *jezgro* (engl. *kernel*) koji je zadužen za upravljanje računarskim resursima, komunikaciju sa hardverom, kao i obezbeđivanje interfejsa ka korisničkim programima. Korisnički programi (u koje ubrajamo i aplikativne programe, ali i druge komponente operativnog sistema koje nisu deo jezgra) pristupaju resursima računara isključivo putem *interfejsa jezgra* operativnog sistema pomoću koga od jezgra zahtevaju odgovarajuće usluge. Interfejs jezgra se obično sastoji iz skupa funkcija koje se nazivaju *sistemski pozivi* (engl. *system call*). Sistemski poziv obezbeđuje bezbedan transfer kontrole jezgru operativnog sistema, kako bi ono moglo da korisničkom programu pruži traženu uslugu.

Sistemski pozivi su obično implementirani na nivou asemblerskog jezika. S obzirom da se softver danas uglavnom razvija koristeći programske jezike visokog nivoa, obezbeđuju se različite *sistemske biblioteke* koje omogućavaju programerima da upućuju zahteve jezgru operativnog sistema iz različitih programskih jezika.

Sa druge strane, da bi korisnik mogao da komunicira sa operativnim sistemom, obezbeđen je *korisnički interfejs*. U pitanju je korisnički program (ili skup programa) koji se pokreće odmah nakon što se korisnik prijavi na sistem, a koji omogućava korisniku da izdaje komande, pokreće korisničke programe i prati njihov rad, kao i da koristi podatke na spoljnim memorijama.

Pored toga, uz operativni sistem se obično isporučuju i različiti programi koji obezbeđuju različite servise, poput servisa štampe, mrežnih servisa, servisa za pokretanje periodičnih akcija za održavanje sistema, i sl. Ovi servisi su opcioni i korisnik može konfiguracijom sistema odabrati koji će servisi biti aktivni.

Najzad, uz operativni sistem se obično isporučuje i skup uslužnih programa koji omogućavaju korisniku da jednostavnije upravlja svojim sistemom, obavlja razne administrativne poslove, instalira dodatni softver i sl.

5.1.2 Programi i procesi

U terminologiji operativnih sistema, *proces* predstavlja program u izvršenju. Na taj način razlikujemo programe koji se izvršavaju na računaru od onih koji stoje negde zapamćeni u spoljnoj memoriji i nisu aktivni. Preciznije, proces predstavlja kontekst u kome se program izvršava, a koji kreira operativni sistem. Taj kontekst se sastoji iz različitih struktura podataka koje operativni sistem interno održava za svaki pokrenuti program, a koje sadrže informacije o trenutnom stanju u kome se program nalazi prilikom izvršenja, kao i informacije o resursima koji su programu dodeljeni (poput memorije, otvorenih fajlova, kanala za komunikaciju sa drugim procesima i sl.). Procesi se unutar operativnog sistema obično identifikuju brojevima procesa (engl. *process identifier* (PID)). Prilikom pokretanja programa, operativni sistem najpre inicijalizuje proces i dodeljuje mu

jedinstven PID. Nakon toga mu dodeljuje memoriju i u nju učitava programski kôd programa i inicijalizuje njegove podatke. Nakon toga mu (u nekom trenutku) dodeljuje procesor i program kreće da se izvršava.

Proces može biti kreiran od strane jezgra operativnog sistema ili od strane drugog procesa. Proces može kreirati druge procese i u okviru njih pokretati druge programe. Proces koji kreira drugi proces ćemo u tom slučaju nazivati *roditeljski proces* (engl. *parent process*), dok će pokrenuti proces biti *dete proces* (engl. *child process*). Posredno, procese može kreirati i korisnik tako što pokreće programe putem korisničkog interfejsa operativnog sistema. Sâm korisnički interfejs je takođe proces koji je upravo i namenjen za komunikaciju sa korisnikom (i za pokretanje procesa u ime korisnika).

Tokom svog rada, proces može zahtevati resurse od operativnog sistema. U cilju dodele tih resursa, operativni sistem može inicijalizovati dodatne strukture podataka. Na primer, ako proces želi da otvori neki fajl na disku koji želi da čita, operativni sistem će kreirati strukturu podataka koja će sadržati informacije o lokaciji otvorenog fajla na disku, broju bajta do koga se stiglo sa čitanjem, bafere koji sadrže pročitane bajtove i sl. Procesu će, po otvaranju fajla, biti prosleđena *ručka*, koja predstavlja neku vrstu identifikatora otvorenog fajla pomoću koje proces može u nastavku pristupati otvorenom fajlu i obavljati operacije sa njim. Po zatvaranju fajla, kreirane strukture podataka se uništavaju, a memorijski resursi se oslobađaju.

Proces može završiti svoj rad na više načina. Najprirodniji način je da se završi izvršavanje programskog kôda pokrenutog programa, čime program vraća kontrolu operativnom sistemu, a ovaj dealocira resurse dodeljene procesu i uklanja pridružene strukture podataka iz memorije. Drugi način je da program bude prekinut protiv svoje volje. To može uraditi ili operativni sistem (ako program pokuša da uradi nešto što nije dozvoljeno, poput izvršenja neke nedozvoljene instrukcije ili pristupa nedozvoljenoj zoni memorije), ili korisnik podsredstvom korisničkog interfejsa operativnog sistema i odgovarajućih alata za upravljanje procesima. Time se šalje signal operativnom sistemu da dati proces treba likvidirati, što operativni sistem radi oduzimajući mu resurse i uklanjajući ga iz evidencije aktivnih procesa.

5.1.3 Pokretanje i zaustavljanje operativnog sistema

Prilikom uključivanja računara, procesor odmah započinje sa radom. Njegov programski brojač (registar procesora koji sadrži adresu instrukcije koju sledeću treba izvršiti) je obično fabrički podešen na neku fiksiranu početnu vrednost, pa je potrebno obezbediti da počev od te adrese imamo program koji prvi treba da se izvrši po uključivanju računara. Ovo je odgovornost onoga ko ugrađuje procesor u svoj računar, a u slučaju modernih desktop i laptop računara, to su proizvođači matičnih ploča, koji u njih ugrađuju ROM memorije koje sadrže taj početni program koji se prvi izvršava. Tradicionalno, taj program se naziva BIOS (engl. *basic-input-output-system*).[1] Uloga BIOS programa je da inicijalizuje hardverske komponente, izvrši neke osnovne testove ispravnosti, a da zatim preda kontrolu sledećoj softverskoj komponenti u nizu, koja se naziva *punilac* (engl. *boot loader*). Ovu komponentu po pravilu obezbeđuje sâm operativni sistem koji je instaliran na računaru. Punilac se, kao i sâm operativni sistem, nalazi na nekoj od spoljnih memorija, tipično na hard ili SSD disku. Uobičajena konvencija je da se punilac (ili njegov početni deo, u slučaju složenijih punilaca), nalazi u okviru nultog sektora hard diska, koji je poznat pod nazivom MBR (engl. *master boot record*). BIOS učitava program iz MBR-a u memoriju i predaje mu kontrolu.

Uloga punioca je da pronađe ostale komponente operativnog sistema na disku i učita ih u operativnu memoriju. Kako ovaj postupak zavisi od vrste operativnog sistema, punilac mora biti kompatibilan sa operativnim sistemom i mora biti upoznat sa strukturom operativnog sistema. Otuda se punilac obično i isporučuje i instalira zajedno sa operativnim sistemom.[2] Nakon što punilac učita jezgro operativnog sistema u operativnu memoriju i preda mu kontrolu, njegova uloga se završava.

Jezgro operativnog sistema najpre inicijalizuje svoje strukture podataka, postavi procesor u odgovarajući režim rada, inicijalizuje različite sistemske atribute (poput sistemskog vremena, vremenskog intervala tajmera i sl.) i uspostavi okruženje za izvršavanje korisničkih procesa. Nakon toga, jezgro pokreće prvi korisnički program, čime se započinje proces inicijalizacije korisničkog okruženja i pokretanje sistemskih servisa. Na kraju tog procesa, korisniku se omogućava prijava na sistem, obično unosom korisničkog imena i lozinke. Ukoliko prijava bude uspešna, pokreće se korisnički interfejs operativnog sistema, što dalje omogućava korisniku da po želji pokreće svoje korisničke programe, upravlja svojim podacima ili izvršava različite administrativne poslove.

Na kraju rada sa računarom, korisnik pokreće proceduru isključivanja računara. Tom prilikom se zaustavljaju svi procesi koji trenutno rade na računaru, a zatim se kontrola predaje jezgru koje posebnim postupkom isključuje računar.

[1] U novije vreme, BIOS je ustupio svoje mesto programu koji se naziva UEFI (engl. *unified extensible firmware interface*), a koji služi istoj svrsi.

[2] Korisnici *Windows* operativnog sistema nisu ni svesni da punilac postoji, jer ih instalacioni program o tome ne informiše. Sa druge strane, *GNU/Linux* korisnici jako dobro znaju šta je punilac, čak uglavnom imaju i mogućnost izbora različitih punilaca prilikom instalacije (najpoznatiji su *Lilo* i *Grub*).

Elektronsko izdanje (2024)

5.1.4 Funkcije operativnog sistema

U ovom odeljku ukratko opisujemo osnovne funkcije operativnog sistema. Kao što ćemo videti, operativni sistem obavlja vrlo složene zadatke koji zahtevaju sa jedne strane interakciju sa hardverom na niskom nivou, a sa druge strane komunikaciju sa korisničkim programima i korisnikom na visokom nivou. Otuda je razvoj kvalitetnog operativnog sistema veoma zahtevan posao (po mišljenju mnogih, zahtevniji od razvoja većine aplikativnih programa).

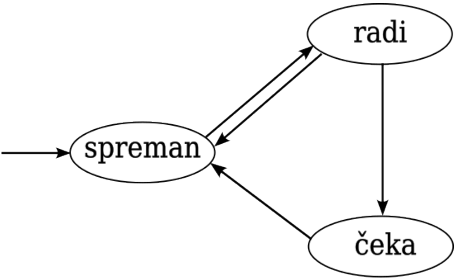
5.1.4.1 Upravljanje procesorom

Centralni procesor (engl. *Central processing unit* (CPU)) je, uz operativnu memoriju, najznačajnija hardverska komponenta svakog računara. Njegova uloga je da izvršava mašinske programe koji se nalaze u operativnoj memoriji računara. Za izvršavanje programa procesoru je potrebno neko vreme. Ovo vreme, koje nazivamo i *procesorsko vreme*, najznačajniji je resurs računarskog sistema koji programi koriste. Ukoliko bi se u memoriji nalazio samo jedan program, tada bi upravljanje ovim resursom bilo trivijalno – prosto bismo svo procesorsko vreme dodelili tom jednom programu. Sa druge strane, treba imati u vidu da je procesor najbrža hardverska komponenta računara. Sve druge komponente, od operativne memorije, preko spoljašnjih memorija, do ulazno-izlaznih uređaja, značajno su sporije. Otuda se prilikom izvršavanja programa najveći deo vremena neće trošiti za izvršavanje mašinskih instrukcija u procesoru, već ce se tipično trošiti na interakciju sa drugim sporijim komponentama hardvera, poput ulazno-izlaznih uređaja. Dok program čeka na izvršenje spore ulazno-izlazne operacije, procesor stoji u mestu i ne radi ništa. Pritom, vreme izvršavanja ulazno-izlazne operacije iz ugla procesora traje veoma dugo – izraženo u broju instrukcija koje bi procesor mogao da izvrši za to vreme, to može biti i više hiljada instrukcija. To znači da će efikasnost korišćenja procesorskog vremena kao resursa biti veoma niska, naročito u slučaju programa koji intenzivno koriste ulaz i izlaz.

Kako bi se efikasnost korišćenja procesora povećala, u memoriju treba učitati više programa. U svakom trenutku, jedan program se izvršava na procesoru, dok ostali programi čekaju da dobiju procesor. Kada program koji se izvršava mora da se zaustavi da bi sačekao neku sporu ulazno-izlaznu operaciju, operativni sistem predaje procesor na korišćenje drugog programu koji nastavlja svoj rad. Na ovaj način se obezbeđuje da procesor stalno bude uposlen, što značajno povećava njegovu efikasnost.

Opisano povećanje efikasnosti iskorišćenja procesora je istorijski bilo inicijalni motiv za uvođenje *multiprocesiranja*, tj. držanja više programa u operativnoj memoriji koji se naizmenično izvršavaju na procesoru. U današnje vreme, dodatni motiv jeste i povećanje udobnosti korišćenja računara kroz mogućnost obavljanja većeg broja poslova istovremeno (npr. surfujemo internetom, a istovremeno nam u pozadini svira muzika ili nam je otvoren prozor u kome uređujemo neki dokument). Danas je potpuno uobičajeno da se u svakom trenutku na računaru izvršava više desetina programa istovremeno.

Sa druge strane, multiprocesiranje čini upravljanje procesorskim vremenom kao resursom računara mnogo komplikovanijim. Sada procesorsko vreme nije ekskluzivni resurs jednog programa, već se mora deliti između većeg broja programa, zadovoljavajući pritom više različitih zahteva, od efikasnosti iskorišćenja procesora, preko pravednosti raspodele procesorskog vremena između programa do udobnosti za korisnika koji zahtevaju visok stepen interaktivnosti (tj. žele brz odziv svojih aplikacija). Da bi se sve to postiglo, tokom prethodnih decenija razvijeni su različiti *algoritmi raspoređivanja procesa* koji pokušavaju da zadovolje sve ove zahteve. Ovi algoritmi su implementirani u okviru operativnog sistema.



Slika 5.1: Graf stanja procesa

Da bismo bolje objasnili raspoređivanje procesa, razmotrimo najpre moguća *stanja* u kojima se može nalaziti neki proces tokom svog izvršenja. Graf stanja procesa i mogućih prelaza između njih dat je na slici 5.1. Prilikom kreiranja, proces dolazi u stanje *spreman* (engl. *ready*) koje ukazuje da procesor može započeti svoje izvršavanje na procesoru u proizvoljnom trenutku. U svakom trenutku možemo imati veći broj procesa koji su u ovom stanju. Sve procese koji su u stanju *spreman* operativni sistem drži u *redu spremnih procesa* (engl. *ready queue*). Iz ovog

reda operativni sistem uzima jedan proces (u skladu sa nekom politikom koja je implementirana u algoritmu raspoređivanja) i dodeljuje mu procesor, tj. preusmerava procesor na izvršenje odgovarajućeg programa. Ovaj proces prelazi iz stanja *spreman* u stanje *radi* (engl. *running*). Kada proces koji radi dođe do tačke kada mu je potrebna neka ulazno-izlazna operacija, on zahteva od operativnog sistema da tu operaciju za njega obavi (videćemo kasnije da procesi ne mogu sami baratati ulazno-izlaznim uređajima, već samo mogu da zamole operativni sistem da to za njih uradi). Operativni sistem tada proces prebacuje u stanje *čeka* (engl. *wait*), u kom će ostati sve dok ulazno-izlazni uređaj ne završi zahtevanu operaciju i o tome ne obavesti operativni sistem. Tada će operativni sistem informacije o obavljenoj operaciji dostaviti procesu i prebaciti ga iz stanja *čeka* u stanje *spreman*. Proces se dodaje u red spremnih procesa i čeka da dobije procesorsko vreme kako bi nastavio sa radom. Primetimo da samo procesi koji se nalaze u stanju *spreman* učestvuju u nadmetanju za dobijanje procesorskog vremena. Procesi u stanju *čeka* nisu kandidati za raspoređivanje u tom trenutku, s obzirom da oni ne mogu nastaviti svoj rad dok se ne završi zahtevana ulazno-izlazna operacija.

U najjednostavnijem scenariju raspoređivanja, proces prolazi kroz ciklus *spreman-radi-čeka-spreman*. Drugim rečima, proces koji radi može izgubiti pristup procesoru samo ako dođe do tačke u svom izvršenju u kojoj mu je potrebna ulazno-izlazna operacija. Tada on prelazi u stanje *čeka*. Sa stanovišta efikasnosti upotrebe procesora, ovo je najbolja moguća strategija, jer nema razloga prekidati rad procesa koji efektivno koristi procesor u punoj meri. Sa druge strane, ovakva strategija može da ne bude fer prema drugim procesima koji čekaju na procesor, ukoliko program previše dugo radi bez zahteva za ulazno-izlaznim operacijama (npr. programi koji vrše neka intenzivna izračunavanja i retko koriste ulaz i izlaz). Takođe, u moderno vreme, interaktivnost je veoma bitna karakteristika operativnih sistema. Korisnik želi da aktivno koristi više različitih programa i očekuje da ti programi brzo reaguju na njegove komande. Zbog toga je potrebno korisniku obezbediti utisak da se svi programi izvršavaju sve vreme, iako mi znamo da to nije moguće, s obzirom da imamo samo jedan procesor. Ovo se postiže tzv. *podelom vremena* (engl. *time sharing*). Ideja je da se svakom procesu odredi maksimalno vreme koje može da koristi u jednom krugu, pre nego što procesor prepusti drugom procesu. Ovo vreme nazivaćemo *vremenski kvantum* i obično je relativno kratko (npr. 50ms). Ukoliko proces koji je u stanju *radi* prekorači vremenski kvantum, a da pritom nije zahtevao ulazno-izlaznu operaciju, operativni sistem ga automatski prekida i prebacuje u stanje *spreman*, a iz reda spremnih procesa bira drugi proces koga prebacuje u stanje *radi* i prepušta mu procesor na korišćenje. Na ovaj način obezbeđujemo da se procesi dovoljno često smenjuju na procesoru, što ostavlja utisak korisniku da oni svi istovremeno rade. Efektivno, ako n procesa raspoređujemo na ovaj način na jednom procesoru date brzine, korisnik će imati utisak da računar poseduje n procesora koji su n puta manje brzine, pri čemu svaki od tih procesora sve vreme izvršava jedan od datih n procesa.

Osnovna operacija koju operativni sistem obavlja prilikom upravljanja procesorom jeste oduzimanje procesora jednom procesu i njegovo prepuštanje drugom procesu. Ovaj postupak se naziva *zamena konteksta* (engl. *context swich*). Pri svakoj zameni konteksta, potrebno je sačuvati stanje svih registara procesora (uključujući i programski brojač i pokazivač steka), kako bi proces kasnije mogao nastaviti tamo gde je stao. Vrednosti registara procesora se upisuju u posebnu zonu u memoriji koja se naziva *kontrolni blok procesa*. Svaki proces ima svoj kontrolni blok koji je deo struktura podataka operativnog sistema, što znači da se za svaki od procesa zna u kom su stanju ostavili registre u trenutku kada je njihov rad prekinut. Nakon što sačuva stanje registara prekinutog procesa, operativni sistem učitava u registre vrednosti sačuvane u kontrolnom bloku procesa koji je odabran da nastavi svoj rad. Učitavanjem njegovog programskog brojača kontrola se predaje tom procesu koji nastavlja sa radom.

5.1.4.2 Upravljanje operativnom memorijom

Operativna memorija (poznata i kao RAM memorija) je drugi najznačajniji resurs računara. Koristi se za čuvanje programskog kôda, podataka i ostalih informacija pridruženih aktivnim procesima, kao i programskog koda i struktura podataka samog operativnog sistema. Svaki proces konzumira određeni memorijski prostor u RAM-u. Otuda ovaj prostor smatramo resursom koji je dodeljen procesu. Kao i u slučaju procesora, i ovde bi upravljanje memorijskim prostorom kao resursom bilo trivijalno kada bismo imali samo jedan program u memoriji – prosto bismo svu memoriju dali na raspolaganje tom programu. Ukoliko pak želimo da u svakom trenutku imamo veći broj aktivnih procesa, moraćemo svakom od njih da dodelimo deo memorijskog prostora, što samo upravljanje ovim resursom čini komplikovanijim.

Prvi problem koji nastaje u slučaju koegzistencije više programa u memoriji je mogućnost da jedan proces slučajno ili namerno pristupi podacima drugog procesa ili samog operativnog sistema, što stvara ozbiljne funkcionalne i bezbednosne rizike. Da bismo to sprečili potrebno je da na neki način izolujemo memorijske prostore pojedinačnih procesa i sprečimo adresiranje memorijskih lokacija izvan tog prostora. Ovo se postiže upotrebom *virtuelne memorije*. Ideja je da svaki proces ima svoj zasebni *virtuelni adresni prostor* određene veličine. U pitanju je fiktivni adresni prostor koji ne postoji zaista, već se njegova egzistencija emulira od strane procesora, u saradnji sa operativnim sistemom. Virtuelni adresni prostor je jedini prostor koji je vidljiv procesu

tokom izvršavanja. On sadrži kompletan mašinski program, kao i sve podatke koje proces koristi tokom svog rada. Sve instrukcije programa referišu isključivo na adrese iz tog virtuelnog adresnog prostora. Takođe, adresa tekuće instrukcije programa koja se nalazi u programskom brojaču procesora je takođe adresa iz tog virtuelnog prostora. Adrese lokacija unutar virtuelnog adresnog prostora nazivaju se *virtuelne adrese*. Na primer, ako je veličina virtuelnog prostora 1MB (tj. 2^{20} bajtova), tada je opseg virtuelnih adresa kojima proces može pristupati od 0 do $2^{20} - 1$. Instrukcije programa mogu koristiti samo adrese iz ovog opsega i sve takve adrese se smatraju adresama unutar tog virtuelnog prostora. Na ovaj način se sprečava pristup podacima drugih procesa, s obzirom da proces može pristupati samo lokacijama u svom virtuelnom prostoru.

Kao što je već rečeno, virtuelni adresni prostor ne postoji zaista. Ono što zaista postoji jeste *fizički adresni prostor*, koji čini fizička RAM memorija koja je instalirana na našem računaru. Za razliku od virtuelnog prostora koji je zaseban za svaki proces, fizički prostor je jedinstven i zajednički za sve procese. Virtuelnim adresama svakog procesa pridružujemo fizičke adrese u RAM-u gde će odgovarajući podaci ili instrukcije zaista biti smeštene. Na koji način se vrši ovo pridruživanje zavisi od konkretne implementacije virtuelne memorije. Ono što je jedino važno je da se taj proces obavlja automatski i potpuno nevidljivo za sam proces koji fizičkih adresa uopšte nije ni svestan. Kada proces želi da pristupi nekoj adresi u svom virtuelnom prostoru, ta adresa se propušta kroz deo procesora koji se zove *memorijska jedinica* (engl. *memory unit*) koja uz pomoć informacija koje je unapred obezbedio operativni sistem automatski prevodi virtuelnu adresu u fizičku adresu na kojoj se traženi podatak zaista nalazi, a zatim procesor pristupa toj fizičkoj adresi u RAM memoriji.

Još jedan problem koji može nastati prilikom upravljanja memorijom jeste problem nedostatka prostora u fizičkoj memoriji, do koga može doći u slučaju pokretanja velikog broja programa i njihovog učitavanja u operativnu memoriju. Ovaj problem je naročito bio prisutan u nekom ranijem periodu, kada su RAM memorije bile relativno male. Problem nedovoljne količine memorije se tada rešavao tako što se programi nisu učitavali celi u memoriju, već samo oni delovi koji se trenutno koriste. Programer je morao sam da vodi računa o tome koji su delovi programa trenutno učitani i da po potrebi zahteva od operativnog sistema da iz memorije izbaci neke delove i da ubaci neke druge. Ova tehnika bila je poznata i kao *overlay* tehnika, i bila je prilično zahtevna i komplikovana za programera. U novije vreme, i ovaj problem se elegantno rešava pomoću tehnike virtuelne memorije. Naime, ne moraju sve virtuelne adrese nekog procesa u svakom trenutku imati pridružene fizičke adrese. Obično su samo neki delovi virtuelnog adresnog prostora „učitani" u fizičku memoriju, dok se ostali delovi virtuelnog prostora mogu čuvati u spoljnoj memoriji (tipično na hard ili SSD disku). Ovakvo učitavanje virtuelnog adresnog prostora nazivaćemo *parcijalno učitavanje*. Kada proces pristupi nekoj virtuelnoj adresi za koju se ispostavi da nema pridruženu fizičku adresu, dolazi do automatskog prekida rada programa i predaje kontrole operativnom sistemu. Operativni sistem razrešava ovu situaciju tako što sa hard diska učita deo virtuelnog prostora procesa koji sadrži traženu adresu u neki slobodni deo operativne memorije i u skladu sa tim ažurira svoje strukture podataka koje čuvaju informacije o odgovarajućem pridruživanju virtuelnih i fizičkih adresa. Nakon toga operativni sistem vraća kontrolu prekinutom procesu koji, ne znajući šta se uopšte dogodilo, ponovo pokušava da izvrši istu instrukciju. Ovog puta će proces prevođenja virtuelne adrese u fizičku biti uspešan, s obzirom da je odgovarajući deo virtuelnog prostora učitan u fizičku memoriju. Proces će uspešno izvršiti instrukciju i nastaviti sa daljim radom. Važno je razumeti da se ceo ovaj postupak obavlja potpuno nevidljivo za sam proces, pri čemu se odgovarajući delovi virtuelnog adresnog prostora učitavaju u memoriju automatski, onda kada su potrebni. Otuda, programer koji piše program može smatrati da je ceo njegov virtuelni prostor sve vreme dostupan i ne mora preduzimati nikakve akcije da bi to obezbedio. Ukupna veličina svih virtuelnih adresnih prostora svih aktivnih procesa je obično značajno veća od veličine fizičke memorije računara, ali to ne predstavlja problem, zato što se ne čuvaju celi virtuelni prostori u RAM-u, već samo delovi koji se trenutno koriste. Ovo korisniku računara ostavlja utisak da ima znatno više memorije nego što je zaista ima. Ipak, treba imati u vidu da preveliki broj aktivnih procesa može značajno pogoršati performanse računara, ukoliko to dovede do prepunjenosti fizičke memorije. Ukoliko se ispostavi da za učitavanje zahtevanog dela virtuelnog prostora nekog procesa nema dovoljno slobodnog mesta u fizičkoj memoriji, operativni sistem će morati da izbaci deo virtuelnog prostora nekog drugog procesa iz fizičke memorije i prebaci ga na hard disk, kako bi oslobodio prostor. Ovo se takođe obavlja automatski, od strane operativog sistema i za posledicu ima značajno usporavanje rada računara u slučaju prepunjenosti operativne memorije. U tom slučaju će operativni sistem morati neprestano da prebacuje podatke sa hard diska i na hard disk, koji je značajno sporiji od operativne memorije.

Dodeljivanje fizičke memorije procesu može biti *kontinualno* i *diskontinualno*. U prvom slučaju, svakom procesu se dodeljuje odgovarajući broj *susednih* adresa fizičke memorije, tj. dodeljuje mu se fizički adresni prostor „u komadu". Ovakav pristup je veoma jednostavan, jer se učitavanje i uklanjanje procesa iz memorije jednostavno implementira. Takođe, implementacija virtuelne memorije u slučaju kontinualnog dodeljivanja fizičke memorije je veoma jednostavna. Pretpostavimo da je veličina virtuelnog prostora procesa 64KB, kao i da je procesu dodeljen kontinualni prostor u fizičkoj memoriji veličine 64KB, počev od fizičke adrese 1024. Za implementaciju virtuelne memorije, potrebno je da procesor sadrži dva posebna registra. Prvi registar, označimo ga sa B, predstavlja

Elektronsko izdanje (2024)

bazni registar koji sadrži fizičku adresu od koje počinje kontinualni prostor u fizičkoj memoriji koji je dodeljen procesu (u našem primeru, u taj registar treba učitati 1024). Drugi registar, označimo ga sa L, predstavlja *granični registar* koji sadrži prvu fizičku adresu nakon završetka kontinualnog prostora koji je dodeljen procesu (tj. prvu adresu koja *ne* pripada našem procesu). U našem primeru, u registar L biće učitan zbir $1024+64 \cdot 1024$. Translacija virtuelne adrese va u fizičku adresu fa vrši se veoma jednostavno, po formuli $fa = va + B$. Prilikom izračunavanja fizičke adrese, memorijska jedinica mora da proveri da li je dobijena fizička adresa u opsegu koji je dodeljen našem procesu. U tu svrhu je potrebno samo ispitati da li je $fa < L$. Dakle, cela memorijska jedinica će se sastojati iz jednog binarnog sabirača i jednog komparatora. Vrednosti bazne i granične adrese za svaki od aktivnih procesa se čuvaju u strukturama podataka operativnog sistema. Prilikom promene konteksta, registri B i L će biti od strane operativnog sistema automatski postavljeni na baznu i graničnu adresu procesa koji je raspoređen na izvršavanje u procesoru. Na taj način će svaki proces pristupati isključivo adresama iz opsega koji mu je dodeljen.

Dva su glavna nedostatka kontinualnog dodeljivanja fizičke memorije procesu. Prvi nedostatak je to što ovakav način dodeljivanja memorije podrazumeva da se ceo virtuelni prostor nalazi u fizičkoj memoriji u kontinualnom opsegu memorijskih adresa. To znači da će parcijalno učitavanje virtuelnog prostora procesa u cilju uštede fizičke memorije biti otežano, jer će različiti delovi virtuelnog adresnog prostora koji se zasebno po potrebi učitavaju morati da budu učitavani u susedne zone u memoriji, što često neće biti moguće jednostavno realizovati. Drugi nedostatak je pojava *fragmentacije* fizičke memorije. Naime, kada se neki proces završi, memorijski prostor koji je on zauzimao se oslobađa. Na tom mestu u memoriji sada imamo slobodan prostor koji je veliki onoliko koliko je prostora zauzimao proces koji se upravo završio. Operativni sistem će taj slobodan prostor moći da dodeli nekom novom procesu samo ako taj novi proces ne zahteva više memorije. U suprotnom, operativni sistem će morati da pronađe neki drugi slobodan prostor u memoriji, a taj prostor će ostati neupotrebljen. Kako se tokom rada računara procesi stalno pokreću i završavaju sa radom, nakon nekog vremena možemo imati veliki broj „rupa" u memoriji koje predstavljaju slobodni prostor. Sada se može lako dogoditi da za neki novi proces mi imamo dovoljno slobodnog prostora posmatrano ukupno, ali ga nemamo „u komadu", s obzirom da je slobodni prostor fragmentisan na veliki broj delova.

Oba ova nedostatka se rešavaju upotrebom nekontinualnog dodeljivanja fizičkog memorijskog prostora procesu. U ovom scenariju, nije neophodno da se ceo virtuelni prostor procesa preslika u kontinualni deo fizičkog prostora, tj. nije neohodno da uzastopne virtuelne adrese budu preslikane u uzastopne fizičke adrese. Na ovaj način mi možemo virtuelni adresni prostor izdeliti na delove i svaki od delova učitati u posebne zone fizičke memorije. Sada naknadno učitavanje dela virtuelnog adresnog prostora u memoriju ne predstavlja problem, jer je dovoljno pronaći bilo koji deo memorije koji je slobodan. Takođe, fragmentacija više neće postojati, jer će svi slobodni delovi memorije moći da budu popunjeni delovima virtuelnog adresnog prostora odgovarajuće veličine. Sa druge strane, nekontinualno dodeljivanje fizičke memorije je znatno komplikovanije za implementaciju, s obzirom da operativni sistem mora za svaki proces čuvati informacije o svim delovima virtuelnog adresnog prostora (da li su učitani u RAM i počev od koje adrese). Memorijska jedinica procesora mora da učitava ove informacije i na osnovu njih preračunava fizičke adrese, što samu memorijsku jedinicu čini kompleksnijom.

Jedan od najčešćih načina upravljanja memorijom kod savremenih računara je zasnovan na *straničenju* (engl. *paging*). U pitanju je sistem virtuelne memorije zasnovan na nekontinualnom dodeljivanju fizičkih adresa. U ovom sistemu, svi procesi imaju virtuelni adresni prostor iste veličine. U cilju ilustracije, pretpostavimo da imamo 32-bitni virtuelni adresni prostor. To znači da su sve virtuelne adrese 32-bitne, pa je veličina virtuelnog adresnog prostora svakog procesa 4GB. Virtuelni prostor procesa se logički deli na *stranice* (engl. *page*) jednake veličine. Određenosti radi, neka su sve stranice veličine 4KB (što je 2^{12} bajtova). To znači da se ceo virtuelni prostor procesa sastoji iz 2^{20} stranica od po 2^{12} bajtova (ukupno 2^{32} bajtova). Možemo smatrati da se sada 32-bitna virtuelna adresa (va) sastoji iz dva dela: viših 20 bitova predstavljaju redni broj stranice (označimo ga sa ha), dok nižih 12 bitova (označimo ih sa la) predstavljaju *pomeraj* (engl. *offset*) u okviru stranice.

Sa druge strane, fizički adresni prostor se takođe deli na delove koji su jednake veličine kao i stranice (u našem primeru 4KB). Nazivamo ih *okviri stranica* (engl. *page frames*). Primera radi, pretpostavimo da imamo 16GB (2^{34}) fizičkog adresnog prostora. Ovaj prostor se deli na 2^{22} okvira od po 2^{12} bajtova. U svaki okvir može biti učitana jedna stranica virtuelnog adresnog prostora nekog od aktivnih procesa. Pritom, svaka stranica može biti učitana u bilo koji slobodan okvir, tj. ne postoji zahtev da susedne stranice u virtuelnom adresnom prostoru nekog procesa budu u susednim okvirima fizičkog adresnog prostora. Ovo znači da prilikom učitavanja programa u memoriju njegove stranice mogu biti razbacane po celom fizičkom adresnom prostoru na proizvoljan način. Kako bismo mogli da izvršimo prevođenje virtuelne adrese u fizičku, neophodno je da za svaku stranicu znamo u kom se okviru nalazi. U ovu svrhu, operativni sistem za svaki proces održava *tabelu stranica* (engl. *page table*). U pitanju je tabela koja sadrži po jednu stavku za svaku od 2^{20} stranica datog procesa. Stavke su jednake veličine (npr. 8 bajta) i nalaze se jedna za drugom u memoriji, počev od neke fizičke adrese koja je poznata operativnom sistemu. U procesoru postoji poseban registar (označimo ga sa PTA - *page table address*) koji sadrži *fizičku* adresu tabele stranica za proces koji se trenutno izvršava u procesoru (prilikom zamene konteksta,

u PTA registar se učitava odgovarajuća adresa). Svaka stavka sadrži, između ostalog, informaciju o tome da li je odgovarajuća stranica prisutna u fizičkoj memoriji i, ako jeste, u kom okviru. Proces prevođenja virtuelne u fizičku adresu se sada obavlja na sledeći način: najpre se na osnovu višeg dela virtuelne adrese ha odredi redni broj stranice virtuelnog adresnog prostora. Zatim se pomoću PTA registra pristupa stavci tabele stranica sa tim rednim brojem (tj. pristupa se 8-bajtnom prostoru na adresi $PTA + 8 \cdot ha$). Ova stavka se učitava u memorijsku jedinicu i iz nje se utvrđuje da li je stranica učitana u fizičku memoriju ili ne. Ako jeste, tada se iz stavke pročita 22-bitni redni broj okvira u fizičkoj memoriji gde se stranica nalazi. Na ovo 22-bitno polje se dopisuje niži deo virtuelne adrese la, čime se dobija 34-bitna fizička adresa podatka u memoriji (ovo je zato što je pomeraj u okviru jednak pomeraju u stranici). Ukoliko se ispostavi da stranica nije učitana u fizičku memoriju, dolazi do prekida rada programa (tzv. *greška straničenja*, engl. *page fault*). Kontrola se predaje operativnom sistemu koji sa hard diska učitava traženu stranicu u neki slobodan okvir u fizičkoj memoriji. Ažurira se odgovarajuća stavka u tabeli stranica, a zatim se kontrola vraća procesu koji ponovo pokušava da izvrši istu instrukciju. Ovom prilikom prevođenje uspeva, jer se stranica sada nalazi u memoriji. Ako prilikom učitavanja nove stranice nema slobodnih okvira, tada operativni sistem bira „žrtvu" – stranicu koju će izbaciti iz svog okvira i iskopirati je na hard disk, kako bi se okvir oslobodio za novu stranicu. Pritom, izbačena stranica ne mora pripadati istom procesu. Odabir prave „žrtve" nije ni malo jednostavan, jer je cilj smanjiti broj skupih grešaka straničenja u budućnosti, što zahteva anticipaciju budućeg korišćenja stranica koje se nalaze u okvirima u memoriji.

5.1.4.3 Upravljanje ulazno-izlaznim uređajima

Ulazno-izlazni uređaji omogućavaju komunikaciju računara sa korisnikom, kao i sa drugim uređajima i računarima. Tradicionalni ulazni uređaji za komunikaciju sa korisnikom su tastatura i miš, a izlazni je monitor. U novije vreme, sve se češće koriste i ekrani osetljivi na dodir, kao vrsta ulazno-izlaznog uređaja. Pored toga, u ulazno-izlazne uređaje spadaju različite vrste štampača i skenera, kao i multimedijalni uređaji poput zvučnika, mikrofona, kamera i sl. Za komunikaciju sa drugim računarima koriste se različite vrste mrežnih uređaja, poput mrežnih i bežičnih (engl. *wireless*) kartica. Najzad, posebnu vrstu ulazno-izlaznih uređaja čine i *spoljne memorije* koje se koriste za skladištenje podataka i softvera (poput hard i SSD diskova, CD/DVD diskova, fleš memorija i sl.).

Procesor računara komunicira sa ulazno-izlaznim uređajima podsredstvom *ulazno-izlaznih kontrolera*. Kontroler se povezuje na magistralu i procesor sa njim komunicira na veoma sličan način kao i sa operativnom memorijom, razmenjujući *podatke*, *komande* i *statusne informacije* putem magistrale. Kontroler se, sa druge strane, povezuje sa samim uređajem na način koji je prilagođen prirodi tog uređaja.

Skup komandi koje razume ulazno-izlazni kontroler veoma zavisi od vrste uređaja, proizvođača, pa čak i samog modela uređaja. Otuda bi direktna komunikacija sa ulazno-izlaznim kontrolerima od strane korisničkih programa bila veoma teška, jer bi programer morao da poznaje detalje komandnog jezika, kao i da obezbedi podršku za veliki broj različitih tipova i modela uređaja. Takođe, direktna komunikacija sa kontrolerima predstavlja i bezbednosni izazov, jer bi pogrešnom upotrebom uređaj mogao i da se ošteti ili zloupotrebi. Zbog toga se komunikacija sa ulazno-izlaznim uređajima prepušta operativnom sistemu, a njegov deo zadužen za taj posao naziva se *ulazno-izlazni podsistem*. Ovaj sistem se sastoji iz dva sloja. Na nižem sloju nalaze se upravljački programi zaduženi za direktnu komunikaciju sa kontrolerima. Ovi programi su poznati i kao *drajveri* (engl. *device drivers*) i u njima su implementirani svi detalji komandnog jezika konkretnog uređaja. Za svaki tip i model uređaja operativni sistem mora biti snabdeven odgovarajućim drajverom. Vrlo često, drajveri se isporučuju od strane proizvođača hardvera, zajedno sa samim uređajem. Na višem sloju implementiran je interfejs ka drugim delovima operativnog sistema i korisničkim programima, a koji je nezavisan od konkretnog tipa i modela uređaja. Ovim interfejsom se obezbeđuje *apstrakcija hardvera*, jer se drugim delovima operativnog sistema i korisničkim programima obezbeđuje unifikovani pristup različitim uređajima, što značajno olakšava korišćenje samih uređaja.

U okviru ulazno-izlaznog podsistema implementirani su različiti *baferi*. Bafer predstavlja strukturu podataka koja može skladištiti podatke koji se prenose između dve strane u komunikaciji – u ovom slučaju između ulazno-izlaznog kontrolera i operativnog sistema. Uloga bafera je da amortizuje razliku u brzini između dve strane koje komuniciraju. Na primer, kada neka aplikacija šalje podatke izlaznom uređaju (npr. štampaču), ona ih prosleđuje operativnom sistemu koje te podatke dalje prosleđuje drajveru za odgovarajući uređaj, a ovaj ih šalje samom kontroleru, putem magistrale, koristeći odgovarajući komandni jezik. Međutim, ako je brzina slanja podataka prevelika, kontroler neće moći da ih prihvati, te postoji opasnost od gubitaka podataka. Da se to ne bi desilo, operativni sistem podatke privremeno smešta u bafer. Podaci se zatim iz bafera prosleđuju drajveru onom brzinom kojom uređaj može da ih prihvati. Ukoliko se bafer prepuni, tada će operativni sistem prestati da prihvata podatke od aplikacije, dok se u baferu ne napravi dovoljno mesta.

Elektronsko izdanje (2024)

5.1.4.4 Upravljanje podacima

Podaci se na računaru skladište u spoljnim memorijama (poput hard i SSD diskova, fleš memorija i sl.). Računar ove uređaje vidi kao ulazno-izlazne uređaje i sa njima komunicira putem odgovarajućih kontrolera (npr. SATA kontroler se koristi za komunikaciju sa hard i SSD diskovima, dok se USB kontroler koristi za komunikaciju sa USB fleš memorijama). Svaka spoljna memorija se može logički posmatrati kao niz adresibilnih lokacija fiksirane veličine. Na primer, hard diskovi se mogu posmatrati kao nizovi *sektora*, tipične veličine 512 bajtova, čije adrese počinju od nule. Putem ulazno-izlaznog podsistema, moguće je vršiti očitavanje ili upis pojedinačnih sektora hard diska i na taj način baratati sa skladištenim podacima.

Iako bi, teorijski, korisnički programi mogli da sa spoljnim memorijama na ovaj način razmenjuju sirove podatke (tj. konkretne bajtove, brojeve, tekst i sl.), tako nešto bi vrlo brzo dovelo do haosa, jer bi vrlo teško bilo pronaći podatke, kao i sprečiti da aplikacije greškom oštete podatke upisane na uređaju.

Zbog toga se podaci na spoljnim memorijama organizuju na način koji obezbeđuje konzistentan, efikasan i bezbedan pristup. Ovo se postiže podsredstvom specijalnih struktura podataka koje čine *sistem datoteka* (engl. *file system*). Pod *datotekom* ili *fajlom* (engl. *file*) podrazumevamo skup logički povezanih podataka koji čine jednu celinu. Na primer, fajl može biti neki tekstualni dokument, slika, video zapis, ali i računarski program. Svaki fajl ima svoje ime koje ga identifikuje, kao i sadržaj koji je predstavljen nizom bajtova, u formatu koji zavisi od specifičnog tipa fajla. Na nekim fajl sistemima, tip fajla može biti određen *ekstenzijom*, tj. specifičnim nastavkom imena (npr. *.txt*, *.doc*, *.jpg*, *.exe*, i sl.), dok na drugim sistemima naziv fajla nema uticaja na tip, već se tip prepoznaje na drugačiji način. Pored imena, tipa i sadržaja, fajl može imati i niz drugih karakteristika poput veličine (u bajtovima), vlasništva, prava pristupa, datuma poslednje modifikacije i sl.

Sistem datoteka obično podrazumeva dva aspekta – *logički* i *fizički*. Logički aspekt predstavlja način na koji se podaci koji se nalaze skladišteni u okviru sistema datoteka predstavljaju korisniku. Danas su uobičajeni *hijerarhijski* sistemi datoteka, kod kojih se fajlovi organizuju u *direktorijume* ili *fascikle*. Svaki direktorijum može sadržati fajlove, kao i druge direktorijume (koje nazivamo *poddirektorijumi*). Na vrhu hijerarhije nalazi se *koreni direktorijum* (engl. *root directory*) koji sadrži (posredno) sve podatke koji se nalaze u okviru tog sistema datoteka. Direktorijumi kao i fajlovi imaju svoja imena. Ovakav pristup omogućava korisniku da svoje podatke organizuje tako što ih razvrstava na kategorije i podkategorije, čime je omogućeno lakše pronalaženje fajlova.[3]

Fizički aspekt podrazumeva na koji način se podaci fizički skladište u spoljnoj memoriji. Ovo je obično sakriveno od korisnika i odgovornost je operativnog sistema. Setimo se da se memorijski prostor spoljne memorije može posmatrati kao niz adresibilnih jedinica. Uobičajeno je da sistem datoteka na početku tog memorijskog prostora skladišti meta-informacije o sebi (tip sistema datoteka, verzija, i sl.) kao i strukture podataka koje sadrže informacije o fajlovima i direktorijumima i omogućavaju pronalaženje sadržaja konkretnih fajlova u okviru fajl sistema. U nastavku prostora nalaze se sami fajlovi, tj. njihovi sadržaji.

Postoji više tipova sistema datoteka koji se danas aktivno koriste. Neki od njih su *FAT*, *NTFS*, *ext4*, *XFS*, *ReiserFS* i sl. Svaki operativni sistem podržava određeni skup tipova sistema datoteka i može pristupati samo sistemima datoteka tih tipova. Sistemi *FAT* i *NTFS* su razvijeni za operativne sisteme *DOS* i *Windows* od strane kompanije Majkrosoft, ali su podržani i od strane *GNU/Linux* operativnog sistema. Sistemi *ext4* i *ReiserFS* su razvijeni za *Linux* i podrazumevano nisu podržani od strane *Windows* operativnog sistema (mada postoji mogućnost instalacije dodatnog sistemskog softvera koji omogućava pristup ovakvim sistemima datoteka od strane *Windows* korisnika).

Da bi spoljna memorija mogla da se koristi za skladištenje podataka, potrebno je da se na njoj inicijalizuje sistem datoteka. Ovo se radi pomoću sistemskih alata koji se isporučuju sa samim operativnim sistemom. Postupak kreiranja sistema datoteka poznat je i kao *formatiranje*. Nakon formatiranja, korisnik dobija pristup sistemu datoteka koji je inicijalno prazan, tj. u njemu nema fajlova niti direktorijuma, izuzev korenog direktorijuma. Nakon toga, korisnik može kreirati fajlove i direktorijume i na taj način organizovati svoje podatke.

Pojedini uređaji poput hard i SSD diskova se mogu od strane operativnog sistema logički posmatrati kao više nezavisnih spoljnih memorija. Disk se može logički podeliti na *particije* – međusobno disjunktne celine, pri čemu se svaka particija sastoji iz skupa susednih sektora. Informacije o particionisanju (tj. od kog do kog sektora se prostire koja particija) se obično nalaze u početnim sektorima diska koji nisu deo ni jedne particije. Sada se sistemi datoteka mogu nezavisno kreirati u svakoj od particija.

Lokacija svakog fajla ili direktorijuma u okviru hijerarhijskog sistema datoteka se opisuje *putanjom*, koja može biti *apsolutna* i *relativna*. Apsolutna putanja se dobija navođenjem svih direktorijuma u stablu prateći put od korenog direktorijuma do datog fajla ili direktorijuma. Na primer, na operativnom sistemu *UNIX*, putanja `/etc/apache/httpd.conf` predstavlja apsolutnu putanju koja određuje lokaciju fajla `httpd.conf` (u okviru direktorijuma `apache` koji je poddirektorijum direktorijuma `etc` koji se nalazi u okviru korenog direktorijuma

[3]Zanimljivo je da mnogi korisnici računara ne koriste ovu mogućnost, već sve fajlove smeštaju u isti direktorijum (tipično na „radnu površinu" korisničkog interfejsa operativnog sistema). Kada se sledeći put budete mučili da na radnoj površini pronađete nešto što Vam treba, setite se da ste mogli da fajlove razvrstate u direktorijume, što bi Vam verovatno olakšalo pretragu.

`/`). Sa druge strane, relativna putanja predstavlja putanju nekog fajla ili direktorijuma u odnosu na neki drugi direktorijum u stablu direktorijuma (tipično u odnosu na *tekući direktorijum*, tj. radni direktorijum koji je pridružen procesu koji pristupa sistemu datoteka). Na primer, ako je tekući direktorijum `/etc/`, tada putanja `apache/httpd.conf` predstavlja relativnu putanju fajla `httpd.conf` u odnosu na direktorijum `/etc/`.

Uobičajeno je da sistemi datoteka podržavaju neku vrstu zaštite podataka od neovlašćenog korišćenja, obično kroz sistem *prava pristupa*. U okviru sistema datoteka se, sa određenim stepenom granulacije, mogu definisati pojedinačna prava određenih korisnika i grupa korisnika nad datim fajlom ili direktorijumom (poput prava čitanja, modifikacije, kreiranja, brisanja, pokretanja programa i sl.). Operativni sistem je odgovoran da spreči svaki neovlašćen pristup fajlovima i direktorijumima u okviru sistema datoteka.

5.1.4.5 Komunikacija između procesa

Često postoji potreba da procesi međusobno komuniciraju, razmenjuju podatke i sinhronizuju se u cilju zajedničkog obavljanja nekog kompleksnog zadatka. U tu svrhu, operativni sistemi obično obezbeđuju različite mehanizme interprocesne komunikacije. Najjednostavniji mehanizmi podrazumevaju slanje jednostavnih poruka kojima se proces obaveštava o određenom događaju ili podstiče na neku akciju (poput *UNIX signala*). Složeniji mehanizmi komunikacije omogućavaju razmenu proizvoljnih podataka putem kreiranog kanala (primer takvih mehanizama su *cevi* (engl. *pipe*) i *priključci* (engl. *socket*) na *UNIX* sistemima). Ovi kanali mogu biti *jednosmerni* (engl. *half-duplex*) ili *dvosmerni* (engl. *full-duplex*). Ovi mehanizmi se tipično koriste za komunikaciju sa sistemskim servisima koje koriste različiti aplikativni programi (poput sistema za pristup štampačima), ali i za komunikaciju aplikacija sa korisničkim interfejsom. Najzad, kao naročito efikasan vid komunikacije između procesa može se koristiti *deljena memorija* (engl. *shared memory*). Ovaj način komunikacije podrazumeva da dva procesa dele neki zajednički deo virtuelnog adresnog prostora. Samim tim, ono što jedan proces upiše u tu deljenu memoriju automatski je vidljivo drugom procesu u okviru njegovog virtuelnog adresnog prostora. Na ovaj način procesi mogu brzo i efikasno razmenjivati velike količine podataka.

Na kraju, napomenimo da procesi koji komuniciraju ne moraju se obavezno izvršavati na istom računaru. Zahvaljujući umrežavanju, omogućeni su mehanizmi komunikacije između procesa koji se izvršavaju na različitim računarima koji su međusobno povezani. U tu svrhu se koriste *mrežni protokoli* koji definišu pravila komunikacije, a koji su nezavisni od konkretnog hardvera ili operativnog sistema.

5.1.4.6 Komunikacija sa korisnikom

Da bi korisnik mogao da koristi svoj računar, neophodno je da može da pokreće svoje aplikacije, upravlja njihovim radom, kao i da koristi resurse svog računara, poput podataka i ulazno-izlaznih uređaja. U tu svrhu, svaki operativni sistem sadrži komponentu koja se zove *korisnički interfejs* (engl. *user interface* (UI)). Korisnički interfejs predstavlja skup uslužnih programa koji su povezani sa standardnim ulazno-izlaznim uređajima poput tastature, miša i monitora, preko kojih mogu komunicirati sa korisnikom. Ovi programi se pokreću prilikom pokretanja operativnog sistema i prijavljivanja korisnika na sistem, čime se kreiraju odgovarajući procesi koji su aktivni dokle god je korisnik prijavljen. Korisnik može korisničkom interfejsu izdavati *komande* kojima od operativnog sistema zahteva neku akciju. Te akcije mogu podrazumevati pristup nekom fajlu u okviru fajl sistema, promenu tekućeg direktorijuma, kao i pokretanje programa. Kada se izda odgovarajuća komanda, tada proces korisničkog interfejsa odgovara na tu komandu tako što za korisnika izvršava odgovarajuću akciju. Na primer, ako je korisnik želeo da pokrene neki program, proces korisničkog interfejsa će kreirati novi proces i u okviru njega pokrenuti željeni program kome će predati kontrolu nad standardnim ulazom i izlazom, čime će nadalje biti omogućena komunikacija korisnika sa pokrenutim programom. Kada se program završi, kontrola će biti vraćena korisničkom interfejsu koji će spremno čekati novu komandu korisnika.

Tradicionalni korisnički interfejs je bio *tekstualni*, u smislu da su se komande zadavale u tekstualnoj formi, što je podrazumevalo da korisnik poznaje sintaksu komandnog interfejsa. Poruke koje je korisnički interfejs izdavao korisniku su takođe bile tekstualne. U novije vreme, tekstualni korisnički interfejsi su ustupili mesto *grafičkim korisničkim interfejsima* (engl. *graphical user interface* (GUI)). Grafički interfejs podrazumeva da korisnik izdaje akcije pomoću miša, pritiskom na različite grafičke elemente koji su prikazani na ekranu (poput dugmadi, menija, ikona i sl.). Korisnički intefejs, kao i aplikacije koje se u okviru njega pokreću, može korisniku takođe saopštavati poruke koristeći grafičke elemente (prozore, dijaloge, slike, animacije i sl.). Ovakav način korišćenja je znatno jednostavniji i udobniji, a naročito je prilagođen osobama skromnog tehničkog znanja. Ipak, naglasimo da tekstualni korisnički interfejsi i dalje nude mnoge mogućnosti koje grafički interfejsi nemaju. Otuda tekstualni korisnički interfejs i dalje ostaje prvi izbor naprednijih korisnika računara.

5.1.5 Hardverska podrška operativnim sistemima

Da bi operativni sistemi mogli da implementiraju svoje funkcionalnosti opisane u prethodnim odeljcima, neohodna je izvesna podrška od strane hardvera (pre svega od strane procesora). U ovom odeljku opisujemo neke osnovne hardverske funkcionalnosti neophodne za implementaciju operativnih sistema.

5.1.5.1 Nivoi privilegija procesora

Već je rečeno da korisnički programi ne bi trebalo da direktno pristupaju hardveru računara, već isključivo putem interfejsa operativnog sistema. Takođe, postoje registri procesora (poput registra PTA koji sadrži adresu tabele stranica) koji služe isključivo za implementaciju funkcija operativnog sistema, pa ne bi bilo dobro da korisnički programi mogu da im pristupaju. Kako bi se fizički sprečilo da korisnički programi direktno pristupaju resursima koji im nisu namenjeni, procesor obično podržava više različitih *nivoa privilegija* (engl. *privilege level*). U svakom trenutku, procesor se nalazi na jednom od mogućih nivoa privilegija. Svaki od nivoa podrazumeva određeni skup registara i instrukcija procesora koje se mogu koristiti na tom nivou. Ako program pokuša da pristupi registru ili izvrši instrukciju koja nije u tom skupu, dolazi do prekida rada programa i predaje kontrole odgovarajućoj funkciji operativnog sistema koja je za takvu situaciju predviđena (što obično dovodi do likvidacije procesa). Na primer, na arhitekturi *x86* (i *x86-64*) postoje četiri nivoa privilegija, označeni brojevima od 0 do 3. Nivo 0 je najviši nivo privilegija i u njemu su dostupne sve instrukcije i svi registri procesora. Svaki sledeći nivo podrazumeva manji skup instrukcija i registara koji su dostupni. Nivo 3 apsolutno sprečava svaki neovlašćen pristup hardveru računara kao i specijalnim registrima procesora predviđenim za implementaciju funkcija operativnog sistema. Po uključivanju računara, procesor se podrazumevano nalazi u najvišem nivou privilegija (nivo 0). Posebnom instrukcijom procesor se može prebaciti u neki od nižih nivoa po želji. Sa druge strane, kada se nalazimo na nižem nivou privilegije, ne postoji instrukcija koja bi nam omogućila da se vratimo na viši nivo. To znači da proces koji se izvršava sa nižim privilegijama (poput korisničkog procesa) neće moći da tek tako dobije više privilegije koje bi mu omogućile pristup osetljivim delovima sistema. Ono što procesor obično obezbeđuje jesu specijalni mehanizmi privremenog i kontrolisanog prelaska na viši nivo privilegija radi obavljanja sistemskih poslova (poput izvršenja sistemskih poziva operativnog sistema ili obrade prekida). Ovi mehanizmi su rezervisani isključivo za pozivanje kôda koje obezbeđuje jezgro operativnog sistema. Drugim rečima, ne postoji način da korisnički program izvrši proizvoljan kôd u privilegovanom režimu rada.

Za implementaciju operativnog sistema obično je dovoljno da procesor podržava dva nivoa privilegija: jedan koji obezbeđuje pristup svim resursima procesora (poput nivoa 0 na x86 arhitekturi) i jedan koji uskraćuje pristup svim sistemski osetljivim resursima procesora (poput nivoa 3 na x86 arhitekturi). Na ovom prvom se obično izvršava isključivo kôd jezgra operativnog sistema. Nazivamo ga i *nivo jezgra* (engl. *kernel level*). Na ovom drugom se izvršavaju svi korisnički programi (ne samo aplikativni programi, već i sistemski programi i delovi operativnog sistema koji ne zahtevaju privilegovani režim, poput korisničkog interfejsa). Nazivamo ga i *korisnički nivo* (engl. *user level*).

5.1.5.2 Podrška za upravljanje memorijom

Većina savremenih procesora podržava mehanizam straničenja za implementaciju virtuelne memorije. U tu svrhu, postoji memorijska jedinica koja vrši automatsko prevođenje virtuelnih u fizičke adrese, na osnovu tabele stranica čija se fizička adresa nalazi u posebnom registru (koji smo mi označavali kao PTA). Osnovni parametri koji definišu mehanizam straničenja jesu veličina virtuelnog i fizičkog adresnog prostora, kao i veličina stranica i okvira. Mnogi procesori podržavaju više različitih varijanti straničenja, kada su u pitanju vrednosti ovih parametara. Tako, na primer, x86 arhitektura podržava dve različite veličine stranica: 4KB i 4MB. Izbor ovih vrednosti parametara može se vršiti uključivanjem odgovarajućih bitova u specijalnom kontrolnom registru procesora, dostupnom samo iz privilegovanog nivoa. Ovo postavljanje parametara vrši operativni sistem prilikom svoje inicijalizacije.

Struktura i raspored bitova stavki tabele stranica je takođe hardverski definisana i operativni sistem se mora sa tim uskladiti. Pored rednog broja okvira fizičke memorije koji datu stranicu sadrži, stavka tabele stranica obično sadrži i druge bitove koji, između ostalog, određuju i nivo privilegija potreban za pristup toj stranici. Na primer, moguće je sprečiti pristup toj stranici od strane korisničkog programa, ako ona sadrži neke osetljive podatke operativnog sistema. Slično, moguće je podesiti da stranica bude nepromenljiva (engl. *read-only*), što je korisno za stranice koje sadrže programski kod. Na ovaj način hardver omogućava fino podešavanje prava pristupa memoriji od strane procesa. S obzirom da se i ulazno-izlazni kontroleri obično mapiraju u adresni prostor procesora, na ovaj način se može obezbediti i kontrola pristupa registrima ulazno-izlaznih kontrolera.

5.1.5.3 Sistem prekida

Sistem prekida predstavlja mehanizam koji omogućava zaustavljanje programa bez njegove volje u proizvoljnom trenutku i prenošenje kontrole operativnom sistemu. Ovo je vrlo važno u kontekstu operativnih sistema, kako bi on mogao da obavlja svoje funkcije. Bez sistema prekida ne bi bilo moguće zaustaviti program kada on jednom krene da se izvršava, osim da isključimo računar ili da sačekamo da on sam završi sa svojim radom. Potreba za prekidanjem programa javlja se u različitim situacijama.

Prvi slučaj je vezan za obradu asinhronih događaja izazvanih interakcijom sa ulazno-izlaznim uređajima računara. Na primer, takvi događaji nastaju pritiskom tastera na tastaturi, pomeranjem miša ili priključivanjem uređaja na USB podsistem. Ove događaje obično izaziva korisnik i mogu se desiti u bilo kom trenutku. Kada se dese, potrebno ih je obraditi u što kraćem roku od strane odgovarajućeg drajvera uređaja u okviru ulazno-izlaznog podsistema. Dva su razloga zbog čega je obrada ovakvih događaja hitna. Prvo, korisnik očekuje brzu reakciju sistema na svoje komande, tj. zelimo da sistem bude interaktivan i komunicira sa korisnikom u realnom vremenu. Drugo, memorijski prostor koji ulazno-izlazni kontroleri sadrže je obično ograničen i podaci pristigli od ulaznog uređaja ne mogu biti dugo sačuvani u registrima kontrolera, tj. postoji mogućnost njihovog gubitka. Zbog toga ih je potrebno što pre prebaciti u memoriju i smestiti u odgovarajuće strukture operativnog sistema koji će ih dalje obrađivati. Sistem prekida omogućava ulazno-izlaznim uređajima da privuku pažnju procesora kada se takav događaj desi, tako što aktiviraju poseban *signal prekida* koji se dovodi do procesora podsredstvom posebnog ulaznog priključka. Procesor nakon svake instrukcije proverava da li je ovaj signal aktiviran i, ako jeste, automatski zaustavlja dalje izvršavanje tekućeg procesa, pamti stanje registara u kontrolnom bloku procesa i zatim poziva unapred datu proceduru operativnog sistema koja je zadužena za *obradu prekida*. Prilikom izvršavanja ove procedure, procesor obično prelazi u privilegovani režim rada (ovo je često jedini način da se iz neprivilegovanog pređe u privilegovani režim rada, a dešava se automatski prilikom izazivanja prekida). Nakon obrade prekida, procesor se vraća u neprivilegovani režim rada i vraća kontrolu prekinutom procesu, tako što iz njegovog kontrolnog bloka kopira sačuvane vrednosti registara procesora (uključujući i programski brojač). Opisani mehanizam prekida spada u tzv. *hardverske prekide*, jer ih izaziva hardver, asinhrono, kao reakciju na spoljni događaj.

Drugi slučaj vezan je za implementaciju algoritma raspoređivanja procesa, u slučaju da algoritam podrazumeva prekidanje (za potrebe implementacije podele vremena). Tada je potrebno pustiti proces da se izvršava a zatim ga prekinuti kada istekne vremenski kvantum koji mu je dodeljen, kako bi drugi proces mogao da dobije priliku da se izvršava. Opisani proces je takođe omogućen zahvaljujući hardverskim prekidima. Obično u računarskom sistemu postoji poseban kontroler koji se naziva *programibilni tajmer* (engl. *programmable interval timer* (PIT)). Ovaj uređaj se može posebnim komandama programirati tako da generiše periodični signal za prekid u ravnomernim intervalima. Ovo obično obavi operativni sistem u fazi inicijalizacije. Zahvaljujući ovom periodičnom signalu koji pristiže u ravnomernim intervalima garantovano je da će programi biti prekidani, a kontrola biti vraćana operativnom sistemu nakon isteka programiranog vremenskog kvantuma. Operativni sistem obrađuje ovaj prekid tako što pokreće algoritam raspoređivanja i određuje sledeći proces koji treba da nastavi sa izvršavanjem.

Treći slučaj je mehanizam implementacije sistemskih poziva. Setimo se da se sistemskim pozivima zahteva od jezgra sistema da za potrebe i u ime pozivajućeg procesa obavi neku sistemsku funkciju ili obezbedi neki resurs procesu. Ove funkcionalnosti mora obavljati jezgro sistema i mora ga obavljati u privilegovanom režimu rada. Međutim, korisnički proces se izvršava u neprivilegovanom režimu, tako da je neophodno obezbediti kontrolisani prelazak u privilegovani režim prilikom poziva kôda jezgra operativnog sistema. Ovo se obavlja tako što se posebnom instrukcijom procesora namerno izazove prekid. Ovakvi prekidi se nazivaju *softverski prekidi*, jer ih izaziva sam programski kod procesa koji se trenutno izvršava. Dakle, proces sam sebe namerno prekida da bi prepustio kontrolu operativnom sistemu i omogućio prelazak u privilegovani režim rada. Operativni sistem tada izvršava sistemski poziv i nakon toga vraća kontrolu samoprekinutom procesu (vraćajući pritom procesor u neprivilegovani režim rada).

Četvrti slučaj predstavljaju situacije u kojima neka instrukcija procesa iz nekog razloga ne može uspešno da se izvrši. Razlog može da bude nedefinisana operacija (poput deljenja nulom), nedozvoljeni pristup nekom resursu (npr. pristup registru ili pozivanje instrukcije koja nije dostupna u trenutnom nivou privilegija), ili nedostupnost nekog resursa (poput ranije opisane greške straničenja). U takvim situacijama se automatski generiše prekid, a kontrola se predaje operativnom sistemu koji u privilegovanom režimu rada pokušava da obradi nastalu grešku. Ovakvi prekidi poznati su i pod nazivom *izuzetci* (engl. *exceptions*). Neki izuzetci se mogu uspešno razrešiti (poput greške straničenja), nakon čega se kontrola vraća prekinutom procesu. Neki drugi se obično ne mogu razrešiti (poput pristupa nedozvoljenom resursu), što za rezultat obično ima likvidaciju procesa.

5.2 Operativni sistemi zasnovani na UNIX-u

Operativni sistem *UNIX* (čita se *juniks*) nastao je u Belovim laboratorijama davne 1969. godine. Originalni autor bio je Ken Tompson[4]. Prve verzije UNIX-a programirane su na asemblerskom jeziku računara PDP-11 i samo su se na tom računaru mogle prevoditi i izvršavati. Par godina kasnije, Tompsonov kolega iz Belovih laboratorija Denis Riči razvio je programski jezik C koji je, za razliku od drugih programskih jezika višeg nivoa koji su postojali u to vreme, bio pogodan za sistemsko programiranje, s ozbirom da je omogućavao pristup memorijskim adresama i hardveru na niskom nivou. Operativni sistem UNIX je tada ponovo isprogramiran na programskom jeziku C, što je omogućilo njegovo prevođenje i instalaciju na drugim hardverskim platformama. Tako je UNIX postao prvi operativni sistem koji je nadživeo računar za koji je inicijalno razvijen i zahvaljujući tome uspeo da u različitim oblicima preživi do današnjih dana (u trenutku pisanja ovih redova, to je punih 55 godina!).

Zahvaljujući fleksibilnim licencama pod kojima je inicijalno bio objavljen, UNIX se uspešno širio u okviru stručne i akademske zajednice. Različiti autori su ga dalje unapređivali, a nastajale su i potpuno nove verzije ovog sistema, među kojima je najpoznatija bila verzija nastala na univerzitetu u Berkliju, poznata kao *BSD UNIX* (engl. *Berkeley Software Distribution*). Neki derivati BSD sistema (poput sistema *FreeBSD*, *OpenBSD*, *NetBSD* i sl.) se i danas aktivno koriste na savremenim računarima.

Početkom 80tih godina XX veka UNIX licence postaju restriktivnije, te slobodni razvoj i upotreba ovog sistema od strane zajednice postaje otežana. Počinju da dominiraju komercijalne varijante sistema, poput HP-UX (kompanija HP), AIX (kompanija IBM), Solaris (Kompanija Sun) i Xenix (kompanija Majkrosoft). Kao protivteža ovakvom scenariju nastaje projekat *GNU* (engl. *GNU is Not Unix* (GNU)) sa ciljem razvoja operativnog sistema kompatibilnog sa UNIX-om koji će garantovano zauvek biti distribuiran pod slobodnom licencom, eliminišući time zavisnost od komercijalnih interesa kompanija koje su stajale iza drugih verzija UNIX-a. U okviru GNU projekta su najpre razvijeni programerski alati (editor, prevodilac, debager, linker, i sl.), kao i kompletno korisničko okruženje operativnog sistema (biblioteke, korisnički interfejs, osnovni alati za rad sa sistemom datoteka i sl.). Ono što je nedostajalo bio je najkomplikovaniji deo operativnog sistema – jezgro. Taj deo upotpunjen je pojavom *Linux* jezgra koji je početkom 90tih godina razvio finski programer Linus Torvalds. Ugradnjom Linux jezgra u GNU okruženje nastaje *GNU/Linux* operativni sistem[5]. Postoje različite *distribucije* GNU/Linux operativnog sistema koje se međusobno razlikuju pre svega po izboru softvera koji dolazi sa sistemom, kao i po inicijalnoj konfiguraciji. Neke od najpoznatijih su *Ubuntu*, *Mint*, *Debian*, *Fedora*, *Slackware* i sl.

5.3 Komandno okruženje operativnog sistema UNIX

Tradicionalni korisnički interfejs za UNIX-zasnovane operativne sisteme je *ljuska* (engl. *shell*). U pitanju je tekstualni korisnički interfejs koji zahvaljujući svojoj bogatoj sintaksi nudi ogromne mogućnosti korisniku. Ljuska omogućava jednostavno pokretanje komandi sa zadavanjem opcija komandne linije, ali i složenije načine pokretanja programa koji koriste preusmeravanje standardnog ulaza i izlaza, kao i nadovezivanje više programa koristeći *cevi* (engl. *pipe*). Ljuska takodje uključuje i sopstveni programski jezik koji omogućava korisniku da isprogramira složene postupke za rešavanje kompleksnijih zadataka, koji uključuju pokretanje većeg broja programa, rad sa fajlovima i sl. Zahvaljujući ljusci i njenim mogućnostima, korisnik može da kombinujući brojne, mahom jednostavne alate koji su mu dostupni obavlja veoma složene administrativne i druge poslove.

5.3.1 Pokretanje i prvi koraci

Pre pojave grafičkih korisničkih interfejsa, ljuska je bila podrazumevani korisnički interfejs na UNIX-zasnovanim sistemima. Otuda se ljuska pokretala automatski, nakon prijave korisnika na sistem. U današnje vreme, većina UNIX-zasnovanih sistema su podešeni tako da grafičko okruženje bude podrazumevano. Ipak, većina iskusnih UNIX korisnika će pre ili kasnije poželeti da se dokopa ljuske, kako bi mogli da obavljaju različite rutinske poslove. U tu svrhu se obično pokreće neka aplikacija sa grafičkim interfejsom koja predstavlja *emulator terminala* (često se zove *Terminal* ili *Console*).[6] Pokretanjem ove aplikacije otvara se prozor u okviru koga se pokreće i prikazuje ljuska. Prozor emulatora terminala zauzima deo ekrana koji kada je u fokusu preuzima vezu sa tastaturom i omogućava korisniku da komunicira sa ljuskom koja je u njemu pokrenuta, emulirajući tako fizički terminal.

[4] Pored UNIX-a, ime Kena Tompsona se vezuje i za UTF-8 kodiranje teksta, kao i za tzv. *Tompsonove automate* u teoriji formalnih jezika.

[5] Vrlo često ga nazivamo i samo *Linux*, iako je to pogrešno, s obzirom da je Linux samo jezgro koje je bez ostalih komponenti razvijenih u okviru GNU projekta potpuno neupotrebljivo.

[6] Pojam *terminal* tradicionalno označava tastaturu i monitor pomoću kojih jedan korisnik komunicira sa računarom. Računar može imati više terminala, što omogućava istovremeni rad više korisnika.

Elektronska verzija (2024)

Postoji više različitih vrsta ljuski koje su međusobno veoma slične, a razlikuju se pre svega u nekim detaljima sintakse. Na sistemu može biti instalirano više ljuski, a korisnik može odabrati svoju podrazumevanu ljusku koja će se automatski pokretati. Neke od poznatijih ljuski su *sh* (i iz nje izvedena ljuska *bash*), *csh*, *tcsh* i sl. Kako je *bash* podrazumevana ljuska koji se koristi pod GNU/Linux sistemima, sve što u nastavku sledi odnosiće se upravo na *bash* ljusku.

5.3.1.1 Odziv ljuske

Bez obzira na koji način je ljuska pokrenuta, korisniku se najpre prikazuje *odziv* (engl. *prompt*) ljuske, poput:

```
pera@desktop:programiranje$
```

Znak $ na kraju odziva naziva se *odzivni znak*. Tekst koji stoji ispred njega sadrži dodatne informacije i može se konfigurisati po želji korisnika. U našem primeru, ovaj tekst sadrži redom ime korisnika (`pera`), ime računara (`desktop`) i ime tekućeg direktorijuma (`programiranje`).

5.3.1.2 Jednostavno pokretanje programa. Putanja i promenljive okruženja

U nastavku odziva od korisnika se očekuje da unese komandu. Pritiskom na taster ENTER, komanda se analizira od strane ljuske, i ako ne postoje sintaksne greške, komanda se pokreće. Neke jednostavne komande su ugrađene u samu ljusku i izvršavaju se od strane samog procesa ljuske. Ipak, većina komandi podrazumeva pokretanje nekog drugog programa sa diska računara. U tom slučaju će proces ljuske kreirati novi proces i u okviru njega pokrenuti dati program. Uobičajena sintaksa pokretanja komande je:

```
naziv_programa arg1 arg2 ...
```

Pritom, `naziv_programa` predstavlja ime programa koji treba pokrenuti, dok su `arg1`, `arg2`, ... opcioni *argumenti komandne linije* koji se predaju programu. Ime programa predstavlja ime *izvršnog fajla* u kome se obično nalazi mašinski kôd programa, ili kôd napisan na nekom drugom programskom jeziku za koji postoji odgovarajući interpretator (to može biti programski jezik same ljuske ili neki drugi jezik poput jezika *Perl* ili *Python*). Po konvenciji, izvršni fajlovi na sistemu UNIX nemaju ekstenziju, ali korisnik mora imati dozvolu za pokretanje fajla. Kako bi ljuska mogla da na disku pronađe i pokrene program sa tim imenom, mora da zna putanju do direktorijuma u kome se izvršni fajl nalazi. U tu svrhu ljuska ima definisanu listu direktorijuma koje treba pretražiti, a koja se naziva *putanja*. Putanja se čuva kao vrednost *promenljive okruženja* (engl. *environment variable*) koja se zove `PATH`. Vrednost ove promenljive može se očitati na sledeći način:

```
echo $PATH
```

Komanda `echo` prosto ispisuje tekst koji joj se daje kao argument komandne linije. Sintaksa `$PATH` označava vrednost promenljive koja je navedena iza znaka `$` (kada bismo izostavili znak `$`, ljuska bi doslovno ispisala tekst `PATH`). Na primer, ispis gornje komande mogao bi da izgleda ovako:

```
/usr/local/bin:/usr/bin:/bin:/usr/games
```

Dakle, u pitanju su apsolutne putanje direktorijuma razdvojene dvotačkom. Kada pokrećemo komandu, ljuska redom pretražuje navedene direktorijume i pokreće prvi program sa tim imenom koji pronađe. Vrednost promenljive `PATH` se može podešavati, npr.:

```
PATH=$PATH:/usr/sbin
```

Na ovaj način se uzima tekuća vrednost promenljive `PATH` i na nju se dopisuje niska `:/usr/sbin`. Tako dobijena niska se postavlja za novu vrednost promenljive `PATH`. Sada bi `echo` komanda od malopre ispisala:

```
/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/sbin
```

Dakle, sada imamo jedan dodatni direktorijum u kome ljuska može tražiti izvršne fajlove pri pokretanju komandi. Napomenimo da će ova promena biti aktivna samo tokom tekuće sesije. Kada zatvorimo ljusku i pokrenemo je ponovo, ponovo ćemo imati originalnu vrednost promenljive `PATH`. Naime, promenljiva `PATH` se, kao i druge promenljive okruženja, prilikom pokretanja ljuske postavlja na neku podrazumevanu vrednost koja je zadata u odgovarajućim konfiguracionim fajlovima. Samim tim, svaki put kada pokrenemo ljusku, promenljiva `PATH` će biti postavljena na istu početnu vrednost. Promene koje mi „ručno" vršimo u okviru ljuske biće aktivne samo

Elektronska verzija (2024)

do kraja tekuće sesije. Ako želimo da trajno promenimo podrazumevanu vrednost `PATH` promenljive, moramo da promenimo odgovarajuće konfiguracione fajlove (što prevazilazi okvire ovog teksta).

Promenljiva `PATH` je samo jedna od promenljivih okruženja. Postoje i druge promenljive i svaka od njih ima neko posebno značenje za ljusku. Na primer, promenljiva `PWD` sadrži putanju do tekućeg direktorijuma ljuske, promenljiva `HOME` sadrži putanju do početnog direktorijuma prijavljenog korisnika, promenljiva `USER` sadrži ime prijavljenog korisnika, i td. Takođe, korisnik može definisati nove promenljive okruženja koje neće imati nikakvo predefinisano značenje za ljusku, ali će imati značenje za korisnika i programe koje će u nastavku pokretati. Na primer:

```
export BOJA=plava
```

Ovim se kreira nova promenljiva okruženja `BOJA` i dodeljuje joj se vrednost `plava`. Ključna reč `export` neophodna je prilikom inicijalnog kreiranja promenljive, kako bi ona postala deo okruženja ljuske (inače bi bila samo lokalna promenljiva ljuske). Vrednost ove promenljive se kasnije može menjati (sintaksom npr. `BOJA=crvena`) ili očitavati njena vrednost (sintaksom `$BOJA`). Spisak svih trenutno definisanih promenljivih okruženja može se dobiti komandom:

```
export
```

Važno je napomenuti da se promenljive okruženja automatski prenose procesima koje ljuska kreira, tako da će svi programi koje pokrenemo iz ljuske imati na raspolaganju sve promenljive koje su nasleđene iz ljuske. Operativni sistem obezbeđuje interfejs kojim se u okviru korisničkih programa mogu očitavati vrednosti promenljivih okruženja, a konkretna sintaksa zavisi od programskog jezika u kom je program napisan. Na ovaj način, promenljive okruženja se mogu koristiti kao mehanizam komunikacije između ljuske i programa koji se u okviru nje pokreću.

Ukoliko se program koji pokrećemo ne nalazi ni u jednom od direktorijuma navedenim u promenljivoj `PATH`, tada kažemo da program nije *u putanji*. Ako program nije u putanji, ljuska neće moći da ga pokrene i prikazaće grešku, npr.:

```
abcd
```

Pokretanjem nepostojećeg programa `abcd` dobijamo poruku:

```
bash: abcd: command not found
```

Ponekad se program koji želimo da pokrenemo nalazi u nekom drugom direktorijumu koji nije u putanji. Kako bi ljuska mogla da ga pronađe, možemo navesti apsolutnu ili relativnu putanju do izvršnog fajla:

```
/home/pera/programiranje/moj_program arg1 arg2
```

Dakle, umesto da navedemo samo ime programa `moj_program`, navodimo celu putanju. Ovo je znak za ljusku da ne treba da traži program u predefinisanim direktorijumima iz promenljive `PATH`, već mu je eksplicitno data putanja u kojoj se program nalazi. Ako je tekući direktorijum ljuske `/home/pera`, tada možemo navesti i relativnu putanju:

```
programiranje/moj_program arg1 arg2
```

Najzad, ako je tekući direktorijum ljuske `/home/pera/programiranje`, tada možemo pokrenuti komandu:

```
./moj_program arg1 arg2
```

Ovde je bitno primetiti da pre naziva programa stoji `./`. Naime, tačka (`.`) predstavlja oznaku tekućeg direktorijuma. Samim tim, `./moj_program` predstavlja relativnu putanju u odnosu na tekući direktorijum. Iako je i `moj_program` takođe ispravno zadata relativna putanja u odnosu na tekući direktorijum, komanda:

```
moj_program arg1 arg2
```

bi izazvala grešku, jer bi ljuska smatrala da se radi o programu koji je u putanji, pa bi pretraživala direktorijume iz promenljive `PATH`. Navođenjem `./` ispred naziva programa mi govorimo ljusci da program ne treba tražiti u putanji, već da ga treba tražiti u tekućem direktorijumu.

Elektronska verzija (2024)

5.3.1.3 Povratna vrednost programa

Od svakog programa se očekuje da ljusci vrati *povratnu vrednost*. U pitanju je ceo broj koji ljusci (ili korisniku) može pružiti informacije o tome kako je protekao rad programa. Po konvenciji, vrednost 0 označava da je sve prošlo u redu, dok vrednosti različite od nule predstavljaju različite kôdove grešaka. Tačno značenje kôda greške zavisi od programa i obično se može pronaći u pratećoj dokumentaciji.

Način na koji program vraća vrednost ljusci zavisi od programskog jezika u kome je program napisan. Na primer, programi pisani u programskim jezicima C ili C++ vraćaju povratnu vrednost svoje glavne (*main*) funkcije.

Da bi korisnik očitao povratnu vrednost programa čije je izvršavanje upravo završeno u okviru ljuske, može pokrenuti komandu:

```
echo $?
```

5.3.1.4 Argumenti komandne linije

Argumenti komandne linije predstavljaju dodatne informacije koje se prosleđuju pokrenutom programu. U okviru programa postoji mogućnost pristupa argumentima komandne linije, pri čemu konkretna sintaksa ponovo zavisi od samog programskog jezika u kome je program napisan. Značenje argumenata komandne linije zavisi od samog programa, dok ljuska argumente vidi prosto kao niske simbola i ne pridaje im nikakvo značenje. Ljuska podrazumevano smatra da su argumenti komandne linije međusobno razdvojeni razmacima. Na primer:

```
moj_program prvi drugi treci
```

U ovom primeru, ljuska će smatrati da postoje tri argumenta komandne linije i programu `moj_program` će prilikom pokretanja predati niz koji sadrži tri niske: `prvi`, `drugi` i `treci`. Ukoliko želimo da neki argument komandne linije sadrži razmak, tada je neohodno navesti argument pod apostrofima:

```
moj_program 'prvi argument' drugi treci
```

Sada ponovo imamo tri argumenta: `prvi argument`, `drugi` i `treci`. Dakle, prvi argument se sastoji iz dve reči. Da smo zaboravili da stavimo apostrofe:

```
moj_program prvi argument drugi treci
```

ljuska bi smatrala da imamo četiri argumenta komandne linije, redom `prvi`, `argument`, `drugi` i `treci`. Umesto apostrofa, moguće je koristiti i navodnike:

```
moj_program "prvi argument" drugi treci
```

U datom primeru, efekat će biti potpuno isti. Razlika između apostrofa i navodnika je u tome što navodnici dopuštaju *ekspanziju promenljivih*, tj. zamenu imena promenljive njenom vrednošću. Na primer:

```
echo "Moj tekuci direktorijum je $PWD"
```

U ovom primeru će najpre `$PWD` biti zamenjeno vrednošću promenljive `PWD` (npr. `/home/pera/programiranje`), a zatim će tako dobijeni tekst biti prosleđen kao (jedini) argument komandne linije programu `echo`. Ovo je isto kao da smo pozvali komandu:

```
echo "Moj tekuci direktorijum je /home/pera/programiranje"
```

što kao efekat ima ispis:

```
Moj tekuci direktorijum je /home/pera/programiranje
```

Sa druge strane, da smo napisali:

```
echo 'Moj tekuci direktorijum je $PWD'
```

ljuska ne bi vršila ekspanziju promenljive, pa bi tekst `$PWD` ostao nepromenjen i kao takav bi bio prosleđen programu `echo` kao deo argumenta komandne linije. Efekat ove komande bio bi ispis teksta:

```
Moj tekuci direktorijum je $PWD
```

Elektronska verzija (2024)

Na kraju, posmatrajmo još i sledeći primer:

```
moj_program "Moja boja: $BOJA"
```

U ovom slučaju će `$BOJA` biti zamenjeno vrednošću ove promenlive (npr. `plava`), a onda će ceo tekst `Moja boja: plava` biti prosleđen programu kao *jedan* argument komandne linije. Sa druge strane:

```
moj_program Moja boja: $BOJA
```

će takođe proizvesti ekspanziju promenljive, ali će sada tekst `Moja boja: plava` biti prosleđen programu kao niz od *tri* argumenta komandne linije: `Moja`, `boja:` i `plava`. Dakle, ekspanzija promenljivih se vrši i kada nema navodnika, ali navodnici omogućavaju da se više reči razdvojenih razmacima tretiraju kao jedan argument, umesto da ih ljuska posmatra kao posebne argumente komandne linije.

Pored ekspanzije promenljivih, ljuska omogućava i druge vrste ekspanzija u okviru komandne linije. Na primer, simboli `*` i `?` omogućavaju zadavanje putanja do svih fajlova čiji se nazivi uklapaju u neki obrazac. Posmatrajmo komandu:

```
cat *.html
```

Program `cat` prihvata na komandnoj liniji jednu ili više putanja do fajlova, a zatim sadržaje svih tih fajlova redom izlistava na standardnom izlazu. U ovom primeru će se sadržaji svih fajlova sa ekstenzijom `.html` prikazati na standardnom izlazu. Naime, sintaksa `*.html` se od strane ljuske automatski razvija u listu svih fajlova čija se putanja uklapa u dati obrazac, razdvojenih razmacima. U našem primeru, to su svi fajlovi u tekućem direktorijumu čiji se naziv završava sa `.html`. Simbol `*`, dakle, predstavlja nula ili više proizvoljnih karaktera. Slično, mogli smo navesti:

```
cat /home/pera/*/index.html
```

Ovog puta se argument `/home/pera/*/index.html` zamenjuje listom svih putanja tog oblika. Ako nema ni jednog takvog fajla, tada navedeni argument ostaje nepromenjen, tj. nema ekspanzije. Pored simbola `*`, moguće je navesti i simbol `?`. Ovaj simbol predstavlja tačno jedan proizvoljan karakter. Sada se pri pozivu komande:

```
cat /home/pera/*/index?.html
```

navedeni argument zamenjuje listom svih postojećih putanja tog oblika, gde se `*` zamenjuje sa nula ili više proizvoljnih karaktera, a `?` tačno jednim proizvoljnim karakterom. Najzad, komanda:

```
cat /home/pera/{index,map}.html
```

će na standardnom izlazu prikazati sadržaje fajlova `index.html` i `map.html` iz direktorijuma `/home/pera`. Naime, sintaksa `/home/pera/{index,map}.html` se razvija u listu sastavljenu od dva argumenta `/home/pera/index.html` i `/home/pera/map.html` razdvojenih razmakom.

Navedene ekspanzije spadaju u tzv. *ekspanzije putanje* (engl. *path expansion*). Pored ovih ekspanzija, postoji i veliki broj drugih, kako ekspanzija, tako i drugih sintaksnih mogućnosti ljuske poput komandnih ekspanzija, evaluacije izraza, složenih naredbi i sl., čiji detaljni opis prevazilazi okvire ovog teksta. Zajedničko za sve ove sintaksne elemente ljuske je da podrazumevaju upotrebu simbola koji imaju specijalno značenje za ljusku. Takvi simboli su, na primer, `$`, `*`, `?`, `{`, `}`, `(`, `)`, `[`, `]`, `!`, `;`, `|`, `<`, `>` i sl. Kada ljuska u nazivu komande ili u okviru argumenata komandne linije uoči neki od ovih simbola, ona ih tumači u skladu sa njihovim specijalnim značenjem. Upotreba apostrofa sprečava specijalno tumačenje simbola, tj. primorava ljusku da sve ove simbole tretira kao obične karaktere, te da odgovarajuće argumente komandne linije prenese programu doslovno, bez ikakvih promena. Na primer, ako kao argument komandne linije navedemo `'$PWD'` ili `'/etc/*.conf'`, neće doći do odgovarajuće ekspanzije, zato što koristimo apostrofe, pa će simboli `$` i `*` biti tumačeni kao obični karakteri. Za razliku od apostrofa, unutar navodnika pojedini specijalni simboli (poput simbola `$`, ali ne i simbola `*` i `?`) zadržavaju specijalno značenje. Zato će se `"$PWD"` razviti u vrednost promenljive `PWD`, dok će `"/etc/*.conf"` ostati takav kakav je, bez ekspanzije. Navedimo još jedan zanimljiv primer:

```
grep '[a-zA-Z]*[0-9]{2}' ulaz.txt
```

Elektronska verzija (2024)

Program `grep` se koristi za pronalaženje uzoraka u tekstu. Prvi argument komandne linije opisuje uzorak koji se traži na jeziku tzv. *regularnih izraza* čiji detaljan opis ovde izostavljamo, zbog nedostatka prostora. U ovom primeru, tražimo sve uzorke koji se sastoje iz nula ili više malih ili velikih slova za kojima slede tačno dve cifre. Drugi argument je putanja do fajla koji se pretražuje. Program ispisuje sve linije fajla koje sadrže traženi obrazac. Primetimo da smo prvi argument programa smestili između apostrofa. Ovo je zato što bi inače različiti specijalni simboli u sintaksi obrazca (poput `[`, `-`, `]`, `{`, `}`) bili tumačeni kao specijalni simboli ljuske. Mi to ne želimo, već želimo da se navedeni obrazac doslovno preda programu `grep` koji će ga dalje tumačiti u skladu sa svojim internim pravilima.

5.3.1.5 Tekući direktorijum

Tekući direktorijum ljuske je takođe važan parametar koji se prenosi na procese pokrenute u okviru ljuske. Sve relativne putanje se uvek odnose na ovaj direktorijum. Tekući direktorijum se prilikom pokretanja ljuske postavlja na početni korisnički direktorijum (dat promenljivom `HOME`). U toku rada, možemo ga promeniti komandom

```
cd <putanja>
```

gde je `<putanja>` apsolutna ili relativna putanja do direktorijuma za koji želimo da postane novi tekući direktorijum. Na primer, ako se nalazimo u direktorijumu `/home/pera` i želimo da promenimo tekući direktorijum na `/home/pera/programiranje`, možemo zadati komandu:

```
cd /home/pera/programiranje
```

tj. da zadamo apsolutnu putanju, ili:

```
cd programiranje
```

čime zadajemo relativnu putanju. U oba slučaja, efekat će biti isti. Ako pretpostavimo da je `/home/pera` početni direktorijum prijavljenog korisnika, tada je vrednost promenljive `HOME` upravo `/home/pera`. To znači da možemo napisati i:

```
cd $HOME/programiranje
```

i efekat će, nakon ekspanzije promenljive, biti isti. Najzad, pokretanje komande `cd` bez argumenata za efekat ima vraćanje na početni direktorijum korisnika. Dakle, komanda:

```
cd
```

je ekvivalentna komandi:

```
cd $HOME
```

Tekući direktorijum se uvek može očitati bilo prikazom vrednosti promenljive `PWD`:

```
echo $PWD
```

bilo pozivom komande:

```
pwd
```

Elektronska verzija (2024)

5.3.1.6 Preusmeravanje standardnog ulaza i izlaza

Svaki program može učitavati podatke sa *ulaza*, kao i saopštavati rezultate svog rada na *izlazu*. Ulaz i izlaz mogu biti fajlovi na disku koji su otvoreni za čitanje, odnosno pisanje od strane operativnog sistema. Specijalno, postoji *standardni ulaz* koji je podrazumevano povezan sa tastaturom, kao i *standardni izlaz* koji je povezan sa ekranom monitora. Kada program pokuša da čita sa standardnog ulaza, dolazi do prekida rada procesa (tj. proces odlazi u stanje *čeka*), a operativni sistem podsredstvom drajvera od kontrolera tastature zahteva da mu prosledi unos koji korisnik unosi. Kada korisnik unese tekst i pritisne ENTER, uneti tekst se prosleđuje procesu i on ponovo može da nastavi sa izvršavanjem (tj. vraća se u stanje *spreman*). Slično, kada program piše na standardni izlaz, odgovarajući tekst se prikazuje na ekranu.

Jedna od značajnih karakteristika ljuske je da ona omogućava *preusmeravanje standardnog ulaza/izlaza* (engl. *standard input/output redirection*). Ovim mehanizmom se standardni ulaz/izlaz može preusmeriti na neki fajl po želji korisnika. Na primer:

```
moj_program < ulaz.txt
```

Simbol `<` označava preusmeravanje standardnog ulaza, a iza njega se očekuje putanja (apsolutna ili relativna) do fajla na koji se preusmerava standardni ulaz. Ovo znači da će svaki pokušaj čitanja sa standardnog ulaza unutar programa za efekat imati čitanje iz fajla `ulaz.txt`, a ne sa tastature. Slično:

```
moj_program > izlaz.txt
```

preusmerava standardni izlaz na fajl koji je zadat putanjom koja se navodi nakon simbola `>`. Ovo znači da će svaki ispis na standardni izlaz unutar programa za efekat imati pisanje u fajl `izlaz.txt` umesto na ekran. Moguće je istovremeno preusmeriti i ulaz i izlaz:

```
moj_program < ulaz.txt > izlaz.txt
```

Kada je u pitanju preusmeravanje standardnog izlaza, podrazumevano ponašanje je da se navedeni fajl otvara za pisanje, a njegov prethodni sadržaj se briše. Ukoliko želimo da njegov prethodni sadržaj ostane nepromenjen, a da se novi sadržaj dopisuje na kraj fajla, možemo koristiti operator `>>` umesto `>`:

```
moj_program < ulaz.txt >> izlaz.txt
```

Pored standardnog izlaza, postoji i *standardni izlaz za greške*. Ovaj izlaz je takođe podrazumevano povezan sa ekranom i koristi se za saopštavanje poruka o greškama korisniku. Prilikom preusmeravanja, on se može posebno preusmeriti, tako da se poruke o greškama ispisuju u poseban fajl. Na primer:

```
moj_program < ulaz > izlaz.txt 2> greske.txt
```

Putanja do fajla na koji se preusmerava standardni izlaz za greške se navodi nakon operatora `2>` (ili `2>>`, u slučaju dopisivanja sadržaja). Primetimo da ako samo navedemo:

```
moj_program < ulaz.txt > izlaz.txt
```

tada će se samo standardni izlaz preusmeravati na `izlaz.txt`, dok će se poruke o greškama saopštene preko standardnog izlaza za greške i dalje prikazivati na ekranu. Dakle, standardni izlaz i standardni izlaz za greške se uvek posebno preusmeravaju.

U svim prethodnim primerima, fajlovi `ulaz.txt` i `izlaz.txt` su se nalazili u tekućem direktorijumu. Moguće je navoditi i složenije putanje koje vode ka fajlovima u drugim direktorijumima. Na primer:

```
moj_program < programiranje/ulaz.txt
```

preusmerava ulaz na fajl `ulaz.txt` u poddirektorijumu `programiranje` tekućeg direktorijuma. Slično:

```
moj_program > ../razno/izlaz.txt
```

preusmerava standardni izlaz na fajl `izlaz.txt` koji se nalazi u okviru direktorijuma `razno` koji se nalazi u roditeljskom direktorijumu tekućeg direktorijuma. Dakle, oznaka `..` u putanji uvek označava roditeljski direktorijum tekućeg direktorijuma (setimo se da je oznaka `.` označavala tekući direktorijum).

Elektronska verzija (2024)

5.3.1.7 Opcije programa

Kao što je ranije rečeno, argumente komandne linije ljuska vidi kao obične niske i ne pridaje im nikakvo predefinisano značenje, već tumačenje tih argumenata ostavlja samom programu. Ipak, većina programa pod UNIX-om prati neke uobičajene konvencije kada su u pitanju argumenti komandne linije. Jedan poseban tip argumenata komandne linije koji prepoznaje većina programa i daje im poseban značaj jesu *opcije*. Opcijama se bliže određuje način rada programa. Kako bi se sintaksno razlikovale od ostalih argumenata komandne linije (poput fajlova ili komandi), obično počinju simbolom `-`. Na primer, komanda:

```
wc moj_fajl.txt
```

podrazumevano broji linije, reči i bajtove u fajlu koji je dat kao argument komandne linije. Međutim, ako zadamo opciju `-l`:

```
wc -l moj_fajl.txt
```

program će promeniti svoje ponašanje i brojaće samo linije. Program `wc` će znati da argument `-l` nije naziv fajla (što je uobičajeni prvi argument ove komande), već opcija, s obzirom da počinje sa `-`, pa će je tumači na odgovarajući način. Slično, postoje opcije `-w` (samo reči) i `-c` (samo bajtove). Ako navedemo:

```
wc -l -c moj_fajl.txt
```

dobićemo broj linija i broj bajtova, dok se reči neće brojati.

Opcije su obiņo kratke, sadrže jedno ili dva slova, što olakšava njihovo navođenje. Ipak, mnogi korisnici smatraju da je bolje da naziv opcije bude duži i opisniji, jer se tako lakše pamti, a komande postaju čitljivije. Ovakve, „dugačke" opcije se obično navode sa prefiksom `--`. Na primer, komanda `wc` ima i dugačke verzije opcija `-l`, `-w`, `-c`, a one glase `--lines`, `--words` i `--bytes`. Sada možemo navesti:

```
wc --lines --bytes moj_fajl.txt
```

Ova komanda će imati isti efekat kao prethodna, a izgleda jasnije i čitljivije (ali je i duža).

Napomenimo na kraju da iako se opisane konvencije u najvećoj meri poštuju od većine standardnih UNIX programa, ipak treba imati u vidu da je tumačenje argumenata komandne linije u potpunosti u domenu programa, tako da mogu postojati programi koji od ovih konvencija odstupaju. Kako biste se upoznali sa opcijama koje konkretan program prepoznaje i njihovim značenjem, najbolje je konsultovati dokumentaciju koja je isporučena uz program. Ova dokumentacija je obično dostupna u formi *stranica sa uputstvom za korišćenje* (engl. *manual pages*). Ove stranice se otvaraju pomoću programa `man`:

```
man wc
```

Program `man` izlistava uputstvo programa koji je naveden kao argument komandne linije. Uputstvo se može listati pomoću strelica za *gore*/*dole* (ili *PageUp*/*PageDown* tastera, u slučaju dugačkih uputstava). Izlazak iz `man` programa se postiže pritiskom na taster `Q`.

Inače, `man` stranice nisu dostupne samo za programe. Postoje `man` stranice i za funkcije standardne biblioteke programskog jezika C, kao i za sistemske pozive UNIX sistema. Na primer:

```
man strlen
```

otvara stranicu posvećenu funkciji `strlen()` standardne C biblioteke, dok komanda:

```
man fork
```

otvara stranicu posvećenu sistemskom pozivu `fork()`.

Više detalja o samom programu `man` možete pročitati iz njegove `man` strane:

```
man man
```

Elektronska verzija (2024)

5.3.1.8 Nadovezivanje programa

Nadovezivanje programa podrazumeva da se pokreće istovremeno više programa, pri čemu se standardni izlaz prvog preusmerava na standardni ulaz drugog, standardni izlaz drugog na standardni ulaz trećeg i td. Ovo se postiže mehanizmom *cevi* (engl. *pipe*) koji predstavlja jedan od najjednostavnijih mehanizama interprocesne komunikacije pod UNIX-om. Ljuska omogućava jednostavno nadovezivanje programa na sledeći način:

```
echo "Uvod u informatiku" | wc -w
```

Dakle, operatorom `|` možemo postići da se standardni izlaz komande `echo` preusmeri na standardni ulaz programa `wc`. Program `wc`, u odsustvu fajla navedenog na komandnoj liniji, svoj ulaz čita sa standardnog ulaza. Samim tim, izlaz programa `echo` (tj. tekst `Uvod u informatiku`) neće biti prikazan na ekranu, već će biti prosleđen programu `wc` koji će prebrojati reči u tom tekstu i dobijeni broj (3) prikazati na svom standardnom izlazu (koji nije preusmeren, pa će biti prikazan na ekranu). Pogledajmo još jedan primer:

```
cat primer.txt | head -7 | tail -5
```

Program `cat` ispisuje na standardnom izlazu sadržaj fajla koji je naveden kao argument komandne linije (u ovom primeru `primer.txt`). Međutim, njegov izlaz se preusmerava na standardni ulaz programa `head` koji ispisuje prvih 7 linija sa ulaza (opcija `-7`). Međutim, njegov izlaz je dalje preusmeren na standardni ulaz programa `tail`, koji ispisuje poslednjih 5 linija sa svog ulaza (opcija `-5`). Krajnji rezultat je da će deo sadržaja fajla `primer.txt` od treće do sedme linije biti prikazane na standardnom izlazu.

Evo još jednog primera:

```
cat primer.txt | sort | uniq
```

Ovog puta se sadržaj fajla `primer.txt` preusmerava na standardni ulaz programa `sort`, koji sortira linije fajla u rastućem leksikografskom poretku i ispisuje ih na standardni izlaz. Međutim, njegov izlaz je preusmeren na standardni ulaz programa `uniq` koji sa svog ulaza eliminiše susedne linije koje su identične. Na ovaj način smo iz fajla `primer.txt` izbacili duplikate linija, jer će nakon sortiranja duplikati svakako biti susedni, pa će ih `uniq` eliminisati. Dobijeni rezultat se ispisuje na standadnom izlazu.

Prikazani primeri pokazuju kako nadovezivanjem programa koji vrše jednostavne operacije nad ulazom možemo obavljati veoma složene poslove. UNIX okruženje uključuje veliki broj ovakvih jednostavnih alata i njihovo dobro poznavanje je preduslov za uspešno korišćenje nadovezivanja u cilju obavljanja rutinskih poslova u ljusci.

5.3.2 Upravljanje korisnicima UNIX sistema

Operativni sistem UNIX je višekorisnički sistem. Svaki korisnik ima sopstveno korisničko ime, identifikacioni broj (engl. *user ID* (UID)), lozinku, svoj osnovni korisnički direktorijum, kao i svoju podrazumevanu ljusku. UNIX sve korisnike deli u dve kategorije: *administrator* (sa korisničkim imenom `root`) i *obični korisnici*. Administrator ima ID jednak 0 i kao takav ima poseban tretman od strane UNIX sistema. On ima prava pristupa svim fajlovima i direktorijumima, kao i pravo da pokreće sve programe i izvršava sve operacije nad sistemom. Obični korisnici imaju ograničen pristup i mnoge operacije (poput nekih sistemskih poziva) im nisu dostupni. Takođe, u okviru sistema datoteka se mogu postaviti ograničenja pristupa fajlovima i direktorijumima za obične korisnike.

Pored korisnika, postoje i *grupe korisnika*. Svaki korisnik mora pripadati bar jednoj grupi (to je njegova *podrazumevana grupa*), i to se određuje prilikom kreiranja korisnika. Korisnik se naknadno može dodavati i u druge grupe po želji. Smisao grupa je u davanju dodatnih prava pristupa nad nekim fajlovima i drugim resursima za određeni skup korisnika. Grupa takođe ima svoje ime i identifikacioni broj (engl. *group ID* (GID)).

Nova grupa se može dodati komandom `groupadd`:

```
groupadd moja_grupa
```

Novi korisnici se mogu dodavati komandom `useradd`:

```
useradd pera -d /home/pera -m -g users -s /bin/bash -c "Pera Peric"
```

Ovom komandom se kreira korisnik čije je korisničko ime `pera`, UID je jednak prvom sledećem dostupnom broju, korisnički direktorijum je `/home/pera` (koji će biti kreiran ako već ne postoji, zahvaljujući opciji `-m`), podrazumevana grupa mu je `users` (koja bi trebalo da već postoji), podrazumevana ljuska mu je `/bin/bash`, a puno ime mu je `Pera Peric`.

Nakon kreiranja korisnika, potrebno je postaviti lozinku za korisnika. Ovo se može uraditi komandom:

Elektronska verzija (2024)

```
passwd pera
```

nakon čega je potrebno dva puta uneti željenu lozinku. Prilikom unosa lozinke ne prikazuje se ništa na ekranu, iz bezbednosnih razloga. Nakon toga, korisnik se može prijaviti na sistem koristeći tu lozinku. Nakon prijave (ili kada pokrene emulator terminala, u slučaju grafičkog korisničkog interfejsa), otvoriće mu se njegova podrazumevana ljuska, a tekući direktorijum biće njegov korisnički direktorijum.

Ako želimo da korisnik bude član i nekih dodatnih grupa, možemo prilikom kreiranja korisnika dodati i opciju `-G`:

```
useradd pera -d /home/pera -m -g users -G moja_grupa,wheel,audio -s /bin/bash
```

Ova komanda će korisnika dodatno učlaniti i u grupe `moja_grupa`, `wheel` i `audio`. Ove grupe nazivaćemo *suplementarne grupe* tog korisnika. Suplementarne grupe možemo menjati i naknadno:

```
usermod -a -G games
```

Ovom komandom ćemo dodatno korisnika učlaniti u grupu `games`. Listu svih suplementarnih grupa korisnika uvek možemo videti komandom:

```
groups
```

Neke od grupa imaju neko predefinisano značenje za sistem i bitno je dodati korisnika u te grupe da bi dobio neke određene privilegije. Na primer, grupa `wheel` obično omogućava korisniku da pod određenim uslovima izvršava komande kao administrator. Grupa `cdrom` omogućava korisniku da koristi CD i DVD diskove, dok grupa `plugdev` omogućava rad sa uređajima poput fleš memorija. Na ovaj način, administrator može fino kontrolisati šta ko od korisnika može da radi na sistemu.

Korisnik se može ukoloniti komandom:

```
userdel pera
```

Slično, grupa se može ukloniti komandom:

```
groupdel moja_grupa
```

Sve ove komande može pokretati isključivo `root` korisnik. Posledica je da običan korisnik ne može sam sebe dodavati u grupe, kreirati nove korisnike i davati im pristup i sl. Ovo je bitno iz bezbednosnih razloga. Izuzetak je komanda `passwd`, nju može pokretati i bilo koji korisnik, bez argumenata. Tom prilikom pokrenuće se procedura za promenu lozinke tog korisnika, pri čemu će se prvo očekivati od korisnika da unese svoju trenutnu lozinku.

Na računarski sistem može biti priključeno više terminala, a zahvaljujući umrežavanju, postoji i mogućnost udaljenog pristupa sistemu. To znači da u svakom trenutku možemo imati više korisnika koji su prijavljeni na sistem. Da bismo videli sve korisnike koji su trenutno prijavljeni, možemo pokrenuti komandu:

```
finger
```

Ista komanda se može koristiti i za dobijanje više informacija o korisniku, bez obzira da li je trenutno prijavljen ili ne:

```
finger pera
```

Najzad, spisak korisnika koji su se prijavljivali na sistem u prethodnom periodu može se videti komandom:

```
last
```

Elektronska verzija (2024)

5.3.2.1 Korisnici i procesi

Svaki proces ima svog *vlasnika*. Vlasništvo nad procesima se nasleđuje, tj. kada jedan proces pokrene drugi, dete proces će imati istog vlasnika kao i roditeljski proces. Prvi procesi koji se kreiraju od strane jezgra prilikom pokretanja sistema su u vlasništvu `root` korisnika. Kada se korisnik uspešno prijavi na sistem, njegov korisnički interfejs (ljuska ili grafičko okruženje) pokreće se u vlasništvu tog korisnika. Na dalje, svi procesi koje korisnik bude kreirao podsredstvom korisničkog interfejsa biće u njegovom vlasništvu. Vlasništvo nad procesom određuje prava koja proces ima, a koja odgovaraju pravima i privilegijama korisnika koji je vlasnik procesa.

Pored vlasnika, svaki proces ima i korisničku grupu kojoj pripada, kao i listu suplementarnih grupa. Ove informacije se takođe postavljaju prilikom prijavljivanja korisnika na sistem (na osnovu podrazumevane korisničke grupe korisnika i njegove liste suplementarnih grupa), a na dalje se nasleđuju od strane svih procesa koje korisnik pokrene putem korisničkog interfejsa. Informacije o grupama kojima proces pripada su bitne za operativni sistem kada odlučuje o tome da li proces ima prava da izvrši određenu operaciju. Ovo se pre svega odnosi na pristup fajlovima i direktorijumima, o čemu će biti više reči kasnije.

5.3.3 Rad sa sistemom datoteka

Svi sistemi datoteka se pod UNIX-om vide kao jedno jedinstveno stablo direktorijuma. Koreni direktorijum je označen kosom crtom (`/`). Svi sistemi datoteka *montiraju* se na neke od čvorova ovog stabla koje nazivamo *tačke montiranja*. Particija diska na kojoj je instaliran sam operativni sistem se montira u tački `/`, pa će koreni direktorijum sistema datoteka te particije odgovarati putanji `/`. U okviru ovog sistema datoteka možemo kreirati bilo koji prazan direktorijum, a zatim njega iskoristiti kao tačku montiranja za neki drugi sistem datoteka (na drugoj particiji ili drugom uređaju). U tu svrhu se koristi komanda `mount`:

```
mount -t ext4 /dev/sda2 /mnt/tacka
```

U ovom primeru, opcijom `-t ext4` zadat je *tip* sistema datoteka koji se montira (*ext4* je standardni tip sistema datoteka koji se koristi na GNU/Linux sistemima). Pretpostavka je da je sistem datoteka koji montiramo ispravno formatiran i da je tog tipa, inače montiranje neće biti uspešno. Argument `/dev/sda2` označava uređaj na kome se sistem datoteka nalazi. U ovom primeru, u pitanju je druga particija prvog hard diska. Treći argument `/mnt/tacka` je apsolutna putanja koja označava tačku montiranja. Ova putanja mora ukazivati na direktorijum koji postoji u okviru trenutnog stabla i trebalo bi da bude prazan. Nakon izvršavanja ove komande direktorijum `/mnt/tacka` više neće biti prazan, veće se u njemu videti sve ono što postoji u korenom direktorijumu sistema datoteka uređaja koji je montiran u toj tački. Drugim rečima, stablo direktorijuma montiranog sistema datoteka se ugrađuje u jedinstveno stablo UNIX sistema datoteka kao podstablo sa korenom u tački montiranja. Sistem datoteka se može kasnije i *demontirati*, tj. ukloniti iz jedinstvenog stabla, komandom `umount`:

```
umount /mnt/tacka
```

Napomenimo da je za montiranje i demontiranje obično neophodno biti administrator sistema, osim ako nije drugačije podešeno. Za pregled svih trenutno montiranih sistema datoteka, moguće je pokrenuti komandu:

```
mount
```

Napomenimo da se sistemi datoteka koji postoje na ugrađenim diskovima montiraju automatski prilikom podizanja sistema (ovo se postiže odgovarajućom konfiguracijom sistema). Sa druge strane, sistemi datoteka na prenosivim uređajima poput CD/DVD diskova ili fleš memorija se montiraju kada se priključe (ranije se to radilo ručno, `mount` komandom, dok se u moderno vreme to obavlja prilično automatski, zahvaljujući grafičkom korisničkom interfejsu).

5.3.3.1 Tipovi fajlova na sistemu UNIX

UNIX sistem razlikuje nekoliko tipova fajlova. Osnovni tip fajlova su *regularni fajlovi*, koji predstavljaju sve ono što uobičajeno podrazumevamo pod fajlom, poput tekstualnih fajlova, izvršnih fajlova, bibloteka, slika, video i audio zapisa i sl. Za razliku od nekih drugih operativnih sistema, UNIX ne pravi razliku među regularnim fajlovima na osnovu ekstenzije u nazivu fajla, već ostavlja programu koji koristi fajl da utvrdi konkretan tip fajla na osnovu sadržaja.[7]

[7] Ekstenzije u nazivima mogu postojati, ali one su tu pre svega da korisniku omoguće da lakše raspoznaje fajlove po tipovima. Takođe, grafički korisnički interfejs može fajlovima na osnovu ekstenzije pridruživati aplikaciju koja će automatski otvarati fajl kada se klikne na njega.

Elektronska verzija (2024)

Drugi tip fajlova koji UNIX razlikuje jesu *direktorijumi*. UNIX direktorijume vidi kao fajlove čiji je sadržaj spisak imena fajlova i direktorijuma koji se u njemu nalaze, uz reference na strukture podataka (tzv. *i-čvorove*, engl. *i-node*) u sistemu datoteka koji sadrže bliže informacije o tim fajlovima i direktorijumima. Svaki direktorijum uvek sadrži dve posebne stavke u sebi: jedna koja ukazuje na njega samog (označena sa `.`) i jedna koja ukazuje na roditeljski direktorijum (označena sa `..`). Ovo nam omogućava da koristimo putanje poput `./fajl` ili `../../fajl`.

Posebni tipovi fajlova se koriste za predstavljanje uređaja u okviru računarskog sistema. U pitanju su fajlovi koji nemaju sadržaj, već se pisanjem u te fajlove i čitanjem iz njih direktno komunicira sa drajverom za odgovarajući ulazno-izlazni uređaj. Ovi fajlovi se tradicionalno nalaze u direktorijumu `/dev/`, a kreira ih jezgro operativnog sistema automatski, prilikom inicijalizacije ulazno-izlaznih uređaja. Na primer, fajlovi poput `/dev/sda`, `/dev/sdb` predstavljaju hard (ili SSD) diskove ili druge memorijske uređaje poput fleš memorija. Ukoliko odgovarajući uređaj na sebi ima formirane particije, za njih se kreiraju posebni fajlovi `/dev/sda1`, `/dev/sda2` i sl. Fajlovi poput `/dev/cdrom`, `/dev/dvd` mogu ukazivati na uređaje za čitanje CD ili DVD diskova. Fajl `/dev/mouse` obično ukazuje na miša.

Simbolički linkovi (engl. *symbolic link*) predstavljaju poseban tip fajlova koji predstavljaju reference na druge fajlove u okviru sistema datoteka.

Pored navedenih tipova fajlova, UNIX raspoznaje još i posebne tipove fajlova koji se koriste u interprocesnoj komunikaciji, tj. pisanjem u ove fajlove i čitanjem iz njih vrši se komunikacija sa procesima koji su te fajlove kreirali. U ove tipove fajlova spadaju FIFO fajlovi i UNIX priključci (engl *UNIX socket*).

Kao što smo videli, pored regularnih fajlova, direktorijumi, uređaji kao i kanali interprocesne komunikacije se predstavljaju fajlovima. Ovo je jedna od najvažnijih karakteristika UNIX sistema, a ogleda se u krilatici „Sve je fajl" (engl. *Everything is a file*). Ovo znači da se u okviru UNIX sistema većina resursa predstavlja fajlom, što omogućava unifikovani pristup tip resursima.

5.3.3.2 Struktura UNIX sistema datoteka

Prilikom instalacije UNIX sistema, u okviru korenog direktorijuma kreiraju se neki standardni direktorijumi koji imaju svoju unapred određenu ulogu u sistemu. Neki od njih su:

- `/bin`, `/sbin` sadrže osnovne UNIX programe (izvršne fajlove) i po pravilu se nalaze u putanji ljuske
- `/etc` sadrži konfiguracione fajlove, obično u tekstualnom formatu
- `/dev` sadrži fajlove koji predstavljaju interfejs ka uređajima
- `/boot` sadrži fajlove jezgra sistema i punioca koji se koriste prilikom uključivanja računara i podizanja sistema
- `/lib`, `/lib64` sadrže osnovne sistemske biblioteke
- `/home` sadrži korisničke direktorijume
- `/usr` sadrži instalirani korisnički softver
- `/tmp` sadrži privremene fajlove koje procesi kreiraju

Ostavljamo čitaocu da dalje istraži ove direktorijume i bolje se upozna sa njihovim sadržajem.

5.3.3.3 Prava pristupa

Da bi rad sa sistemom datoteka bio bezbedan, neophodno je ograničiti prava pristupa korisnicima kada su u pitanju fajlovi i direktorijumi. Kao što je ranije rečeno, `root` korisnik (tačnije, procesi u njegovom vlasništvu) ima neograničena prava i može pristupati svim fajlovima i direktorijumima i vršiti sve moguće radnje nad njima. Ostalim korisnicima se nameću restrikcije prema ulozi koju oni imaju kada je u pitanju konkretan fajl ili direktorijum. Svaki fajl ili direktorijum ima svog *vlasnika* kao i *grupu vlasnika*. Kada se fajl ili direktorijum kreira, njegov vlasnik i grupa vlasnik će inicijalno biti redom vlasnik i grupa vlasnik procesa koji ga je kreirao. To znači da će svi fajlovi ili direktorijumi koje korisnik kreira podsredstvom korisničkog interfejsa biti inicijalno u njegovom vlasništvu. Vlasnik fajla se naknadno može promeniti, ali samo od strane `root` korisnika. Drugim rečima, obični korisnici ne mogu da „otuđe" svoje fajlove i direktorijume (ovo je takođe bitno iz bezbednosnih razloga). Sa druge strane, običan korisnik može promeniti grupu vlasnika nekog svog fajla ili direktorijuma, ali pritom nova grupa vlasnik može biti samo neka od grupa kojoj korisnik pripada.

Promena vlasnika fajla (od strane `root` korisnika), obavlja se komandom `chown`:

Elektronska verzija (2024)

```
chown pera /var/file
```

Ovim vlasnik fajla `/var/file` postaje korisnik `pera`. Ako želimo da istovremeno promenimo i grupu vlasnika, možemo pokrenuti sledeću komandu:

```
chown pera:users /var/file
```

Sada je novi vlasnik fajla `/var/file` korisnik `pera`, a nova grupa vlasnik je `users`. Ako želimo da promenimo samo grupu vlasnika, možemo pokrenuti komandu:

```
chgrp users /var/file
```

Prilikom utvrđivanja da li neki proces (koji nije u vlasništvu `root` korisnika) ima prava da obavi određenu operaciju sa fajlom ili direktorijumom, operativni sistem razvrstava korisnike u tri kategorije. Prvu kategoriju čini vlasnik fajla ili direktorijuma, drugu čine svi korisnici koji pripadaju grupi koja je vlasnik fajla ili direktorijuma, a treću čine svi ostali korisnici. Za svaku od ove tri kategorije u sistemu datoteka se čuva informacija o pravima koje korisnici iz tih kategorija imaju nad tim fajlom ili direktorijumom. Ta prava mogu biti:

- pravo čitanja (označeno sa `r`): u slučaju fajla, to znači da možemo otvoriti fajl za čitanje i samim tim videti njegov sadržaj; za direktorijum ovo pravo omogućava izlistavanje sadržaja direktorijuma.
- pravo pisanja (označeno sa `w`): u slučaju fajla, ovo pravo omogućava procesu da otvori fajl za pisanje i, samim tim, da modifikuje sadržaj fajla. Za direktorijum, ovo pravo omogućava kreiranje novih ili brisanje postojećih fajlova ili direktorijuma u okviru tog direktorijuma.
- pravo izvršavanja (označeno sa `x`): u slučaju fajla, ovo pravo označava da korisnik može pokretati taj fajl kao izvršni program; u slučaju direktorijuma, ovo pravo označava da proces može „ući” u taj direktorijum, tj. postaviti ga za svoj tekući direktorijum ili pristupati fajlovima i direktorijumima u okviru tog direktorijuma.

Napomenimo da ovo poslednje znači da ako želimo da pristupimo nekom fajlu pomoću putanje (apsolutne ili relativne), tada za svaki od direktorijuma koji se pominju u toj putanji moramo imati pravo `x`, a za sam fajl kome pristupamo moramo imati ono pravo koje odgovara operaciji koju želimo da izvršimo (`r` za čitanje, `w` za modifikaciju, `x` za pokretanje).

Prava pristupa fajlu obično prikazujemo u obliku niske koja se sastoji iz devet karaktera: prva tri označavaju prava pristupa vlasnika (redom `r`, `w` i `x`), druga tri označavaju prava pristupa pripadnika grupa, a treća tri označavaju prava pristupa ostalih korisnika. Na primer, niska:

```
rwxr-xr-x
```

označava da vlasnik ima sva tri prava, dok pripadnici grupa i ostali korisnici imaju prava `r` i `x`, dok nemaju pravo `w` (ovo je označeno simbolom `-` umesto `w`).

Prilikom utvrđivanja prava pristupa, operativni sistem prvo utvrđuje da li je vlasnik procesa `root` korisnik. Ako jeste, dodeljuje mu pravo pristupa. U suprotnom, ako je vlasnik procesa vlasnik fajla, prava pristupa određena su pravima koja ima vlasnik. U suprotnom, ako se grupa vlasnik procesa ili neka od suplementarnih grupa procesa poklapa sa grupom vlasnikom fajla, tada su prava pristupa određena pravima koja imaju pripadnici grupe vlasnika fajla. U suprotnom, prava su određena pravima koja nad tim fajlom imaju ostali korisnici.

Prava pristupa se mogu menjati komandom `chmod`:

```
chmod u+x,go-rwx /var/file
```

Ovom komandom se vlasniku fajla `/var/file` (označenom sa `u`) dodaje (`+`) pravo `x`, dok se pripadnicima grupe (`g`) i ostalim korisnicima (`o`) oduzimaju (`-`) sva prava (`r`, `w` i `x`). Promenu prava pristupa može vršiti samo vlasnik fajla (ili `root` korisnik).

5.3.3.4 Navigacija kroz sistem datoteka

Već od ranije znamo da u svakom trenutku možemo videti u kom se direktorijumu trenutno nalazimo pozivanjem komande `pwd`. Da bismo izlistali sadržaj tekućeg direktorijuma (pod pretpostavkom da na to imamo pravo, tj. da imamo pravo `r` nad tim direktorijumom), koristimo komandu:

```
ls
```

U slučaju da želimo da izlistamo neki drugi direktorijum, a ne tekući, možemo navesti putanju do direktorijuma kao argument komandne linije:

```
ls /usr/
```

Izlaz ove komande bi mogao da izgleda ovako:

```
bin/  etc/  games/  include/  info/  lib/  lib64/  man/  sbin/  share/  src/
```

Ukoliko želimo da imamo bogatiji ispis koji uključuje neke korisne informacije o fajlovima, poput prava pristupa, veličine fajla, datuma poslednje modifikacije, vlasnika i grupe vlasnika i sl., možemo zadati opciju `-l`:

```
ls -l /var/www/htdocs/
```

Izlaz ove komande može izgledati ovako:

```
drwxr-xr-x  2 root root 4,0K Feb 13  2021 htdig/
-rw-r--r--  1 root root   45 Jun 11  2007 index.html
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.13676
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.19999
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.23618
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.27667
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.32338
drwxr-xr-x 14 root root 4,0K Oct 19  2023 manual/
```

Svaka linija ovog ispisa daje informacije o jednom od fajlova ili direktorijuma koji se nalaze u okviru izlistanog direktorijuma. Ukoliko je u pitanju direktorijum, tada linija počinje simbolom `d`, u suprotnom počinje simbolom `-`. Nakon toga slede informacije o pravima pristupa, vlasniku, grupi vlasniku, veličini fajla i datumu i vremenu poslednje modifikacije. Na kraju linije navedeno je samo ime fajla (u slučaju direktorijuma, završava se simbolom `/`).

Pod UNIX-om postoji i koncept *skrivenih fajlova*. U pitanju su fajlovi koji se podrazumevano ne prikazuju prilikom izlistavanja direktorijuma. Ovi fajlovi se prepoznaju tako što njihov naziv počinje tačkom. Na primer, ako kreiramo fajl `.moj_fajl`, ovaj fajl će biti skriven, tj. neće se prikazivati na izlazu komande `ls`. Ako ipak želimo da nam se prikažuju i skriveni fajlovi, možemo zadati opciju `-a`:

```
ls -l -a /var/www/htdocs/
```

Izlaz ove komande bi mogao izgledati ovako:

```
drwxr-xr-x  4 root root 4,0K Sep  7 13:46 ./
drwxr-xr-x  7 root root 4,0K Oct 19  2023 ../
-rw-r--r--  1 root root    0 Sep  7 13:46 .hidden.html
drwxr-xr-x  2 root root 4,0K Feb 13  2021 htdig/
-rw-r--r--  1 root root   45 Jun 11  2007 index.html
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.13676
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.19999
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.23618
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.27667
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.32338
drwxr-xr-x 14 root root 4,0K Oct 19  2023 manual/
```

Ono što primećujemo je fajl `.hidden.html` koji ranije nije prikazivan, jer je skriven. Takođe, primećujemo još dva direktorijuma: `./` i `../`. Kao što od ranije znamo, `.` i `..` označavaju redom tekući direktorijum i prethodni (roditeljski) direktorijum. Dakle, svaki direktorijum sadrži referencu na samog sebe, kao i na svog roditelja. Kako ove reference počinju tačkom, skrivene su i neće biti podrazumevano prikazivane pri pozivu komande `ls`.

Elektronska verzija (2024)

5.3.3.5 Kreiranje i brisanje fajlova i direktorijuma

Da bismo kreirali novi direktorijum, potrebno je pokrenuti komandu `mkdir`:

```
mkdir novi_dir
```

Ovim se kreira direktorijum `novi_dir` u tekućem direktorijumu. Ako želimo da kreiramo novi direktorijum u nekom drugom direktorijumu, a ne u tekućem, dovoljno je navesti putanju (apsolutnu ili relativnu) do novog direktorijuma:

```
mkdir /home/pera/novi_dir
```

Ovim će biti kreiran novi direktorijum `novi_dir` u okviru direktorijuma `/home/pera`.

Fajlovi se unutar direktorijuma mogu kreirati podsredstvom programa koje pokrećemo. Na primer, ako pokrenemo editor teksta:

```
emacs novi_fajl.txt
```

otvoriće nam se editor teksta `emacs` u kome ćemo moći da uređujemo fajl. Nakon što sačuvamo fajl i izađemo iz editora, fajl `novi_fajl.txt` biće kreiran u tekućem direktorijumu.

Ako želimo da kreiramo prazan fajl, to možemo najjednostavnije uraditi pomoću komande `touch`:

```
touch novi_fajl.txt
```

Da bismo obrisali fajl, možemo pokrenuti komandu `rm`:

```
rm novi_fajl.txt
```

Sa druge strane, za brisanje direktorijuma potrebno je pokrenuti komandu:

```
rmdir /home/pera/novi_dir
```

Da bi ova komanda uspela, potrebno je da direktorijum koji brišemo bude prazan. U slučaju da nije prazan, možemo najpre obrisati njegov sadržaj, pa onda obrisati i njega. Alternativno, možemo zahtevati rekurzivno brisanje svih fajlova i poddirektorijuma iz datog direktorijuma:

```
rm -R /home/pera/novi_dir
```

Opcija `-R` zahteva od komande `rm` da rekurzivno obriše sve fajlove i direktorijume iz direktorijuma `novi_dir`, nakon čega će biti uklonjen i on sâm.

Podsetimo se da je za kreiranje i brisanje fajlova i direktorijuma unutar nekog direktorijuma *dir* potrebno imati pravo `w` nad tim direktorijumom *dir*. Sa druge strane, nije neophodno imati pravo `w` nad samim fajlom ili direktorijumom koji brišemo. Ipak, `rm` program može prijaviti upozorenje u slučaju brisanja fajla za koji nemamo pravo modifikacije. Da bismo izbegli to upozorenje, dovoljno je zadati opciju `-f`:

```
rm -f -R /home/pera/novi_dir/
```

Ovim će biti obrisan kompletan sadržaj direktorijuma `novi_dir`, kao i sam taj direktorijum, čak i da u njemu postoje fajlovi za koje nemamo pravo modifikacije (ili čak fajlovi u tuđem vlasništvu).

5.3.3.6 Kopiranje i premeštanje fajlova i direktorijuma

Fajl se može jednostavno kopirati komandom `cp`:

```
cp fajl.txt kopija.txt
```

Ovim se u tekućem direktorijumu kreira fajl `kopija.txt` sa identičnim sadržajem kao i `fajl.txt`. Kopija se ne mora kreirati u tekućem direktorijumu, već se može navesti putanja do bilo koje druge lokacije:

```
cp fajl.txt /home/pera/kopija.txt
```

Naravno, potrebno je da imamo pravo `w` nad direktorijumom u kom kreiramo kopiju, kao i pravo `x` za sve direktorijume koji se pominju u putanji. Kao drugi argument komande `cp` možemo navesti i samo putanju do direktorijuma u kom treba napraviti kopiju, bez navođenja naziva kopije:

Elektronska verzija (2024)

```
cp fajl.txt /home/pera/
```

U tom slučaju će u datom direktorijumu biti kreirana kopija sa istim imenom kao i originalni fajl. Ako fajl sa tim imenom već postoji, biće prebrisan, tj. njegov sadržaj će biti zamenjen sadržajem fajla koji se kopira (za ovo je neophodno dodatno imati i pravo `w` nad odredišnim fajlom).

Ako želimo da kopiramo ceo direktorijum, neophodno je da se rekurzivno kopiraju svi njegovi poddirektorijumi i fajlovi. Ovo se može postići opcijom `-R`:

```
cp -R novi_dir/ /home/pera/
```

Ovom komandom se u okviru direktorijuma `/home/pera/` kreira kopija direktorijuma `novi_dir/` sa njegovim celokupnim sadržajem.

Ukoliko želimo da premestimo fajl na drugu lokaciju, umesto da kreiramo kopiju, koristićemo komandu `mv`:

```
mv fajl.txt /home/pera/
```

Ovom komandom se fajl `fajl.txt` premešta u direktorijum `/home/pera/`. Za ovu operaciju, potrebno je imati pravo `w` i nad direktorijumu u kome se fajl trenutno nalazi (kod nas je to tekući direktorijum) i nad direktorijumom u koje se vrši premeštanje (`/home/pera/`), s obzirom da se sadržaj oba direktorijuma menja. Slično, moguće je premeštati čitave direktorijume:

```
mv /home/pera/novi_dir/  .
```

Ovom komandom vrši se premeštanje direktorijuma `novi_dir` iz direktorijuma `/home/pera` u tekući direktorijum (označen sa `.`, ko i obično).

U prethodnim primerima, kao odredište smo navodili putanju do direktorijuma u koji se fajl ili direktorijum premešta. Pritom, fajl ili direktorijum koji se premešta će na svojoj novoj destinaciji imati isto ime koje je imao i do sada. Ako želimo da mu prilikom premeštanja promenimo ime, tj. da se na novom odredištu zove drugačije, dovoljno je samo u nastavku odredišne putanje navesti i novo ime:

```
mv fajl.txt /home/pera/novi_fajl.txt
```

Sada će fajl biti premešten u direktorijum `/home/pera/`, ali će tamo biti sačuvan pod novim imenom `novi_fajl.txt`. Ova mogućnost komande `mv` se koristi i kada želimo samo da preimenujemo fajl, bez premeštanja:

```
mv fajl.txt novi_fajl.txt
```

Ovim se fajl „premešta" iz tekućeg direktorijuma u tekući direktorijum, tj. ne premešta se uopšte. Jedini efekat ove komande je preimenovanje fajla iz `fajl.txt` u `novi_fajl.txt`.

Napomenimo da je premeštanje fajla daleko efikasnija operacija od kopiranja. Naime, prilikom kopiranja, neophodno je kreirati novi fajl i u njega kopirati celokupan sadržaj originalnog fajla. Ovo može zahtevati vreme koje je proporcionalno veličini fajla koji se kopira. Naročito drastičan slučaj može biti kopiranje celih direktorijuma, što podrazumeva kopiranje svih fajlova i direktorijuma koji se u njemu nalaze. Sa druge strane, premeštanje fajla ne podrazumeva fizičko premeštanje njegovog sadržaja na disku, već samo premeštanje reference na fajl iz jednog direktorijuma u drugi. Ova operacija se obavlja vrlo efikasno, bez obzira na veličinu fajla ili direktorijuma koji se premešta.

Ponekad je potrebno obezbediti da referenca na isti fajl ili direktorijum bude dostupna na više različitih lokacija. Da bi se ovo postiglo, pod UNIX-om možemo koristiti *simboličke linkove* (engl. *symbolic link*). Simbolički link predstavlja specijalnu vrstu fajla čija je jedina uloga da referiše na drugi postojeći fajl. Možemo ga kreirati komandom `ln`:

```
ln -s /home/pera/fajl.txt lfajl.txt
```

Ovom komandom se u tekućem direktorijumu kreira fajl `lfajl.txt` koji predstavlja simbolički link koji ukazuje na fajl `/home/pera/fajl.txt`. U nastavku, svaka operacija nad simboličkim linkom (izuzev brisanja i premeštanja) se ne odnosi na link, već na fajl na koji link ukazuje. Na primer:

```
cp lfajl.txt /tmp/kopija.txt
```

Elektronska verzija (2024)

Ovim se sadržaj fajla /home/pera/fajl.txt kopira u fajl /tmp/kopija.txt. Simbolički link zauzima veoma malo prostora na disku (taman koliko je dovoljno da se sačuva putanja do fajla na koji se ukazuje). Može ukazivati na fajl ili direktorijum u bilo kom sistemu datoteka koji je montiran. Simbolički linkovi sami po sebi nemaju prava pristupa, već se prilikom utvrđivanja prava pristupa koriste prava fajla na koji link ukazuje. Takođe, komanda chmod menja prava pristupa fajla na koji link ukazuje:

```
chmod go-rwx lfajl.txt
```

Ovom komandom se članovima grupe i ostalim korisnicima oduzimaju sva prava nad fajlom /home/pera/fajl.txt.

5.3.3.7 Pretraga fajlova

Često nam je potrebno da u okviru sistema datoteka pronađemo fajl ili fajlove sa određenim karakteristikama. Te karakteristike se mogu odnositi na naziv fajla, tip fajla, veličinu fajla, vreme poslednje modifikacije, ili na prava pristupa. Pod UNIX-om, u ovu svrhu se može koristiti komanda find:

```
find /home/pera/ -name '*.html'
```

Ova komanda će pronaći i izlistati sve fajlove u okviru direktorijuma /home/pera (i njegovih poddirektorijuma) čije ime (opcija -name) je oblika *.html, gde * označava bilo koji niz karaktera. Apostrofi oko obrasca *.html su neophodni, da bi se sprečila ekspanzija simbola * u okviru same ljuske. Slično, ako pozovemo komandu:

```
find /home/pera/ -size +5M
```

dobićemo spisak svih fajlova iz direktorijuma /home/pera čija je veličina 5Mb ili više (umesto simbola + mogao je stajati simbol -, što bi nam dalo sve fajlove veličine ne veće od 5Mb). Ako pozovemo komandu:

```
find /home/pera/ -perm -u=w,g=w
```

dobićemo spisak svih fajlova u direktorijumu /home/pera kod kojih i vlasnik (u) i članovi grupe vlasnika (g) imaju pravo w. Ako umesto toga pozovemo komandu:

```
find /home/pera/ -perm /u=w,g=w
```

dobićemo spisak svih fajlova kod kojih *ili* vlasnik *ili* članovi grupe vlasnika imaju pravo w. Dakle, prefiks - označava da sva navedena prava moraju da postoje, dok prefiks / označava da bar jedno od navedenih prava mora da postoji. Možemo vršiti pretragu i prema vremenu poslednje modifikacije. Na primer:

```
find /home/pera -mtime 0
```

Ova komanda će izlistati sve fajlove iz direktorijuma /home/pera koji su modifikovana u poslednja 24 časa. Naime, argument iza opcije -mtime predstavlja broj *celih* dana koji su protekli od poslednje modifikacije fajla (eventualni ostatak vremena se odbacuje). Vrednost 0 stoga označava da je prošlo nula celih dana, tj. manje od 24 sata. Vrednost +1 bi označavala bar jedan ceo dan, tj. više od 24 sata, dok bi vrednost -5 označavala najviše 5 celih dana. Ako želimo da pronađemo sve obične fajlove (tj. da izostavimo direktorijume), imamo sledeću komandu:

```
find /home/pera -type f
```

Tip f označava obične fajlove, dok tip d označava direktorijume. Tip l označava simboličke linkove. Možemo pretraživati sve fajlove koji su u vlašnistvu nekog korisnika:

```
find /home/pera -user mika
```

Ova komanda izlistava sve korisnike u direktorijumu /home/pera čiji je vlasnik mika. Moguće je navoditi i više različitih uslova:

```
find /home/pera -name '*.html' -user mika -perm -u=rw
```

Ova komanda pronalazi sve fajlove u direktorijumu /home/pera sa imenom oblika *.html, u vlasništvu korisnika mika, pri čemu vlasnik poseduje prava r i w. Slično, komanda:

```
find /home/pera -name '*.html' '!' -user pera
```

Elektronska verzija (2024)

Pronalazi sve fajlove u direktorijumu `/home/pera` sa imenom oblika `*.html` koji *nisu* u vlasništvu korisnika `pera`. Negacija uslova postiže se navođenjem operatora `!` ispred odgovarajućeg uslova (apostrofi oko `!` su neophodni da bi se izbeglo specijalno značenje koje operator `!` ima u ljusci).

Komanda `find` podrazumevano izlistava pronađene fajlove. Ako želimo da se nad pronađenim fajlovima izvrši neka druga akcija, možemo je navesti u nastavku komande:

```
find . -name '*.html' -exec cat '{}'
```

Ovom komandom se na sve pronađene fajlove redom primenjuje komanda `cat`. Ova komanda prosto izlistava *sadržaj* fajla na standardnom izlazu. Oznaka `{}` označava putanju do pronađenog fajla (apostrofi su ponovo neophodni, kako bi se izbeglo specijalno značenje koje simboli `{` i `}` imaju u ljusci). Slično komanda:

```
find . -name '*~' -exec rm '{}'
```

briše sve fajlove iz tekućeg direktorijuma čije se ime završava sa `~`. Ekvivalentni efekat bi imala komanda:

```
find . -name '*~' -delete
```

5.3.4 Upravljanje procesima

U prethodnim odeljcima videli smo na koji načini se mogu pokretati procesi u okviru komandne linije. U ovom odeljku detaljnije razmatramo upravljanje procesima.

5.3.4.1 Prednji i pozadinski procesi

Za proces pokrenut u komandnoj liniji (ili grupu procesa, u slučaju nadovezivanja programa) kažemo da je *prednji* (engl. *foreground*), ako su mu standardni ulaz i izlaz povezani sa tastaturom i monitorom. Ovo je podrazumevano ponašanje prilikom pokretanja procesa u komandnoj liniji – komandni interfejs predaje pokrenutom procesu kontrolu nad tastaturom i monitorom i nadalje korisnik komunicira sa tim procesom umesto sa ljuskom. Kada se proces završi, kontrola nad tastaturom i monitorom se vraća ljusci, čime je omogućeno pokretanje novih komandi.

Sa druge strane, proces koji je pokrenut može da ne želi da komunicira sa korisnikom putem tastature i monitora. Postoje procesi koji se pokreću da bi obavili neki drugi zadatak koji ne podrazumeva komunikaciju sa korisnikom. Ako taj proces, pritom, radi dugo, korisniku će ljuska biti blokirana dok se proces ne završi. Umesto toga, postoji mogućnost da se proces pokrene *u pozadini*. Ovakve procese nazivamo *pozadinski procesi* (engl. *background*). Sintaksa pozivanja procesa u pozadini podrazumeva dodavanje simbola `&` na kraj komande:

```
/usr/sbin/sshd &
```

Ovom komandom se pokreće program `/usr/sbin/sshd` u pozadini. To znači da ljuska neće tom procesu predati kontrolu nad tastaturom i monitorom, već će tu kontrolu zadržati za sebe, što će omogućiti pokretanje drugih komandi dok prethodno pokrenuti proces radi u pozadini.

Mehanizam pokretanja procesa u pozadini se obično koristi za pokretanje procesa koji rade jako dugo, bez ikakve interakcije sa korisnikom. Ovakve procese nazivamo *demonski procesi* (engl. *deamon process*). Ovi procesi, iako mogu biti pokrenuti iz ljuske, nisu vezani za ljusku i mogu nastaviti svoj rad čak i kada se ljuska isključi, pa čak i kada se korisnik odjavi sa sistema. Najčešća upotreba demonskih procesa je za implementaciju različitih *servera* – procesa čija je uloga da pružaju različite usluge klijentskim programima (poput servera štampe, veb servera i sl.). Serveri su obično aktivni tokom čitavog rada računara i operativnog sistema.

Druga upotreba pokretanja procesa u pozadini je na sistemima sa grafičkim korisničkim interfejsom, kada iz ljuske (pokrenute u okviru emulatora terminala) želimo da pokrenemo neku aplikaciju sa grafičkim korisničkim interfejsom (poput veb pregledača, programa za tabelarna izračunavanja i sl.). Ovi programi se mogu pokrenuti iz ljuske, ali oni neće koristiti standardni ulaz i izlaz, već će kreirati poseban prozor na ekranu i komuniciraće sa korisnikom putem svog grafičkog interfejsa. Sa druge strane, ljuska će biti bespotrebno blokirana i neće biti moguće pokretati druge programe. Da bi se to sprečilo, možemo pokrenuti aplikaciju u pozadini:

```
firefox &
```

Aplikacija pokrenuta u pozadini se može vratiti u status prednjeg procesa komandom:

```
fg
```

Elektronska verzija (2024)

Ukoliko u nekom trenutku imamo više procesa pokrenutih u pozadini, tada svaki od njih ima dodeljen broj (ovaj broj se ispisuje nakon pokretanja procesa u pozadini). Ako neki od tih procesa želimo da prebacimo u status prednjeg procesa, tada komandi `fg` moramo navesti broj tog pozadinskog procesa:

```
fg 1
```

Takođe, proces koji je pokrenut u pozadini može i sam preći u status prednjeg procesa, ako pokuša da vrši ispis na standardni izlaz.

Sa druge strane, ako je neki proces greškom pokrenut kao prednji proces, možemo ga naknadno prebaciti u status pozadinskog procesa. To radimo tako što najpre na tastaturi pritisnemo kombinaciju tastera `Ctrl-Z`, a zatim u komandnoj liniji zadamo komandu:

```
bg
```

5.3.4.2 Pregled pokrenutih procesa

Pregled pokrenutih procesa može se obaviti komandom:

```
ps
```

Ova komanda izlistava sve aktivne procese pokrenute u okviru ljuske iz koje se komanda pokreće. Ovo obično uključuje samu ljusku kao i proces `ps`. Uz to, prikazuju se i pozadinski procesi pokrenuti iz te ljuske. Ukoliko želimo da prikažemo sve procese u vlasništvu nekog određenog korisnika, možemo pokrenuti komandu:

```
ps -u pera
```

Podrazumevano, za svaki od procesa prikazuje se PID, terminal u okviru koga je pokrenut, vreme izvršavanja procesa, kao i naziv programa. Po želji, različitim opcijama se mogu zahtevati i dodatne informacije o procesima. Na primer:

```
ps -u pera -o user,pid,ppid,state,time,command
```

Opcijom `-o` i argumentima koji (razdvojeni zarezima) slede iza nje zadaju se parametri procesa koje želimo da komanda ispiše. U našem primeru, želimo da za svaki proces ispišemo njegovog vlasnika (`user`), njegov PID (`pid`), kao i PID njegovog roditelja (`ppid`), zatim stanje procesa (`state`), vreme izvršavanja (`time`) i punu komandu (`command`) kojom je proces pokrenut, sa sve putanjom do programa i argumentima komandne linije. Stanje procesa se označava slovima poput `R` (*radi* ili *spreman*), `S` (*čeka*), i sl. Neki uobičajeni ispis koji uključuje najznačajnije parametre procesa (poput ovih koje smo u prethodnoj komandi eksplicitno navodili) se može postići i zadavanjem opcije `-F`:

```
ps -u pera -F
```

Ako želimo da prikažemo sve procese svih korisnika, možemo zadati komandu:

```
ps -e -F
```

Pored programa `ps`, postoji i program `top` koji omogućava interaktivni i dinamički prikaz procesa koji se izvršavaju na sistemu. Program se pokreće jednostavno:

```
top
```

Nakon pokretanja ovog programa, u okviru ljuske se prikazuje tabela sa procesima i njihovim najznačajnijim parametrima. Ta tabela se podrazumevano osvežava na svake tri sekunde, tako da se mogu kontinuirano pratiti resursi koje procesi zauzimaju (poput fizičke i virtuelne memorije i procesorskog vremena). Takođe, u gornjem delu ekrana prikazuje se ukupno zauzeće resursa od strane svih procesa.

Elektronska verzija (2024)

5.3.4.3 Prekidanje rada procesa

Ako želimo da neki proces nasilno prekinemo, možemo mu iz ljuske poslati signal za prekid. Ovo se radi komandom `kill`:

```
kill 2456
```

Ovim se procesu sa PID-om 2456 šalje signal `SIGTERM` za koji je podrazumevana akcija prekid rada procesa. Pojedini procesi mogu ignorisati ovaj signal ili biti programirani da obavljaju neku drugu akciju prilikom pristizanja tog signala. Alternativno, možemo pokrenuti komandu:

```
kill -s SIGKILL 2456
```

Ovim se procesu sa PID-om 2456 šalje signal `SIGKILL` koji se ne može ignorisati, niti je moguće redefinisati njegovo značenje, tako da garantovano ubija proces.

PID procesa koji želimo da likvidiramo uvek možemo da vidimo na izlazu komande `ps`. Ipak, ako nam je zgodnije da navedemo ime programa koji želimo da likvidiramo, možemo pokrenuti komandu:

```
killall -s SIGKILL firefox
```

Da bismo mogli da pošaljemo signal procesu, taj proces mora biti u vlasniku korisnika koji pokreće `kill` komandu, osim u slučaju `root` korisnika koji ima prava da pošalje signal bilo kom procesu.

Elektronska verzija (2024)

Glava 6

Računarske mreže

Ljudi su od svog nastanka imali potrebu da komuniciraju između sebe, deleći na taj način korisne informacije. U početku, razmena informacija je bila direktna i neposredna, dok su kasniji tehnološki napredak i rastuća potreba čoveka za bržom i komunikacijom sa većih udaljenosti rezultirali razvojem novih sistema komunikacije kakve danas poznajemo. U takve sisteme ubrajamo telegrafiju, telefoniju, emitovanje televizijskog i radio signala, i druge. U određenim istorijskim razdobljima oni su igrali ključne uloge u razvoju čovečanstva. Iako i dalje u upotrebi, u novije vreme sve više bivaju zamenjeni tehnološki naprednijim vidovima komunikacije koji nude veće brzine, dostupnost, pouzdanost i bezbednost prenesenih informacija.

Danas je skoro nemoguće zamisliti računar koji nije povezan sa drugim računarima. Većina digitalnih uređaja koje svakodnevno koristimo (telefoni, tableti, televizori, satovi, ...) je takođe umrežena između sebe i povezana sa drugim računarskim sistemima, omogućavajući nam brz pristup korisnim informacijama. Ovakvo međusobno povezivanje računara i drugih uređaja na globalnom nivou, razvoj interneta i servisa koje on nudi (veb, elektronska pošta, video pozivi, društvene mreže, ...) omogućili su nove, do tada nezamislive primene računara, eksponencijalno su povećali broj korisnika ovih uređaja i umnogome im olakšali svakodnevni život. Promene koje je donela ova informaciono-komunikaciona tehnologija su nesagledive u svakoj sferi života, pa se često za njih koristi termin *Četvrta industrijska revolucija*.

U 21. veku, računarske mreže čine okosnicu svih mogućih vidova komunikacije, povezujući ljude, ljude i računarske sisteme, ali i računarske sisteme između sebe. Mreže omogućavaju različitim uređajima (računari, pametni telefoni i satovi, pametni kućni aparati, automobili, ...) da se povežu, dele informacije i podatke i međusobno sarađuju. Računarska mreža se definiše kao *kolekcija međusobno povezanih hardverskih uređaja* (najmanje dva urađaja) koji mogu da *komuniciraju jedni sa drugima* radi *razmene podataka i resursa*. Oni su povezani *komunikacionom opremom*, kao što su kablovi, habovi (eng. *hub*), svičevi (eng. *switch*) i ruteri (eng. *router*) preko koje se, uz pomoć *komunikacionog softvera*, obavlja razmena *digitalnih poruka*. Dakle, uređaji komuniciraju međusobno tako što jedni drugima šalju bitove, nizove nula i jedinica kojima su podaci kodirani.

Vremenom, broj računara koji su međusobno umreženi kontantno je rastao došavši do trenutnog nivoa kada su skoro svi računari na svetu povezani. Takva globalna mreža, mreža svih mreža, naziva se *Internet*. Često se greškom Internet poistovećuje sa računarskim mrežama, koje ipak predstavljaju širi pojam. Nije teško zamisliti određeni broj računara u jednoj učionici međusobno povezanih, koji nemaju vezu sa spoljašnjim svetom, tj. nizu povezani na Internet.

Računarske mreže su drastično transformisale način na koji komuniciramo i delimo informacije. Razumevanje njihove strukture i načina na koji funkcionišu je od ogromnog značaja za efikasno korišćenje njihovog punog potencijala. U nastavku ovog poglavlja, bavićemo se ulogom i scenarijima upotrebe računarskih mreža, komponentama, tipovima mreža, protokolima računarskih mreža, kao i bezbednošću u računarskim mrežama.

6.1 Uloga računarskih mreža

Računarske mreže imaju primene u različitim oblastima, kako u poslovnom svetu, tako i za kućne potrebe. U kompanijama, računarske mreže se tradicionalno koriste za *deljenje resursa, podataka, komunikaciju zaposlenih i njihovu saradnju*. U kućnim scenarijima, računarske mreže se koriste za *pristup informacijama koje su dostupne na Vebu, čitanje novina, časopisa i elektronskih knjiga, dopisivanje, pristup društvenim mrežama i servisima koji nude audio i video sadržaj na zahtev, elektronsku kupovinu, gledanje televizijskih sadržaja,* i slično. U poslednjih par godina, evidentan je razvoj tzv. *Interneta stvari* (eng. *Intenet of Things*) koji predstavlja umrežavanje raznih tipova objekata (vozila, zgrada, stanova, brava, malih i velikih kućnih aparata, senzora, prekidača, i slično) koji u sebi imaju elektronske čipove i mogu da komuniciraju međusobno obezbeđujući jedni drugima različite tipove informacija i usluga.

Elektronska verzija (2024)

Osnovne uloge računarskih mreža mogu se podeliti u sledeće četiri kategorije:

Komunikacija: Jedna od glavnih uloga računarskih mreža je komunikacija. Korišćenjem povezenih računara ljudi mogu da se dopisuju elektronskom poštom i instant porukama, ćaskaju i održavaju video i audio konferencije. U današnje vreme postoje alati i aplikacije koji omogućavaju da veći broj ljudi iako udaljeni sarađuje međusobno i rade na istim zadacima posredstvom mreže.

Deljenje informacija i podataka: Korisnici računarskih mreža mogu pristupati ogromnim količinama informacija koje se nalaze na udaljenim lokacijama, tj. na drugim računarima u okviru iste mreže. Informacije se najčešće prenose preuzimanjem različitih vrsta datoteka, na primer pristupanjem određenim Veb stranicama na Internetu, koje predstavljaju jedan od glavnih izvora informacija.

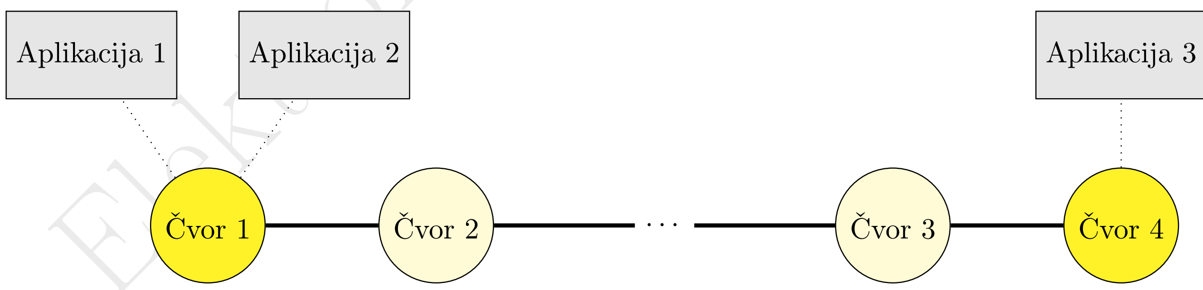
Deljenje softverskih resursa: Osim informacija i dokumenata, računarske mreže mogu služiti za deljenje softverskih resursa. Softveru instaliranom na jednom računaru u okviru jedne mreže mogu pristupati svi korisnici te mreže kojima je to dozvoljeno i rešiti svoje probleme korišćenjem servisa koje taj softver nudi.

Deljenje hardverskih resursa: Korisnici računarskih mreža mogu deliti i zajedno koristiti različite uređaje kada to ima smisla. Na primer, zaposleni u jednoj kompaniji mogu imati jedan štampač koji koristi svima, i svako mu sa svog računara može pristupiti i poslati dokument na štampu.

Deljenje resursa, kako softverskih tako i hardverskih u okviru računarskih mreža ima višestruke koristi. Povećava se efikasnost i iskorišćenost resursa, a u isto vreme *smanjuju se troškovi.* Na primer, nema potrebe instalirati isti softver na svakom pojedinačnom računaru, ako je moguće pristupiti mu preko mreže. Računari na kojima se izvršava takav softver često su boljih performansi od ostalih, pa je vreme izvršavanja na njima kraće. Održavanje jedne instance tog softvera je jednostavnije i jeftinije od pojedinačnog održavanja na svim računarima. Takođe, deljeni štampači u mreži, iako boljih performansi, mogu biti jeftiniji od ukupne cene više pojedinačnih koji bi opsluživali svakog pojedinca u mreži. Cena održavanja i cena štampe na takvim uređajima je po pravilu manja.

6.2 Komponente računarskih mreža

Pojednostavljeno, računarska mreža može se prikazati u obliku *grafa.* Čvorovi u grafu predstavljaju *uređaje koji su deo mreže, a* grane između njih *komunikacione kanale* kojima su uređaji fizički povezani (Slika 6.1).



Slika 6.1: Grafički prikaz komponenti računarske mreže

Spoljašnji (završni) čvorovi u mreži (koji su najčešće povezani samo sa jednim susedom, a na Slici 6.1 prikazani tamnom nijansom žute boje) predstavljaju *korisničke uređaje* u mreži. Tradicionalno, ovi uređaji su najčešće računari, ili pomoćni uređaji poput štampača i skenera. U današnje vreme, umesto samo ovih tipova uređaja, u mrežama na ovim mestima se pojavljuju i drugi digitalni uređaji koji imaju mogućnost umrežavanja (telefoni, tableti, kamere, pametni satovi, pametni kuhinjski uređaji, automobili i drugo). Na njima se izvršavaju *aplikacije* pokrenutne od strane korisnika koji žele da komuniciraju međusobno.

Unutrašnji čvorovi u mreži (koji su najčešće povezani sa više od jednog suseda i prikazani na Slici 6.1 svetlijom nijansom žute) predstavljaju specijalizovane *mrežne uređaje* koji prosleđuju poruke između čvorova. Najčešće su to *ruteri, svičevi i habovi* (Slika 6.2).

Sve ove komponente mogu se podeliti u tri glavne kategorije: *mrežni hardver*, *komunikacioni kanali* i *mrežni softver.*

Elektronska verzija (2021)

Slika 6.2: Mrežni uređaji (ruter, svič i hab)

6.2.1 Mrežni hardver

Svaki uređaj na mreži mora da poseduje specijalizovani deo hardvera koji mu obezbeđuje pristup toj mreži, tj. komunikaciju sa ostalima. Najčešće je to *mrežna kartica*, tj. *mrežni adapter — NIC* (Slika 6.3) koji omogućava fizički pristup uređaja ka komunikacionim kanalima koji ga povezuju sa ostatkom mreže. Svaka mrežna kartica ima svoj *identifikator* u formi fizičke *(MAC)* adrese koja joj se dodeljuje pri proizvodnji, globalno je jedinstvena, a služi kako bi se uređaji jednoznačno identifikovali u međusobnoj komunikaciji. Ona obezbeđuje pristup žičanim ili bežičnim komunikacionim kanalima. Moderne matične ploče na računarima uglavnom poseduju integrisane mrežne adaptere, mada oni mogu biti i eksterni koji se po potrebi dodaju i na taj način jedan uređaj može imati i više adaptera, tj. izlaza na mrežu. U ove svrhe koriste se i drugi specijalizovani uređaji, kao što su *modemi*, *Wi-Fi adapteri* (eng. *Wi-Fi dongles*) i drugi.

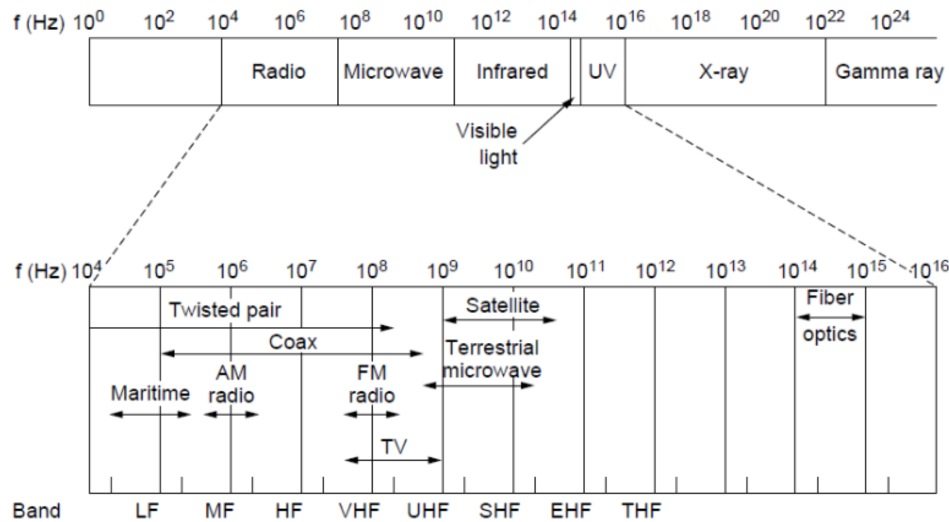
Slika 6.3: Mrežni hardver — različiti tipovi mrežne kartice

6.2.2 Komunikacioni kanali

Pod komunikacionim kanalima podrazumevaju se fizički medijumi koji spajaju čvorove u mreži. Oni mogu biti realizovani *žičano* u formi kablova i *bežično* u formi elektromagnetnih talasa koji se emituju kroz etar (radio talasi, mikro talasi, infracrveni i ultraljubičasti zraci).

Osim ove glavne razlike, komunikacioni kanali se razlikuju i u *6brzini* kojom prenose podatke (eng. *throughput*). Ona se meri u broju bita koje kanal može da prenese u jedinici vremena — *bit po sekundi.* U današnje vreme aktuelna tehnologija dozvoljava prenos ogromnog broja bitova po sekundi, pa se prenos češće izražava u jedinicama *Megabit* (milion bita) po sekundi u oznakama Mbps, Mbit/s ili Mb/s, kao i u *Gigabit* (milijarda bita) po sekundi u oznakama Gbps, Gbit/s ili Gb/s. Iako fizični signali koji putuju kablom ili bežično se kreću brzinama približnim brzini svetlosti, oni se razlikuju u *opsezima svojih frekvencija* (eng. *bandwidth*) od kojih direktno zavisi brzina prenosa podataka. Trenutno se za prenos informacija koriste signali frekvencija od 10^4Hz do 10^{16}Hz (Slika 6.4). Brzina prenosa podataka je *direktno srazmerna* frekvencijskom opsegu. Signali veće frekvencije kakvi se nalaze u optičkim kablovima imaju mogućnost za mnogo veće brzine prenosa podataka od signala u drugim medijumima.

Žičani komunikacioni kanali mogu se realizovati na sledeće načine (Slika 6.5):



Slika 6.4: Podela frekvencionog spektra

Upredene parice (eng. *unshielded twisted-pair*) — *UTP kablovi* predstavljaju uvijene uparene izolovane bakarne žice koje su se inicijalno koristile u telefoniji, ali novije kategorije ovih kablova koriste se i u računarskim mrežama. Oni nude brzine prenosa podataka od 4Mbps, pa čak i do par Gbps.

Koaksijalni kablovi (eng. *coaxial cable*) su se inicijalno koristili za prenos televizijskog signala, a razvojem računarskih mreža koriste se i u lokalnim mrežama. Generalno su pouzdaniji i brži od UTP kablova, sa brzinama od 35Mbps do 10Gbps.

Optički kablovi (eng. *fiber optic cable*) se za razliku od prethodna dva tipa kablova prave od mnogobrojnih veoma tankih *staklenih vlakana* obmotanih izolatorom kroz koje protiče svetlost koju emituje *laser* na jednom kraju vlakana. Na drugom kraju vlakana nalazi se *fotodetektor* koji prihvata poslati svetlosni talas. Spoljašnji uticaji na signal u ovom tipu kabla je još manji, frekvencija svetlosnog signala veća, pa su i brzine koje se postižu u ovim tipovima kablova najveće. U eksperimentalnim laboratorijskim uslovima, koristeći snopove kablova dosegnute su brzine čak i preko 1Pbps (milion Gbps). Kao takavi, koriste se kao okosnica interneta povezujući bitna i velika čvorišta. U novije vreme ovim tipom kabla povezuju se i pojedinačne zgrade (eng. *fiber to the building - FTTB*) i stanovi (eng. *fiber to the home - FTTH*), odakle se do pojedinačnih uređaja razvlače UTP, koaksijalni kablovi ili bežična mreža.

Slika 6.5: Žičani komunikacioni kablovi (UTP, koaksijalni i optički kabl)

Bežična komunikacija se razlikuje od žičane tehnologije po tome što se signali ne prostiru kroz kablove, već se *emituju kroz etar*. Ovakav način komunikacije prirodno podržava *mobilnost* nudeći krajnjem korisniku često

željenu fleksibilnost pri korišćenju prenosnih računara, telefona, tableta i drugih pametnih uređaja koji ne stoje konstantno na jednoj lokaciji. Neke od danas korišćenih tehnologija bežičnog prenosa su:

Bluetooth tehnologija koristi se za komunikaciju na malim udaljenostima, do svega par desetina metara i 2Mbps brzine. Najčešće povezuje računar sa perifernim uređajima kao što su slušalice, miš, tastatura, razni senzori i slično.

Wi-Fi, ili bežični LAN (WLAN) koristi mikro talase frekvencije 2.4GHz ili 5GHz za komunikaciju između uređaja na rastojanju do maksimalno nekoliko stotina metara na otvorenom, ili desetina metara u zatvorenom prostoru. Trenutna tehnologija dozvoljava brzine do nekoliko Gbps.

Ćelijski sistemi su tehnologija bežične komunikacije koja se koristi u mobilnoj telefoniji, gde se signal prenosi preko niza antena. Razvijala se kroz generacije, od prve do trenutno aktuelne pete generacije (5G mobilna telefonija), gde je svaka sledeća donosila poboljšanja u dostupnim brzinama prenosa.

Zemaljski mikrotalasi su tehnologija komunikacije mikrotalasima niske frekvencije između optički vidljivih antena udaljenih međusobno čak i do više desetina kilometara.

Sateliti su tehnologija posredne komunikacije između dve tačke na velikim udaljenostima koje nemaju optičku vidljivost. Odvija se korišćenjem satelita koji se nalaze u orbiti planete Zemlje na udaljenostima od nekoliko stotina do par desetina hiljada kilometara. Koriste se za prenos kako televizijskog i telefonskog signala, tako i za pristup internetu na lokacijama gde su drugi vidovi komunikacije nemogući ili previše skupi.

6.2.3 Mrežni softver

Kako bi mrežni hardver mogao da funkcioniše, potreban je i softver koji će ga *osposobiti i kontrolisati* njegov rad. Kao i svaki kompleksni sistem, mrežni softver je ogranizovan *hijerarhijski*, tj. podeljen je po slojevima. Na nižim slojevima nalazi se deo koji kontroliše sam rad fizičkih uređaja, zadužen je za konverziju bitova u fizičke signale, (de)kodiranje, (de)multipleksiranje, i slično, i kao takav nalazi se implementiran u operativnim sistemima, tj. u *upravljaču (drajveru) mrežnih kartica*, dok se na višim slojevima nalaze delovi koji su više aplikativno orjentisani i nude korisnicima konkretne usluge i servise, kao što su slanje i prijem elektronske pošte, pregledanje Veba, četovanje i slično.

6.3 Tipovi računarskih mreža

Osim po tehnologiji kojom se odvija komunikacija, računarske mreže se mogu međusobno razlikovati po korišćenoj *arhitekturi*, *rasponu* koji mreža pokriva i *topologiji* same mreže.

6.3.1 Arhitektura računarskih mreža

Arhitektura mreže se odnosi na dizajn mreže, uključujući njene komponente, njihov međusobni raspored i odnose, kao i protokole po kojima te komponente komuniciraju. Glavni tipovi mrežne arhitekture su:

Arhitektura klijent-server: U ovom modelu, *klijenti* (uređaji krajnjih korisnika) zahtevaju *usluge* ili resurse od *centralizovanih servera*, koji obrađuju njihove zahteve. Ovo je danas najčešća arhitektura računarskih mreža. Serveri su uglavnom posvećeni određenom tipu zadataka (npr. veb serveri, serveri datoteka, ...).

U samim počecima razvoja računarstva, računarski sistemi su se odlikovali velikim centralnim računarima na koje su bili povezani terminali koji su služili samo za unos podataka i prikaz rezultata. Ovakva organizacija podseća na današnje klijent-server okruženje u kome su mogućnosti klijenata svedene na proste ulazno-izlazne funkcije.

Peer-to-Peer (P2P) arhitektura: Ovaj vid arhitekture predstavlja *mrežu ravnopravnih računara*, za razliku od prethodnog tipa arhitekture gde postoji jasna razlika u ulogama između klijenata i servera. Računari direktno komuniciraju jedan sa drugim, deleći podatke i resurse. Sistem je *decentralizovan*, tj. nije potreban centralni server, i često se koristi u aplikacijama za deljenje velikih datoteka (npr. Napster i Bittorent).

Arhitektura oblaka: Ova mrežna arhitektura omogućava isporuku računarskih usluga preko Interneta, tj. *iznajmljivanje skladištenog prostora, procesorske snage, memorije* i slično od strane krajnjih korisnika *na njihov zahtev.* Analogno deljenju štampača kao resursa od strane više korisnika preko mreže, moguće je deliti i procesorske i memorijske resurse. Ovim krajnji korisnici dobijaju *fleksibilnost* i *skalabilnost* u alokaciji svojih resursa, po ceni koja često može biti manja nego kupovina sopstvenih resursa tog tipa.

Elektronska verzija (2024)

Osim pomenutih tipova arhitekture računarskih mreža postoje druge, kao što su troslojna arhitektura, servisno orijentisana arhitektura (SOA), arhitektura mikroservisa i hibridne arhitekture.

6.3.2 Raspon računarskih mreža

Računarske mreže se mogu klasifikovati i po *geografskom rasponu* koji mreža pokriva. Ova podela je u tesnoj vezi sa korišćenom tehnologijom za komunikaciju jer su neke tehnologije vezane za komunikaciju na manjim rastojanjima (recimo Bluetooth), dok se druge češće koriste kada je signale potrebno preneti na vrlo udaljene lokacije (optički kablovi i sateliti).

Personal area network (PAN) je mreža najmanjeg raspona i koristi se za potrebe jednog korisnika, recimo da spoji računar, telefon i slušalice. Udaljenost uređaja je do nekoliko metara i veza može biti žičana ili bežična.

Local area network (LAN) se koristi za povezivanje većeg broja uređaja na relativno malim rastojanjima, recimo jedan stan ili sprat, nekoliko kancelarija i slično. Tradicionalno, ovi uređaji su se povezivali UTP kablovima, ali u novije vreme se vrlo često koristi i bežična mreža (WLAN).

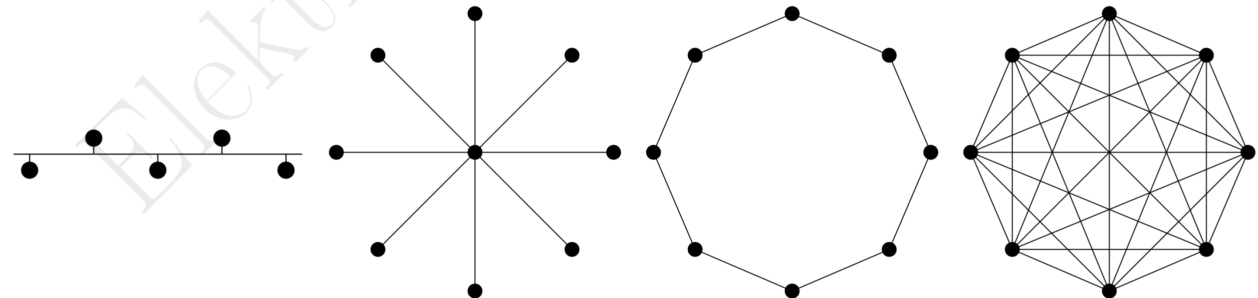
Campus area network (CAN) povezuje više lokalnih mreža u jednu na bliskim lokacijama, recimo u okviru jednog univerziteta, kompanije, studentskog internata i slično. Tehnologija podrazumeva tehnologiju lokalnih mreža, dok se za povezivanje lokalnih mreža između sebe najčešće koriste brži koaksijalni kablovi, UTP kablovi više kategorije ili bežična komunikacija (u slučajevima kada je postavljanje fizičkih kablova nemoguće ili previše skupo).

Metropolitan area network (MAN) takođe poveziju lokalne mreže, ali je njihov broj veći i prostiru se na većem geografskom području nego kod prethodnog tipa mreže. Ovakve mreže pokrivaju cele gradove i za veze između svojih delova koriste brze optičke kablove.

Wide area network (WAN) pokriva prostranstva šira od jednog grada, najčešće čitavu oblast ili državu. Njenu infrastrukturu održavaju telekomunikacione kompanije koje naplaćuju korisnicima pristup istoj. Za povezivanje bitnih čvorišta u mreži koriste se veliki snopovi optičkih kablova, kao i komunikacioni sateliti. Internet je najveća mreža ovog tipa.

6.3.3 Topologija mreža

Način kako su elementi mreže povezani između sebe, raspored čvorova u mreži i kako teče komunikacija između njih predstavlja topologiju mreže. Ona direktno određuje performanse mreže, tj. pouzdanost, skalabilnost i cenu. Postoje osnovna četiri tipa topologije mreža (Slika 6.6):



Slika 6.6: Topologije računarskih mreža (magistrala, zvezda, prsten i potpuna povezanost)

Magistrala je topologija mreže kod koje su sve komponente povezane na *zajednički deljeni kanal*. Signal koji je šalje od jednog čvora ka nekom drugom postavlja se na kanal i dostupan je svim čvorovima koji su povezani na njega. Teorijski, u slučaju prekida magistrale, mreža se deli na dva nepovezana dela, pa čvorovi iz jednog dela ne mogu da komuniciraju sa čvorovima u drugom delu mreže. U slučaju velikog opterećenja može doći do čestog sudaranja poslatih paketa na samoj magistrali, tj. do *zagušenja kanala*, jer se njegov kapacitet ravnopravno deli na sve korisnike. U praksi, kao magistrala, najčešće se koristi koaksijalni kabl. Ovo je u početku bila dominantna topologija lokalnih mreža.

Elektronska verzija (2024)

Zvezda je topologija gde su svi čvorovi u mreži povezani isključivo na jedan *centralni čvor*, pa se sva komunikacija između čvorova odvija preko njega. Za razliku od magistrale gde postoji jedan deljeni kanal, u ovoj topologiji svaki čvor mreže ima svoj kanal do centralnog čvora, pa se zagušenja po pravilu ne javljaju na kanalima, već na centralnom čvoru. To se rešava tako što se kao centralni čvor koristi *svič* koji obezbeđuje nezavisne veze svih čvorova međusobno. U slučaju prekida jednog komunikacionog kanala, svi čvorovi osim tog jednog čiji je kanal u prekidu mogu komunicirati bez problema, dok se u slučaju otkaza centralnog čvora gubi sva komunikacija.

Prsten je topologija slična magistrali kod koje je kanal *kružan*, tj. u obliku prstena. Podaci u ovoj topologiji se šalju samo u jednom smeru, pa se na taj način smanjuje mogućnost sudaranja paketa na kanalu. U slučaju otkaza bilo kog čvora, ostali čvorovi u mreži mogu da komunicijraju nesmetano, dok u slučaju prekida kanala dolazi do totalnog prekida komunikacije.

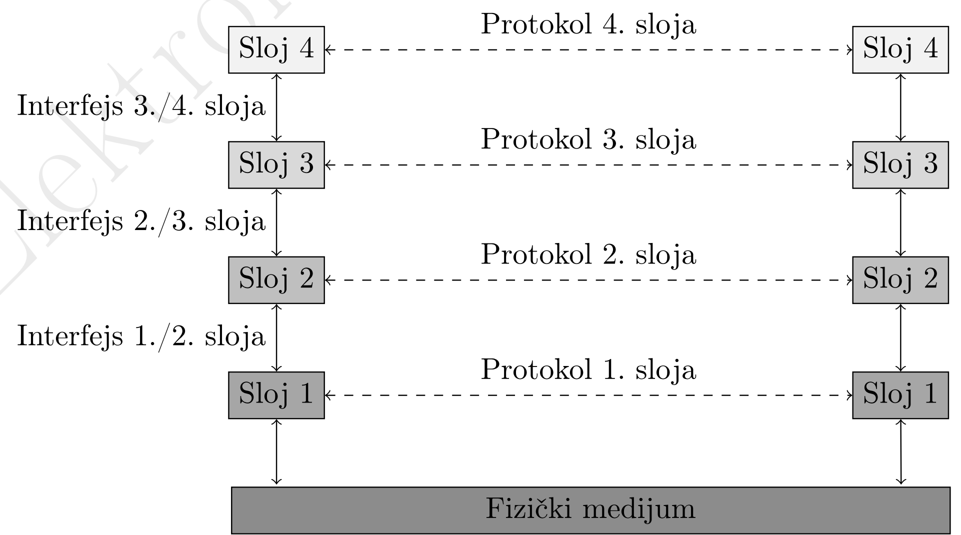
Potpuna povezanost je topologija kog koje su svi čvorovi mreže međusobno *direktno povezani*. Nije pogodna za veće mreže jer je cena postavljanja takve mreže velika. Pri dodavanju novog čvora u postojeću mrežu, potrebno je spojiti novi čvor posebnim kanalima sa svim postojećim čvorovima, što drastično povećava cenu. Ovakva topologija obezbeđuje veću pouzdanost komunikacije jer između svaka dva čvora u mreži, osim direktne veze postoji veći broj redudantnih posrednih veza. Zbog toga se otkazom jednog kanala u komunikaciji ne gubi ništa.

Predstavljene topologije se odnose na lokalne mreže, tj. na mreže sa malim brojem čvorova. Veće mreže nastaju spajanjem manjih, gde svaka od njih može imati svoju topologiju. Ako manje mreže koje ulaze u sastav većih posmatramo kao entitete za sebe, onda i te veće mreže (mreže manjih mreža) imaju svoju globalnu topologiju koja može biti jedan od pomenuta četiri tipa, nezavosno od topologije prisutne u manjim mrežama.

6.4 Slojevitnost mreže i mrežni protokoli

Iako pojam mreže vrlo jednostavno deluje, omogućiti da dve aplikacije na različitim računarima komuniciraju je vrlo kompleksan posao koji zahteva angažovanje eksperata iz različitih oblasti, od hemičara, fizičara i elektrotehničara na hardverskom nivou mreža, do matematičara, računaraca i programera na softverskom nivou.

Zbog svega toga, mreže i mrežni softver se izgrađuju *hijerarhijski*, po novoima, tj. *slojevima* (Slika 6.7). Svaki sloj ima precizno definisanu ulogu i funkciju u celokupnoj komunikaciji, tj. on rešava precizno definisan zadatak. Komunikacija se na određenom sloju u mreži odvija poštujući unapred dogovorena pravila komunikacije, tj. *protokole* odgovarajućeg sloja. Par entiteta koji međusobno komunicijraju po datom protokolu na istom sloju u mreži nazivaju se *parnjaci*. Stvarna komunikacija ne ide direktno između njih, već oni koriste slojevi ispod njih, koji im nude određenu uslugu po unapred dogovorenom *interfejsu*.



Slika 6.7: Slojevitost mreža

Dizajn mreže definiše broj slojeva i protokole u njima, ali najuticajnija su sledeća dva *referentna modela*:

Open Systems Interconnection — OSI je teorijski gledano najbitniji referentni model, definisan od strane Internacionalne organizacije za standarde (ISO), koji definiše *7 slojeva*: *fizički sloj, sloj veze podataka,*

Elektronska verzija (2024)

mrežni sloj, transportni sloj, sloj sesije, sloj prezentacije i aplikativni sloj. Slojevi su navedeni od najnižeg do najvišeg. Na fizičkom sloju se rešavaju hardverski problemi, prenos signala kroz kanal, modulacija signala, multipleksiranje, i slično. Na višim slojevima komunikacija postaje sve više apstraktna. Hardverski problemi bivaju zamenjeni softverskim, pa se na vrhu, u aplikativnom sloju razmatraju protokoli koje direktno koriste dve aplikacije u komunikacji. Svi detalji te komunikacije (brzina slanja, prenos paketa, adresiranje, rutiranje, detektovanje i obrada grešaka u komunikaciji i slično) skriveni su i rešeni na nižim slojevima. U praksi se ovaj referentni model ne koristi kao previše detaljan, raslojen i komplikovan.

TCP/IP je referentni model koji se praktično koristi na Internetu. Jednostavniji je od prethodnog modela, tj. ima *4 sloja* gde je najniži nastao spajanjem dva najniža sloja OSI modela, a aplikativni sloj na vrhu u sebi integriše i dva sloja ispod sebe (sloj sesije i prezentacije).

Najpoznatiji protokoli u računarskim mrežama su:

Internet protokol (IP) je protokol za komunikaciju na mrežnom sloju Interneta. Četvrta verzija ovog protokola *(IPv4)* je trenutno u upotrebi, iako se već godinama postepeno uvodi šesta *(IPv6)*. Osnovni problem koji se rešava ovim protokolom je *rutiranje paketa*, tj. pronalazak puta kojom će podaci putovati od pošaljioca do primaoca. Putanja se nalazi na osnovu adrese primaoca, tzv. *IP adrese*, koja je u IPv4 32-bitna i predstavlja se kao četiri dekadna broja (od po 8 bita), u rasponu od 0 do 255, razdvojena tačkama, recimo 192.168.0.1.

Transfer Control Protocol (TCP) i User Datagram Protocol (UDP) su protokoli transportnog sloja, tj. nalaze se na sloju iznad sloja sa Internet protokolom, pa koriste njegove usluge za svoju realizaciju. TCP je kompleksniji protokol, kod koga postoji uspostava veze pre same komunikacije, kontrola i korekcija grešaka, kontrola brzine prenosa i zagušenja, pa se smatra pouzdanim protokolom. Za razliku od njega, UDP nema sve ove mehanizme, pa je nepouzdan, ali je prostiji i zato neuporedivo brži. Kao takav primenjuje se u audio i video komunikaciji u realnom vremenu, gde je brzina presudni faktor.

HyperText Transfer Protocol (HTTP) je protokol aplikativnog sloja zadužen za prenos HTML stranica, po kome komuniciraju pregledači veba (eng. *browser*) i veb serveri.

SMTP, POP3, IMAP su protokoli aplikativnog sloja zaduženi za slanje i primanje elektronske pošte.

File Transfer Protocol (FTP) je protokol aplikativnog sloja zadužen za prenos datoteka.

SSH je protokol aplikativnog sloja zadužen za bezbedno slanje komandi udaljenom računaru preko Interneta.

6.5 Bezbednost u računarskim mrežama

U početku razvoja računarskih mreža, kada su se one koristile uglavnom u akademskoj zajednici za slanje i primanje elektronske pošte, kao i u kompanijama za deljeni pristup štampačima, bezbednost u mrežama nije igrala bitnu ulogu. Danas, kada milioni korisnika svakodnevno koriste internet za mobilno bankarstvo, kupovinu i slično, bezbednost na mrežama zavređuje mnogo više pažnje. Bezbednost mreže podrazumeva zaštitu računarskih mreža od uljeza, napada i drugih pretnji.

Koristeći praktična rešenja razvijana decenijama, tehnologije i dostignuća iz različitih oblasti, cilj bezbednosti u računarskim mrežama je *ostvarivanje integriteta, poverljivosti i dostupnosti umreženih sistema.* U svojoj najprostijoj formi ona obezbeđuje da zlonamerni korisnici ne mogu pročitati, ili još gore izmeniti, poruke namenjene drugim osobama. Bezbednost u mrežama se bavi zabranom pristupa udaljenim servisima i resursima osobama koje nemaju prava da im pristupaju, ali se bavi i problemom identifikacije korisnika. Identifikacija rešava problem u kome korisnik dobija elektronsku poštu potpisanu od strane recimo neke zvanične institucije, a zapravo ju je poslao zlonameran korisnik koji pokušava da prevari primaoca i na taj način ostvari neku korist za sebe.

Najčešće vrste pretnji uključuju:

Zlonamerni softver (eng. *malware*) je softver koji je dizajniran da ošteti ili iskoristi bilo koji uređaj na koji se može instalirati.

Pecanje (eng. *fishing*) predstavlja pokušaje malicioznih korisnika da se pojedinci prevare da otkriju osetljive informacije. Najčešće je u vidu elektronske pošte gde je cilj ubediti korisnika da komunicira sa nekom zvaničnom (samim tim i bezbednom) institucijom ili svojim poznanikom, kojima će dobrovoljno dati svoje bezbedno osetljive podatke (šifre, bankovne račune, brojeve platnih kartica, pinove i slično).

Elektronska verzija (2024)

DOS napad (eng. *denial of service*) je hakerski napad koji prouzrokuje preopterećenje sistema kako bi bio nedostupan ostalim korisnicima.

Čovek u sredini (eng. *man in the middle*) napad predstavlja presretanje i menjanje komunikacije između dve strane koje i dalje veruju da komuniciraju neposredno.

Skoro sva mrežna bezbednost zasniva se na *kriptografiji* i njenim principima. Mere bezbednosti podrazumevaju postojanje sledećeg:

Zaštitni zidovi (eng. *firewall*) predstavljaju uređaje ili softver koji nadgledaju i kontrolišu sav dolazni i odlazni mrežni saobraćaj na osnovu unapred određenih bezbednosnih pravila.

Sistemi za otkrivanje upada (eng. *Intrusion Detection Systems — IDS*) koji konstantno nadgledaju mrežu u potrazi za sumnjivim aktivnostima i potencijalnim pretnjama.

Šifrovanje (eng. *encryption*) je proces osiguravanja podataka kodiranjem. Na taj način, čineći ih nečitljivim, sprečava se pristup podacima neovlašćenim osobama.

Virtuelne privatne mreže (eng. *Virtual Private Networks — VPN*) predstavljaju mrežnu arhitekturu koja ima za cilj da proširi privatnu mrežu jednom ili više mreža koje se nalaze odvojeno i na taj način obezbede siguran, izolovan udaljeni pristup privatnim mrežama. Uglavnom se koristi u kompanijama koje omogućavaju svojim zaposlenima ili korisnicima (koji se fizički ne nalaze u njihovoj lokalnoj mreži) pristup servisima koji nisu javno dostupni svima.

Kako bi računarska mreža bila bezbedna, neophodno je održavati sav (a posebno mrežni) softver *ažurnim*, redovno *obučavati* korisnike o potencijalnim rizicima i *bezbednom ponašanju* i uspostaviti *restriktivnu kontrolu pristupa* resursima zasnovanu na odgovarajućim korisničkim ulogama.

Elektronska verzija (2024)

Elektronska verzija (2024)

Glava 7

Programski jezici i prevodioci

Početak razvoja programskih jezika je bio u bliskoj vezi sa razvojem računara tj. sa razvojem hardvera 1940-tih godina. 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, tj. direktno na mašinskom jeziku ili na asembleru. Ovi programski jezici nazivaju se i *niži* programski jezici zbog svoje direktne povezanosti sa hradverom.

Prvi programski jezici zahtevali su od programera da bude upoznat sa najfinijim detaljima računara za koji se piše program. 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, nije ih bilo moguće *preneti* na drugačije računare već je za njih bilo neophodno pisati nove programe. Iako su prvi računari bili skupoceni, mnoge kompanije su još više novca trošile na razvoj softvera, zbog kompleksnosti programiranja na niskom nivou. Već u to vreme postojala je ideja o razvoju apstraktnijih programskih jezika koji bi automatski bili prevođeni na mašinski jezik.

Polovinom 1950-ih godina nastali su *viši programski jezici*. Viši programski jezici sakrivaju detalje konkretnih računara od programera i drastično menjaju programiranje. U narednom periodu, dalji razvoj programskih jezika vodio je ka sve većem udaljavanju od hardvera i prilagođavanju programerima, sa ciljem da se programiranje učini lakšim, udobnijim i efikasnijim.

Da bi viši programski jezici mogli da se koriste postoje specijalizovani programi, tzv. *jezički procesori* ili *programski prevodioci*. Programski prevodioci omogućavaju lakšu komunikaciju čoveka i računara. Oni automatski prevode program napisan na višem programskom jeziku, dakle jeziku koji je blizak programeru, na mašinski jezik, jezik koji računar može da izvrši. Na taj način, programer ne mora da bude upućen u različite vrste arhitektura računara, a prenosivost napisanog programa se prebacuje sa programera na programski prevodilac. Razvoj programskih jezika usko je vezan sa razvojem programskih prevodilaca.

Ne postoji *najbolji* programski jezik već se za različite potrebe biraju različiti jezici, a svakodnevno se radi na unapređivanju postojećih i pravljenju novih programskih jezika. Programski jezici nastaju, razvijaju se i nestaju već više od sedamdeset godina. Neki izvori navode da je u realnoj upotrebi bilo do sada oko 250 jezika, a neki izvori koji pretenduju da popišu sve programske jezike koji su ikad postojali navode više od 9000 konkretnih jezika. Veliki broj programskih jezika rezultat je napora da se ubrza i olakša programiranje, odnosno da se nađu najbolji načini za rešavanje različitih praktičnih problema. Programer, u toku svog školovanja, pa i u toku svoje celokupne karijere, ne može da u potpunosti savlada sve bitne i aktuelne programske jezike. Međutim, ukoliko se programski jezici izučavaju uočavanjem bitnih svojstva i karakteristika postojećih jezika, to omogućava da se reletivno lako razumeju i po potrebi brzo savladaju novi programski jezici.

Ne postoji ni *najzastupljeniji* ili *najpopulariniji* programski jezik jer su polja i načini primene za neke jezike skoro neuporedivi. Indeks TIOBE (eng. *TIOBE Programming Community index*) još od davne 1985. godine meri popularnost programskih jezika koristeći razne faktore, kao što su broj programera u svetu koji vladaju nekim jezikom, broj univerzitetskih kurseva, broj kompanija koje koriste neki jezik i slično. Za rangiranje se koristi više popularnih pretraživača veba. Jezik C je dugo godina bio uvek na prvom ili drugom mestu, od 2000. godine jezik Java je takođe dugo bio na prvom ili drugom mestu, a C++ u prvih pet. U januaru 2021. godine poredak prvih osam jezika bio je ovakav: C, Java, Pajton, C++, C#, Visual Basic, JavaScript, PHP. U septembru 2024. godine poredak prvih osam jezika bio je ovakav: Python, C+, Java, C, C#, JavaScript, Visual Basic, Go. Po zastupljenosti koda u određenom programskom jeziku na portalu `github`, poredak je vrlo sličan. Među korisnicima foruma `stackoverflow` koji okuplja hiljade informatičara širom sveta, spisak najčešće korišćenih jezika u godini 2020, izgleda ovako: JavaScript, HTML/CSS, SQL, Python, Java, Bash/Shell, C#,

PHP, TypeScript, C++, C.

Razvojni ciklus jednostavnih programa savremenih viših programskih jezika teče na sledeći način[1]. Nakon faze razumevanja problema koji se rešava, prva faza u razvoju programa je njegovo *pisanje* kojim se u računar unosi *izvorni program* ili *izvorni kôd* (engl. *source code*) pomoću nekog editora teksta ili razvojnog okruženja. Naredna faza je njegovo *prevođenje*, kada se na osnovu izvornog programa na višem programskom jeziku dobija prevedeni kôd na asemblerskom odnosno mašinskom jeziku. Ovako dobijen kôd se naziva *objektni kôd* (engl. *object code*). U fazi *povezivanja* više objektnih programa povezuje se sa objektnim kodom iz standardne biblioteke u jedinstvenu celinu koja se naziva izvršivi program (engl. *executable program*). Povezivanje vrši specijalizovan program *povezivač* ili *uređivač veza*, koji se često naziva i *linker* (engl. *linker*). Faza prevođenja i faza povezivanja često nisu jasno razdvojene, odnosno prevodilac automatski poziva linker nakon faze prevođenja. 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.

7.1 Klasifikacije programskih jezika

Najopštija podela viših programskih jezika je podela po načinu rešavanja problema. Po načinu rešavanja problema, programski jezici se dele na *proceduralne* i *deklarativne*. Većina programskih jezika danas je *proceduralna* što znači da je zadatak programera da precizno opiše način (proceduru) kojim se dolazi do rešenja problema.

Primer 7.1 (Proceduralno programiranje)**.** *Napisati program koji za unete različite cene artikala određuje koji je jeftiniji.*

Proceduralno rešenje ovog problema podrazumeva pisanje algoritma kojim se izračunava jeftiniji artikal. Alogoritam se može definisati na sledeći način.

```
Unesi cenu prvog artikala
Unesi cenu drugog artikala
Da li je cena prvog artika manja od cene drugog artikla?
Ako jeste, odštampaj "Prvi artikal je jeftiniji"
Inače, odštampaj "Drugi artikal je jeftiniji"
```

Ovaj algoritam se može prevesti u naredni Python kôd:

```
cena1 = int(input('Unesi cenu prvog artikla: '))
cena2 = int(input('Unesi cenu drugog artikla: '))

if cena1 < cena2:
    print('Prvi artikal je jeftiniji')
else:
    print('Drugi artikal je jeftiniji')
```

Pokretanjem prethodnog programa za ulazne vrednosti 1000 i 1500 dobija se naredni izlaz

```
Unesi cenu prvog artikla:
1000
Unesi cenu drugog artikla:
1500
Prvi artikal je jeftiniji
```

Značajni proceduralni programski jezici su, na primer, C, Pascal, Python i Java.

Nasuprot proceduralnim jezicima, *deklarativni* programski jezici od programera zahtevaju da precizno opiše problem, dok se mehanizam programskog jezika onda bavi pronalaženjem rešenja problema. Ovo u mnogome olakšava proces programiranja, međutim, zbog nemogućnosti automatskog pronalaženja efikasnih algoritama koji bi rešili široku klasu problema, domen primene deklarativnih jezika je često ograničen. Deklarativni jezici nisu među dominantnim jezicima opšte namene, ali se uspešno koriste u mnogim specifičnim domenima.

Primer 7.2 (Deklarativno programiranje)**.** *Napisati program koji pronalazi sva rešenja naredne kriptoaritmetike:*

[1] Ukoliko se razmatra razvoj netrivijalnog softvera, on obuhvata značajno veći broj faza koje ovde nisu pomenute.

```
ONE + ONE = TWO
```

U kriptoaritmetikama, različitim slovima odgovaraju različite cifre i početna cifra broja ne može biti nula.

Deklarativno rešenje ovog problema podrazumeva da se precizno opiše problem. Najpre je potrebno da se zadaju moguće vrednosti promenljivih O, N, E, T i W. To su vrednosti iz intervala $[0,9]$*, pri čemu O i T ne mogu biti 0. Dalje je potrebno precizno napisati ograničenje koje treba da važi. To se može zapisati na sledeći način:*

```
  O*100 + N*10 + E
+ O*100 + N*10 + E
= T*100 + W*10 + O
```

U programskim jezicima koji imaju podršku za programiranje ograničenja *prethodni uslovi se mogu direktno preslikati u kôd. Na primer, u programskom jeziku Prolog, to se može zapisati na sledeći način*

```
crypto :-
    % Definisi promenljive O, N, E, T, W kao cifre od 0 do 9
    Digits = [O, N, E, T, W],
    Digits ins 0..9,

    % T i O ne smeju biti 0 (jer su brojevi TWO i ONE trocifreni)
    T #\= 0,
    O #\= 0,

    % Sve cifre moraju biti razlicite
    all_different(Digits),

    % Jednakost kroptoaritmetike: ONE + ONE = TWO
    O * 100 + N * 10 + E + O * 100 + N * 10 + E #= T * 100 + W * 10 + O,

    % Nadji resenje za svako slovo.
    label(Digits),

    % Odstampaj resenje.
    format('O = ~d, N = ~d, E = ~d, T = ~d, W = ~d~n', [O, N, E, T, W]),
    format('ONE = ~d~d~d, TWO = ~d~d~d~n', [O, N, E, T, W, O]).
```

Pokretanjem programa dobija se odgovor

```
O = 2, N = 0, E = 6, T = 4, W = 1
ONE = 206, TWO = 412
```

Značajniji deklarativni programski jezici su jezici Prolog i SQL.

7.1.1 Programske paradigme

Programske paradigme predstavljaju različite stilove programiranja koji često služe i za klasifikaciju programskih jezika. Paradigme se razlikuju po konceptima i apstrakcijama koje se koriste da bi se predstavili elementi programa (na primer, promenljive, funkcije, objekti, ograničenja) i koracima od kojih se sastoje izračunavanja (dodele, sračunavanja vrednosti izraza, tokovi podataka, itd.). Broj programskih paradigmi nije tako veliki kao broj programskih jezika. Izučavanje programskih paradigmi značajno olakšava razumevanje programskih jezika kao i očekivanja i mogućnosti koje jezik pruža. Poznavanje određene paradigme nam omogućava da brže i lakše savladamo svaki programski jezik koji toj paradigmi pripada.

7.1.1.1 Osnovne programske paradigme

U okviru programskih paradigmi, jasno su razgraničena četiri stila programiranja: *imperativno*, *funkcionalno*, *objektno-orjentisano* i *logičko* programiranje.

Imperativni jezici. Najkorišćeniji programski jezici danas spadaju u grupu *imperativnih* programskih jezika. U ovim jezicima stanje programa karakterišu *promenljive* kojima se predstavljaju podaci i *naredbe* kojima se vrše određene transformacije promenljivih.

Imperativni jezici razmatraju izračunavanje kao niz iskaza (naredbi) koje menjaju *stanje* programa određeno tekućim vrednostima promenljivih. Vrednosti promenljivih se menjaju naredbom dodele, a kontrola toka programa se vrši korišćenjem sekvence (nizanje naredbi), selekcije (izbor koja će naredba biti izvršena u zavisnosti od uspunjenosti nekog uslova) i iteracije (ponavljanje izvršavanja naredbi). Imperativni jezici su obično izrazito proceduralni.

Primer 7.3 (Programski jezik C)**.** *Program koji određuje koji je od dva artikla jeftiniji se na programskom jeziku C može napisati na sledeći način.*

```
#include<stdio.h>
int main() {
  int cena1, cena2;

  printf("Unesi cenu prvog artikla\n");
  scanf("%d", &cena1);
  printf("Unesi cenu drugog artikla\n");
  scanf("%d", &cena2);

  if(cena1 < cena2)
      print("Prvi artikal je jeftiniji\n");
  else
      printf("Drugi artikal je jeftiniji\n");

  return 0;
}
```

Imperativni jezici, uz objektno-orijentisane jezike, se najčešće koriste u industrijskom, sistemskom i aplikativnom programiranju. U nastavku su dati primeri najznačajnijih imperativnih jezika.

Fortran je prvi viši programski jezik koji je nastao u periodu 1953-1957 godine i koji je kasnije stekao širok krug korisnika. Ime Fortran nastalo je kao skraćenica od *FORmula TRANslating System* i dugo godina je pisano velikim slovima, kao akronim[2]. Projektovanje i razvoj Fortrana vodio je Džon Bakus (engl. *John Backus*) u okviru kompanije IBM. Prvi interpretator za Fortran bio je razvijen 1953. godine. Programiranje je postalo brže, ali novi programi su se izvršavali 10-20 puta sporije nego programi napisani na asembleru. Početna verzija kompilatora za Fortran I objavljena je nekoliko godina kasnije – 1956. godine i imala je oko 25000 linija asemblerskog koda. Kompilirani programi izvršavali su se skoro jednako brzo kao programi ručno pisani na asembleru. Pisanje programa ubrzano je i po 40 puta. Znatno je olakšano i održavanje programa zbog bolje čitljivosti i omogućena je prenosivost između različitih računara (za koje su postojali razvijeni Fortran kompilatori). Već 1958. više od polovine svih programa pisano je na Fortran-u. Ovaj jezik se, uz velike izmene u odnosu na prvobitne verzije, i danas koristi i namenjen je, pre svega, za numerička i naučna izračunavanja.

Cobol je nastao 1959. godine zajedničkom inicijativom nekoliko vodećih kompanija i univerziteta da se napravi jezik pogodan za izradu poslovnih, finansijskih i administrativnih aplikacija. Ime Cobol potiče od *COmmon Business Oriented Language*. Mnoge aplikacije pisane u Cobol-u su i danas u upotrebi, ali se Cobol više ne koristi za razvoj novih aplikacija.

Algol je jedan od najuticajnijih programskih jezika koji je nastao i razvijan od kraja pedesetih do početka sedamdesetih godina prošlog veka. On uvodi razne bitne karakteristike modernih programskih jezika a ime mu potiče od „ALGOrithmic Language“. U njegov razvoj bili su uključeni mnogi znameniti informatičari iz Evrope i Amerike.

Pascal je jedan od naslednika jezika Algol, koji je 1970. godine dizajnirao švajcarski informatičar Niklaus Virt (nem. *Niklaus Wirth*) kao mali i efikasan jezik koji ohrabruje korišćenje strukturiranog programiranja i drugih dobrih praksi programiranja. Jezik Pascal bio je veoma popularan tokom osamdesetih

[2]Ovo važi i za druge programske jezike, čija su imena najpre pisana velikim slovima jer su nastala kao akronimi. Na primer, to važi za jezike Cobol, Algol, Basic, Lisp i Prolog.

i devedesetih godina prošlog veka, ali je ipak smatrano da je njegovo glavno polje nastava programiranja (a ne i industrijske primene). Unapređenja i modifikacije jezika Pascal dovele su do jezika Modula, Oberon i Modula-2 (koje je, osamdesetih godina prošlog veka, razvio takođe Virt), kao i do objektno-orijentisanog jezika Object Pascal.

Basic je inicijalno razvijen 1964. godine za početnike u programiranju, za koje su Fortran i Algol bili previše kompleksni. Ime mu potiče od *Beginner's All-Purpose Symbolic Instruction Code*. Jezik je bio ekstremno jednostavan – prva verzija imala je samo četrnaest naredbi. Popularnost jezika porasla je sa pojavom mikro-računara i personalnih računara. U međuvremenu je razvijeno mnogo njegovih modifikacija i proširenja, od kojih je trenutno najpopularniji Visual Basic.

C je programski jezik opšte namene koji je 1972. godine razvio Denis Riči[3] u Belovim telefonskim laboratorijama (engl. *Bell Telephone Laboratories*) u SAD. Ime C dolazi od činjenice da je jezik nastao kao naslednik jezika B. C je jezik koji je bio namenjen prevashodno pisanju sistemskog softvera i to u okviru operativnog sistema Unix. Međutim, vremenom je počeo da se koristi i za pisanje aplikativnog softvera na velikom broju drugih platformi. C je danas prisutan na širokom spektru platformi — od mikrokontrolera do superračunara. Jezik C značajno je uticao i na razvoj drugih programskih jezika.

Funkcionalni jezici. Funkcionalno programiranje je u ekspanziji. Koncepti i ideje koje su nastale u okviru funkcionalnih jezika sve više prodiru u svakodnevno programiranje i u jezike koji u osnovi pripadaju drugim programskim paradigmama. Funkcionalni jezici razmatraju programiranje kao proces izračunavanja matematičkih funkcija. Koreni funkcionalnog programiranja leže u λ-računu razvijenom 1930-tih kako bi se izučavao pojam izračunljivosti i algoritma. Mnogi funkcionalni programski jezici mogu se smatrati nadogradnjama λ-računa.

Primeri značajnijih funkcionalnih programskih jezika su Lisp, Scheme, ML, Haskell, Erlang, Elixir i Elm. Funkcionalni jezici su najčešće jezici opšte namene. Veoma su koncizni i neki programeri ih svrstavaju u deklarativnu paradigmu.

Lisp je razvio Džon Makarti (engl. *John McCarthy*) 1958. godine na univerzitetu MIT i prvi je funkcionalni programski jezik. Ime Lisp nastalo je od *LISt Processing*, jer jezik podržava listu kao osnovnu strukturu podataka. Lisp je dugo smatran jezikom veštačke inteligencije. Od Lisp-a su se dalje razvili svi savremeni funkcionalni programski jezici. Neke varijante Lispa se i danas koriste.

Haskell je zvanično predstavljen 1990. godine. Osnovna ideja je bila da se standardizuju funkcionalni programski jezici koji su do tada postojali jer ih je bio veliki broj, a nijedan od njih nije bio široko prihvaćen. Haskell se danas često koristi u akademskim krugovima, ali ima i široku primenu u industriji za rešavanje složenih problema, posebno u oblastima gde su ispravnost i pouzdanost koda ključne.

Erlang je razvijen krajem 1980-ih u Ericssonu, švedskoj telekomunikacionoj kompaniji, sa ciljem da olakša izgradnju distribuiranih, skalabilnih i visoko dostupnih sistema koji su tada bili potrebni u telekomunikacijama. Erlang je zvanično pušten u javnost 1998. godine i od tada se koristi u raznim aplikacijama, kao što su telekomunikacioni sistemi, bankarstvo, i masivne online igre. Sa pojavom interneta, ideje koje su razvijane u okviru Erlanga dobijaju na značaju u novom kontekstu i razvijaju se novi programski jezici koji kao osnovu koriste Erlang.

Elixir je nastao 2011. godine sa ciljem da kombinuje robusnost i konkurentnost Erlanga sa modernijim i pristupačnijim sintaksnim konstrukcijama. Kao i Erlang, Elixir je odličan izbor za distribuirane sisteme koji zahtevaju visoku dostupnost, paralelno izvršavanje procesa i otpornost na greške, ali nudi modernije alate i sintaksu prilagođenu savremenom razvoju softvera. Elixir je naročito popularan u razvoju veb aplikacija. I Erlang i Elixir predstavljaju spoj funkcionalnog i konkurentnog programiranja.

Elm je nastao 2012. godine i napravljen je za razvoj korisničkih interfejsa na vebu. Koristi se često u kombinaciji sa jezikom Elixir, u kojem se programira logika sistema, dok se u Elmu programira korisnički interfejs.

Clojure je nastao 2007. godine sa ciljem da obezbedi jednostavan i moćan alat za konkurentno i paralelno programiranje. Clojure se zasniva na jeziku Lisp ali ima osobinu interoperabilonsti sa Javom, tj. izvršava se na javinoj virtulenoj mašini. To omogućava lak pristup postojećim Java bibliotekama i alatima, što ga čini moćnim za korišćenje u okruženjima gde je Java već prisutna. Takođe, stekao je popularnost među programerima koji cene minimalizam i jednostavnost Lisp sintakse.

Primer 7.4 (Programski jezik Haskell). *Napisati program koji računa sumu kvadrata svih neparnih prirodnih brojeva čiji je kvadrat manji od 10000.*

[3] Dennis Ritchie (1941–2011), američki informatičar, dobitnik Tjuringove nagrade 1983. godine.

Elektronska verzija (2024)

```
sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

Logički jezici. Logička paradigma je deklartivna paradigma koja se oslanja na matematičku logiku. Osnovni predstavnik ove paradigme je programski jezik Prolog, koji se često i koristi kao sinonim za logičku paradigmu. Prolog nije jezik opšte namene već se koristi u kontekstu tradicionalne veštačke inteligencije i predstavljanja znanja, za rešavanje logičkih problema, rezonovanje u sistemima koji su zasnovani na pravilima i u integraciji podataka i znanja.

Primer 7.5 (Zagonetka: programski jezik Prolog)**.** *Postoje tri kuće u nizu, svaka je obojena različitom bojom: crvena, plava i zelena. Svaki vlasnik kuće pije različito piće: vodu, čaj i kafu. Svaki vlasnik ima drugačiju životinju za kućnog ljubimca: mačku, psa i pticu.*

Tragovi:

Osoba u crvenoj kući ima mačku. Osoba u zelenoj kući pije kafu. Osoba u plavoj kući pije čaj. Osoba u zelenoj kući nema pticu.

Koristeći tragove, rešiti zagonetku: Ko poseduje psa?

```
resenje :-
  % Tri kuće: Kuca1, Kuca2, Kuca3
  Kuce = [Kuca1, Kuca2, Kuca3],

  % Svaka kuća je predstavljena listom [Boja, Pice, Ljubimac]
  % Inicijalizujemo promenljive za boje, pića i ljubimce
  Kuca1 = [Boja1, Pice1, Ljubimac1],
  Kuca2 = [Boja2, Pice2, Ljubimac2],
  Kuca3 = [Boja3, Pice3, Ljubimac3],

  % Postoje tri moguće boje: crvena, zelena i plava
  Boje = [crvena, zelena, plava],
  % Postoje tri moguća pića: voda, čaj, kafa
  Pica = [voda, caj, kafa],
  % Postoje tri moguća ljubimca: mačka, pas, ptica
  Ljubimci = [macka, pas, ptica],

  % Svaka kuća ima različitu boju, piće i ljubimca
  permutation(Boje, [Boja1, Boja2, Boja3]),
  permutation(Pica, [Pice1, Pice2, Pice3]),
  permutation(Ljubimci, [Ljubimac1, Ljubimac2, Ljubimac3]),

  % Trag 1: Osoba u crvenoj kući ima mačku.
  member([crvena, _, macka], Kuce),

  % Trag 2: Osoba u zelenoj kući pije kafu.
  member([zelena, kafa, _], Kuce),

  % Trag 3: Osoba u plavoj kući pije čaj.
  member([plava, caj, _], Kuce),

  % Trag 4: Osoba u zelenoj kući nema pticu.
  not(member([zelena, _, ptica], Kuce)),

  % Određivanje ko ima psa i ispis rezultata
  member([Boja, Pice, pas], Kuce), % Pronađi kuću sa psom
  format('Osoba koja ima psa živi u kući koja je ~w i pije pice ~w.~n', [Boja, Pice]).
```

Rešenje zagonetke je

```
Osoba koja ima psa živi u kući koja je zelena i pije pice kafa.
```

Objektno-orijentisani jezici. *Objekti* su specijalizovane strukture podataka koje uz polja podataka sadrže i metode kojima se manipuliše tim podacima. Podaci se mogu obrađivati isključivo primenom metoda što smanjuje zavisnosti između različitih komponenata programskog koda i čini ovu paradigmu pogodnu za razvoj velikih aplikacija uz mogućnost saradnje većeg broja programera. Najčešće korišćene tehnike programiranja u objektno orijentisanom programiranju uključuju sakrivanje informacija, enkapsulaciju, apstraktne tipove podataka, modularnost, nasleđivanje i polimorfizam.

Primer 7.6 (Programski jezik Java)**.** *Osnovne karakteristike svake osobe su njeno ime i broj godina. Svaka osoba ume da se predstvi, odnosno da saopšti svoje ime i broj godina. Napisati program u kojem se kreiraju i predstavljaju dve osobe.*

```
class Osoba {
    private String ime;
    private int godine;

    // Konstruktor
    public Osoba(String ime, int godine) {
        this.ime = ime;
        this.godine = godine;
    }

    // Metod za pozdrav
    public void predstaviSe() {
        System.out.println("Zdravo, ja sam " + ime + " i imam " + godine + " godina.");
    }

    public static void main(String[] args) {
        // Kreiranje objekata
        Osoba osoba1 = new Osoba("Ana", 30);
        Osoba osoba2 = new Osoba("Marko", 25);

        // Pozivanje metoda za predstavljanje
        osoba1.predstaviSe();
        osoba2.predstaviSe();
    }
}
```

C++ je kreirao Bjern Stroustrup (danski informatičar) 1986. godine kao direktni naslednik jezika C. C++ se, u trenutku nastanka, mogao smatrati njegovim objektno-orijentisanim proširenjem. C++ je i dalje jedan od najpopularnijih jezika i koristi se za razvoj zahtevnih aplikacija, s jedne strane zbog svojih objektno-orijentisanih svojstava, a s druge zbog bliske veze sa mašinom, u duhu jezika C. U izvesnom smislu, potomkom i unapređenjem jezika C++ može se smatrati jezik C#, koji je razvijen 2000. godine.

Java je objektno-orijentisani programski jezik razvijen 1995. godine sa motivom da bude što manje zavisnosti za fazu izvršavanja i da se kompilirani Java kôd (takozvani bajtkod) može izvršavati na bilo kojoj platformi koja podržava Javu (tj. koja raspolaže Java virtuelnom mašinom) bez ponovnog kompiliranja. Sintaksa jezika Java slična je jezicima C i C++ ali ima manje operacija niskog nivoa. Java omogućava modifikacije koda u fazi izvršavanja. Java je trenutno jedan od najpopularnijih programskih jezika.

C# je nastao oko 2000. godine u kompaniji Micorosoft kao deo njihove .NET inicijative. Po svojim karakteristikama je donekle sličan programskom jeziku Java. Zamišljen kao moderni objektno-orijentisani programski jezik opšte namene koji se karakteriše relativnom jednostavnošću programiranja (poput jezika Java i ovaj jezik vrši automatsko upravljanje memorijom korišćenjem sakupljača otpadaka, pre pristupa elementima niza vrši se provera da li je indeks unutar granica niza, sve promenljive se inicijalizuju na podrazumevane vrednosti, disciplina tipova je prilično striktna i slično). Jezik se stalno obogaćuje i unapređuje. I dalje je veoma popularan izbor za programiranje aplikacija za Windows, veb, pa i za mobilne aplikacije.

Objective C i Swift su programski jezici koji se koriste za razvoj aplikacija na platformama kompanije *Apple*. Objective-C je stariji jezik koji je *Apple* koristio za razvoj aplikacija pre uvođenja jezika Swift. Razvijen je krajem 1980-ih, kao proširenje programskog jezika C, dodajući objektno orijentisane karakteristike. Iako je Objective-C bio glavni jezik za razvoj u kompaniji *Apple* dugi niz godina, njegova relativno složena i zastarela sintaksa dovela je do potrebe za modernijom alternativom. Swift je programski jezik koji je *Apple* predstavio 2014. godine kao modernu, bržu i sigurniju alternativu jeziku Objective-C. Swift je dizajniran tako da bude jednostavan za učenje, a istovremeno zadrži visoke performanse i pouzdanost. Swift je danas primarni jezik za razvoj aplikacija na *Apple* platformama, a zbog svojih modernih karakteristika brzo je postao popularan među programerima.

7.1.1.2 Savremene programske paradigme

Savremeni programski jezici su multiparadigmatski, odnosno u sebi sadrže više različitih stilova programiranja. Savremene programske paradigme uključuju *skript*, *komponentno*, *generičko*, *konkurentno* i *vizuelno* programiranje, kao i paradigme *upitnih jezika* i *programiranja ograničenja*.

Skript programiranje je oblik programiranja koji se koristi za pisanje kratkih, jednostavnih programa ili „skripti“ koje automatizuju zadatke ili upravljaju funkcijama u većim softverskim sistemima. Skript programiranje je vrlo korisno za zadatke u kojima je važna brzina razvoja, fleksibilnost i jednostavnost, čineći ga izuzetno popularnim u raznim oblastima IT-a. U okviru samog skript programiranja, izdvajaju se skript jezici koji se koriste u domenu veb programiranja, skript jezici opšte namene, skript jezici za procesiranje teksta, komandni jezici i jezici sa specifičnim domenom, npr za matematiku i statistiku.

Perl je nastao sredinom osamdesetih godina prošlog veka. Jedna od njegovih ključnih karakteristika su moćne funkcije za obradu teksta, a koristio se u radu sa tekstualnim podacima, sa bazama podataka, u mrežnom programiranju i slično, ali i kao skript jezik (jezik za izvršno okruženje na kojem se zadaje automatizovano izršavanje zadataka) za Linux sisteme.

Python je multiparadigmatski jezik jednostavne i izražajne sintakse koji se interpretira i čija prva verzija je objavljena 1991. godine. Ima elemente objektno-orijentisanih, imperativnih, funkcionalnih jezika itd. Raspolaže ugrađenim strukturama podataka visokog nivoa. Pošto se programi interpretiraju, osnovni ciklus razvoja programa (pisanje-testiranje-debagovanje) odvija se izuzetno brzo. Pajton je najpopularniji jezik u oblastima kao što su istraživanje podataka i mašinsko učenje, ali se koristi i za veb-aplikacije, za mobilne aplikacije, kao i za ugrađene sisteme. U nastavi je Pajton na mnogim mestima kao prvi programski jezik zamenio Pascal, Javu i C.

PHP je nastao 1994. godine kao internet jezik koji može biti ugrađen u HTML kôd. Ime PHP potiče od „Hypertext Preprocessor“. PHP kôd se izvršava na serveru i dinamički generiše HTML sadržaj koji se šalje i prikazuje klijentu. Više od polovine svih veb sajtova kao jezik na strani servera koriste PHP u nekom obliku.

JavaScript je nastao 1995. godine i obično se koristi za programe koji se izvršavaju na strani klijenta (na primer, u okviru pregledača veba). JavaScript omogućava da se sadržaj veb strana menja interaktivno, na primer, u zavisnosti od akcija korisnika. JavaScript danas koristi većina veb strana. JavaScript i PHP mogu da se koriste skladno – prvi na strani klijenta, a drugi na strani servera.

ASP.NET je nastao kreiran 2002. godine, kao skript jezik za veb aplikacije i dinamičko kreiranje veb sadržaja kompanije Microsoft. Ima dosta sličnosti sa jezikom PHP, ali može da se izvršava samo na Windows serverima.

Lua je jezik opšte namene koji je nastao 1993. godine u Brazilu. Jezik Lua je poznat po jednostavnoj sintaksi, efikasnosti i širokoj primeni, naročito u razvoju video igara. Lua podržava proceduralno, objektno orijentisano i funkcionalno programiranje, kao i programiranje vođeno podacima.

Ruby je jezik opšte namene koji se pojavio 1995. godine, u Japanu. Otovorenog je koda, sa fokusom na jednostavnost i produktivnost. Osnovna ideja dizajna jezika je da se adresiraju ljudske potrebe (a ne potrebe računara), odnosno da programiranje učini programere zadovoljnim, produktivnim i srećnim.

Bash je nastao 1987. godine u okviru GNU projekta. To je skriptni jezik za Unix i Unix-olike operativne sisteme, kao što su Linux i macOS. Bash omogućava korisnicima da izvršavaju komande direktno u komandnoj liniji. Koristi se za pisanje skripti koje mogu automatizovati razne zadatke, kao što su instalacija softvera, upravljanje datotekama i izvođenje sistemskih operacija. Bash se često korsiti za administraciju sistema i upravljanje serverima. Različite distribucije Linux-a dolaze sa Bash-om kao podrazumevanim jezikom komandne linije.

Elektronska verzija (2024)

R i S su skrpit jezici specifične namene razvijeni za statističku analizu, vizualizaciju podataka i naučno istraživanje. Jezik S je razvijen 1976. godine u Bell Labs-u i komercijalni je jezik koji je poslužio kao inspiracija za kreiranje jezika R. R je objavljen kao otvoreni softver 1995. godine. R je veoma popularan u akademskim i istraživačkim krugovima, kao i u industrijama koje se bave analizom podataka.

Primer 7.7. *Učenje novog programskog jezika obično započinje razumevanjem programa koji štampa poruku* Hello, world!. *U nastavku su dati primeri u različitim skript jezicima. Možemo da zaključimo kako su svi ovi primeri veoma slični.*

Programski jezik Perl

```
print "Hello, World!\n";
```

Programski jezik Python

```
print("Hello, World!")
```

Programski jezik PHP

```
<?php
echo "Hello, World!";
```

Programski jezik JavaScript

```
console.log("Hello, World!");
```

Programski jezik Lua

```
print("Hello, World!")
```

Programski jezik Ruby

```
puts "Hello, World!"
```

Programski jezik Bash

```
echo "Hello, World!"
```

Konkurentno programiranje je pristup pisanju koda koji omogućava izvršavanje više zadataka u istom vremenskom intervalu (bilo da je to istovremeno, onda kada postoji odgovarajuća harverska podrška na raspolaganju, ili samo vremenski ispreplitano, onda kada je za izvršavanje na raspolaganju samo jedan procesor). Konkurentno programiranje pomaže u optimizaciji korišćenja resursa i povećanju performansi aplikacija. Ovaj pristup je ključan u razvoju modernih softverskih sistema, posebno u aplikacijama koje obrađuju velike količine podataka, zahtevaju visoku dostupnost ili performanse.

Iako je konkurentnost razvijana još od jezika Algol, velika popularnost i potreba za konkurentnim programiranjem nastaje razvojem višeprocesorskih mašina i razvojem računarskih mreža. Podrška za konkurentno programiranje postoji dostupna gotovo u svim jezicima opšte namene, ali postoje i jezici koji seu razvijeni baš sa ciljeme da se olakša i unapredi ovaj stil programiranja.

Go (poznat i kao *Golang*) je programski jezik koji su razvili inženjeri kompanije Google 2007. godine, sa ciljem da stvore jezik koji kombinuje performanse i efikasnost jezika niskog nivoa poput C-a sa jednostavnošću i brzinom razvoja koje nude moderni jezici. Prva stabilna verzija objavljena je 2012.

Elektronska verzija (2024)

godine, a od tada je Go postao veoma popularan, naročito za razvoj servera, mrežnog softvera i za distribuiranu obradu podataka.

Rust Rust je programski jezik koji se ističe obezbeđivanjem sigurnosti u radu sa memorijom, performansama i podrškom za razvoj konkurentnih aplikacija. Razvila ga je kompanija Mozilla 2010. godine, a danas ga razvija i održava Rust Foundation. Rust je osmišljen kao bezbedna i brza alternativa jezicima poput C i C++, sa posebnim fokusom na otklanjanje grešaka koje nastaju zbog nekontrolisanog upravljanja memorijom. Rust je brzo postao popularan među programerima koji rade na visokoperformantnim aplikacijama ili softveru gde su stabilnost i sigurnost kritični. Koristi se za razne projekte, uključujući web servere, igre, komandne linijske alate i komponente sistema kao što je Linux kernel.

Kotlin je moderan programski jezik koji se izvršava na Java virtuelnoj mašini i koristi se primarno za razvoj Android aplikacija. Razvila ga je kompanija JetBrains 2011. godine, dok je prva stabilna verzija objavljena 2016. godine. Google je 2017. godine zvanično podržao Kotlin kao osnovni jezik za razvoj Android aplikacija, što je značajno doprinelo njegovoj popularnosti. Kotlin je postao popularan izbor za Android razvoj zbog svoje čitljivosti, smanjenja količine koda i ugrađene sigurnosti.

Primer 7.8 (Konkurentno sabiranje elemenata niza u jeziku Go)**.** *Izračunavanje zbira elemenata u nizu može se podeliti tako da se izvršava konkurentno za različite delove niza, i zatim se dobijeni zbirovi podnizova saberu. Primer programa koji izračunavanje zbira elemenata u nizu izvršava podelom niza na dva dela dat je u nastavku.*

```
package main

import (
    "fmt"
    "sync"
)

// Funkcija za sabiranje elemenata dela niza
func sumPart(arr []int, result *int, wg *sync.WaitGroup) {
    defer wg.Done()
    partialSum := 0
    for _, v := range arr {
        partialSum += v
    }
    *result = partialSum
}

func main() {
    arr := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    var wg sync.WaitGroup

    // Promenljive za čuvanje rezultata sabiranja
    var sum1, sum2 int

// Duzina jednog podniza
    mid := len(arr) / 2

    // Delimo niz na dva dela i pokrećemo sabiranje u dve gorutine
    wg.Add(2)
    go sumPart(arr[:mid], &sum1, &wg) // sabira prvu polovinu niza
    go sumPart(arr[mid:], &sum2, &wg) // sabira drugu polovinu niza

    // Čekamo da obe gorutine završe
    wg.Wait()

    // Konačna suma
    totalSum := sum1 + sum2
```

```
    fmt.Printf("Suma elemenata niza je: %d\n", totalSum)
}
```

Komponentno programiranje je pristup razvoju softvera koji se fokusira na izgradnju aplikacija korišćenjem nezavisnih, ponovo upotrebljivih komponenti. Ove komponente mogu biti različitih vrsta, uključujući biblioteke, module ili čak mikroservise, a sve zajedno omogućavaju brži i efikasniji razvoj aplikacija. Komponentno programiranje najčešće karakteriše pristup *prevuci i postavi* (eng. *drag and drop*), tj pristup da se kôd ne kuca u potpunosti u tekstulano editoru, veće da se najveći deo koda automatski generiše prevlačenjem i povezivanjem odgovarajućih kompnenti. Komponentno programiranje je slično objektno-orjentisanom programiranju, ali su komponente veće programske celine u odnosu na klase. Komponentno programiranje se može ostvariti u različitim programskim jezicima opšte namene (npr. C, C++, Java, C#, Swift), kroz razvojna okruženja i biblioteke.

Primer 7.9. *Grafički korisnički interfejsi se najčešće prave komponentnim stilom programiranja. Jedna od početnih komponenti je prozor na koji se onda dodaju različite druge komponente, na primer dugmići, tekstualna polja, tekst i slično.*

Vizuelno programiranje koristi grafičko okruženje za kreiranje programa, umesto pisanja koda u tekstualnom obliku. U vizuelnom programiranju, programeri koriste vizuelne elemente kao što su blokovi, ikone, linije i dijagrami za pravljenje programa. Ovaj pristup je često intuitivniji i pristupačniji, posebno za početnike, jer omogućava manipulaciju objekata i logike programa na grafički način. Primeri vizuelnog programiranja za početnike su okruženja *Scratch* i *Blockly*, dok se alati *Node-RED*, *LabVIEW* i *Unreal Engine Blueprints* koriste u industrijskom okruženju.

Primer 7.10. *Razvojno okruženje za* Scratch *je zajedno sa različitim tutorijalima besplatno dostupno na internetu https://scratch.mit.edu/projects/editor/. Naredna slika prikazuje razvojno okruženje i program koji omogućava pokretanje mačke (u gornjem desnom uglu) u krugu za 360 stepeni u smeru kazaljke na satu.*

programski_jezici_Milena/slike/scratch.png

Generičko programiranje je paradigma koja omogućava pisanje koda koji može da se upotrebljava sa različitim tipovima podataka. Osnovna ideja je da se algoritmi i strukture podataka mogu definisati na uopšten način, tako da rade sa bilo kojim tipom podataka, dok god taj tip zadovoljava određene zahteve. Ovakv pristup programiranju obezbeđuje fleksibilnost i ponovnu upotrebljivost napisanog koda. Primeri programskih jezika koji podržavaju različite vidove generičkog programiranja su C++, Java i C#.

Primer 7.11 (Generička funkcija za pronalaženje maksimumam u jeziku C++)**.** *Umesto konkretnog tipa, koristi se tipska promenljiva T, koja omogućava apstrahovanje tipa iz funkcije i upotrebu funkcije na svim tipovima koj imaju definisan operator poređenja >. Na primer, ovako definisana funkcija može se na isti način koristiti za cele brojeve, realne brojeve, i za karaktere.*

```
#include <iostream>
using namespace std;

// Generička funkcija koja računa veću od dve promenljive
template <typename T>
T maksimum(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << maksimum(110, 210) << endl;     // Celi brojevi
    cout << maksimum(5.7, 5.3) << endl;     // Realni brojevi
    cout << maksimum('g', 'e') << endl;     // Karakteri
    return 0;
}
```

Elektronska verzija (2024)

Upitni jezici pripadaju deklarativnoj paradigmi. To su specijalizovani programski jezici koji se koriste za interakciju sa bazama podataka. Oni omogućavaju korisnicima da efikasno pretražuju, manipulišu i upravljaju podacima u različitim vrstama baza podataka. Najpoznatiji upitni jezik je SQL, koji se koristi za rad sa *relacionim* bazama podataka. Osim SQL-a, postoje i drugi upitni jezici, kao što su SPARQL za RDF podatke i XQuery za XML podatke.

SQL (*Structured Query Language*) je standardizovani jezik za rad sa relacionim bazama podataka. Razvijen 1970-ih, SQL je postao osnovni alat za rad s bazama podataka, omogućavajući definisanje, manipulaciju i kontrolu podataka na jednostavan i efikasan način. SQL se koristi za poslovnu analitiku, razvoj aplikacija i rad s velikim količinama podataka, jer omogućava složene upite, agregacije i analize podataka sa visokom efikasnošću.

SPARQL (eng. *SPARQL Protocol and RDF Query Language*) je jezik za upit nad RDF (eng. *Resource Description Framework*) podacima, koji su osnova za rad sa semantičkim webom i povezanim podacima (eng.*linked data*). SPARQL je razvio W3C (eng. *World Wide Web Consortium*) kao standardizovani način pristupa i pretraživanja informacija u RDF formatima. SPARQL je ključan alat u domenima gde se radi sa složenim i međusobno povezanim podacima, kao što su bioinformatika, bibliotečke nauke, istraživačke baze podataka, i opšta analitika podataka.

XQuery (eng. *XML Query Language*) je upitni jezik dizajniran za pretraživanje i manipulaciju XML podacima. Omogućava korisnicima da efikasno pretražuju i izvode informacije iz XML dokumenata, kao i da transformišu te podatke. XQuery je standardizovan od strane W3C.

Primer 7.12. *Naredni SQL upit iz baze podataka koja sadrzi tabelu* `Osobe` *izvlači sve informacije o svim osobama starijim od 24 godine.*

```
SELECT * FROM Osobe WHERE Godine > 24;
```

Programiranje ograničenja je deklarativna paradigma u okviru koje je posao programera da detaljno opiše uslove u sistemu za koji se traži rešenje. Koristi se u domenu optimizacija i rešavanja kombinatornih problema. Kao veoma praktična i potrebna tehnika programiranja, veliki broj programskih jezika opšte namene ima biblioteke koje pružaju podršku za programiranje u ovom stilu (jezici C, C++, Java, Python, C# i mnogi drugi).

Primer 7.13. *Za primer 7.2 koji ilustruje programiranje ograničenja u programskom jeziku Prolog, u nastavku je dat odgovarajući program u programskom jeziku Python. Možemo da primetimo da se ova dva programa ne razlikuju suštinski.*

```
from constraint import Problem, AllDifferentConstraint

# Definisi funkciju ogranicenja kriptoaritmetike
def equation_constraint(o, n, e, t, w):
    # ONE + ONE should equal TWO
    return (o * 100 + n * 10 + e) * 2 == t * 100 + w * 10 + o

# Definisi funkciju ogranicenja da je vrednost razlicita od nule
def non_zero_constraint(value):
    return value != 0

# Napravi instacu problema
problem = Problem()

# Definisi promenljive O, N, E, T, W kao cifre od 0 do 9
problem.addVariables("ONETW", range(10))

# Dodaj ogranicenje da O i T ne mogu biti 0 (jer su brojevi TWO i ONE trocifreni)
problem.addConstraint(non_zero_constraint, ["OT"])

# Sve cifre moraju biti razlicite
problem.addConstraint(AllDifferentConstraint(), "ONEWT")
```

```
# Jednakost kroptoaritmetike: ONE + ONE = TWO
problem.addConstraint(equation_constraint, "ONEWT")

# Izracunaj i odstampaj resenja
solutions = problem.getSolutions()
if solutions:
    for solution in solutions:
        print(f"Solution found: O={solution['O']}, N={solution['N']}, E={solution['E']}, "
              f"T={solution['T']}, W={solution['W']}")
else:
    print("Resenje nije nadjeno.")
```

7.1.1.3 Jezici za obeležavanje podataka/teksta

Jezici za obeležavanje podataka ili teksta, kao što su to, na primer, HTML/CSS, XML i LaTeX, iako veoma bitni u kontekstu programiranja, nisu programski jezici i kao takvi ne navodimo ih u klasifikaciji programskih jezika. Ovim jezicima definiše se struktura teksta ili podataka, ne pišu se programi koji definišu izvršavanje, što je ključno da bi se neki jezik smatrao programskim jezikom. Ovi jezici služe da se podaci ili tekst sistematično označe i da onda, na osnovu tih oznaka, specijalizovani programi mogu da obrađuju te podatke ili da prikažu tekst na odgovarajući način.

Primer 7.14. *HTML definiše strukturu veb stranice i određuje naslove, podnaslove paragrafe i slično. Veb pregledač na osnovu tih informacija (i eventulano dodatnih stilskih informacija kroz CSS) određuje vizuelni prikaz veb stranice. Na primer, naredni tekst je obeležen:*

```
<h1>Ljubičica</h1>
<p>Ljubičice su uglavnom višegodišnje zeljaste biljke. One imaju cetove
zanimljivih  arhitektura i boja, usled čega se mnoge vrste uzgajaju
kao ukrasne.</p>
```

Oznaka <h1> *obeležava početak teksta naslova, dok oznaka* </h1> *obeležava kraj teksta koji pripada tom naslovu. Slično, oznaka* <p> *obeležava početak teksta paragrafa, dok oznaka* </p> *obeležava kraj teksta koji pripada tom paragrafu. Različiti veb pregledači mogu da prikažu ovaj tekst na različite načine.*

Pitanja i zadaci za vežbu

Pitanje 7.1. *Koja su mane mašinski zavisnih programskih jezika?*

Pitanje 7.2. *Koje su prednosti viših programskih jezika?*

Pitanje 7.3. *Zašto postoji veliki broj programskih jezika?*

Pitanje 7.4. *Čemu služe programskih prevodioci?*

Pitanje 7.5. *Šta meri TIOBE indeks?*

Pitanje 7.6. *Koja je osnovna razlika između proceduralnog i deklarativnog programiranja? Navesti primere jezika koji pripadaju proceduralnoj i jezika koji pripadaju deklarativnoj paradigmi.*

Pitanje 7.7. *Koje su četiri osnovne programske paradigme? Navesti osnovne karakteristike svake od njih.*

Pitanje 7.8. *Koja četiri značajna programska jezika su prva istorijski nastala? Kojim paragimama pripadaju ti jezici?*

Pitanje 7.9. *Koje su osnovne karakteristike imperativne paradigme? Navesti bar četiri jezika koji pripadaju imperativnoj paradigmi.*

Pitanje 7.10. *Koje su osnovne karakteristike funkcionalne paradigme? Navesti bar četiri jezika koji pripadaju funkcionalnoj paradigmi.*

Pitanje 7.11. *Koje su osnovne karakteristike objektno-orjentisane paradigme? Navesti bar četiri jezika koji pripadaju objektno-orjentisanoj paradigmi.*

Pitanje 7.12. *Koje su osnovne karakteristike logičke paradigme? Koji jezik se koristi kao sinonim za logičko programiranje?*

Pitanje 7.13. *Koje su osnovne karakteristike i ciljevi skript programiranja? Koji poddomeni skript programiranja postoje? Koji jezici podržavaju ovu paradigmu?*

Pitanje 7.14. *Koje su osnovne karakteristike i ciljevi konkurentnog programiranja? Kada je nastalo konkurentno programiranje i kako je dobilo na značaju? Koji jezici podržavaju ovu paradigmu? Koji funkcionalni programski jezici podržavaju konkurentno programiranje?*

Pitanje 7.15. *Koje su osnovne karakteristike i domeni upotrebe komponentnog programiranja? Kojoj osnovnoj paradigmi je komponentno programiranje najsličnije i zašto?*

Pitanje 7.16. *Koje su osnovne karakteristike i ciljevi vizuelnog programiranja? Koja okruženja podržavaju ovu paradigmu?*

Pitanje 7.17. *Koje su osnovne karakteristike i ciljevi generičkog programiranja? Koji jezici podržavaju ovu paradigmu?*

Pitanje 7.18. *Koje su osnovne karakteristike i domeni upotrebe paradigme upitnih jezika? Koji jezici podržavaju ovu paradigmu?*

Pitanje 7.19. *Koje su osnovne karakteristike i domeni upotrebe paradigme programiranja ograničenja? Koji jezici podržavaju ovu paradigmu?*

Pitanje 7.20. *Zašto jezici za obeležavanje teksta ne pripadaju programskim paradigmama? Koji su najbitniji jezici za obeležavanje teksta?*

7.2 Leksika, sintaksa, semantika i pragmatika 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. Pragmatika programskih jezika se bavi problematikom upotrebe jezika u praktičnim situacijama.

7.2.1 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.

Primer 7.15. *Leksika jezika* UR *mašina je veoma jednostavna i sadrži samo četiri rezervisane reči (Z, S, J, T) i brojevne konstante.*

Primer 7.16. *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 *identifikator*
$;$ *interpunkcija*

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

7.2.2 Sintaksa

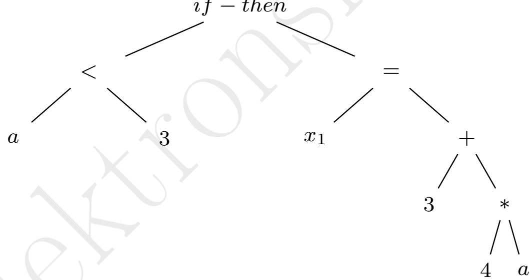
Sintaksa prirodnih jezika definiše načine na koji pojedinačne reči mogu da kreiraju ispravne rečenice jezika. Slično je i sa programskim jezicima, gde se umesto ispravnih rečenica razmatraju ispravni programi.

Primer 7.17. *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* $T(\text{broj}, \text{broj})$. *Na primer, jedan sintaksno ispravan program je*

```
1. Z(0)
2. S(0)
```

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 često predstavlja u obliku stabla.

Primer 7.18. *Fragment koda iz primera 7.16 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*.

7.2.2.1 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.

Primer 7.19. *Za sintaksno ispravan* UR *program iz primera 7.17 može se pridružiti sledeće značenje: „Postavi vrednost nultog registra na nulu i zatim ga uvećaj za jedan.“*

Primer 7.20. *Za kod iz primera 7.16 jezika C++ može se 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`*“.*

Elektronska verzija (2024)

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. Sintaksno ispravni programi ne moraju da imaju ispravno značenje.

Primer 7.21. *Za naredni C++ kôd ne može se definisati ispravno značenje i zbog toga se program ne može prevesti u izvršivi kôd.*

```
int x = 0;
int x = 5;
```

Iako su oba reda sintaksno ispravna, kôd je semantički neispravan jer se dva puta deklariše promenjiva sa istim imenom.

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.

Primer 7.22. *Za naredni C++ kôd može se definisati statičko značenje pa u skladu sa time program se može prevesti u izvršivi kôd. Međutim, za njega se ne može definisati dinamičko značenje, pa u fazi izvršavanja dolazi do greške.*

```
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[4]. U ostalim slučajevima, semantika programskog jezika se opisuje neformalno, opisima zadatim korišćenjem prirodnog jezika. Čest je slučaj da neki aspekti semantike ostaju nedefinisani standardom jezika i prepušta se implementacijama programskih prevodilaca da samostalno odrede potpunu semantiku.

Primer 7.23. *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`.

7.2.3 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 kontekstu programera, pragmatika programskog jezika odnosi se i na praktičnu upotrebu jezika u stvarnim situacijama programiranja, tj. kako se jezik koristi u praksi za postizanje specifičnih ciljeva, kao što su efikasnost, čitljivost i održivost koda. Pragmatika se bavi i pitanjem kako i zašto programeri koriste određene elemente jezika u različitim kontekstima. U nastavku su dati neki primeri pragmatičnih izbora u programskom jeziku C. Biraju se različiti elementi jezika u zavisnosti od specifičnih potreba i problema koje rešavaju.

Primer 7.24. *U programskom jeziku C++ postoje različite vrste petlji. Pragmatika programskog jezika razmatra kako programer odlučuje da li će za neki program koristiti* `for` *ili* `while` *petlju. Na primer,* `for` *petlja se upotrebljava kada za ponavljanje postoji očekivani broj izvršavanja, dok se upotreba* `while` *petlje očekuje onda kada postoji specifičan uslov prekida ponavaljanja. Na primer, za program koji štampa prvih n prirodnih brojeva upotrebljava se* `for` *petlja*

[4] 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.

Elektronska verzija (2024)

```
for(i = 0; i < n; i++) {
    printf("i\n");
}
```

dok se za program koji sabira brojeve koji se unose sa standardnog ulaza sve dok se ne unese broj 0 upotrebljava `while` *petlja*

```
while(broj != 0) {
    scanf("%d", &broj);
    suma += broj;
}
```

Ove odluke utču na bolju čitljivost koda.

Primer 7.25. *Programeri biraju da li će koristiti niz naredbi* `if-else` *ili naredbu* `switch/case`*. Kada postoji mnogo različitih mogućih vrednosti neke promenljive, naredba* `switch/case` *se koristi radi bolje organizacije i čitljivosti koda. S druge strane, ako imamo neki binarni uslov, prirodno je koristiti naredbu* `if-else`*. Na primer, za provera da li je broj paran koristi se naredba* `if-else`*:*

```
if (broj % 2 == 0) {
    printf("Broj je paran!");
} else {
    printf("Broj je neparan!")
}
```

S druge strane, za štampanje ostatka pri deljenju sa brojem 5, preglednije je koristiti naredbu `switch/case`

```
switch (broj % 5) {
    case 1: printf("Jedan\n"); break;
    case 2: printf("Dva\n"); break;
    case 3: printf("Tri\n"); break;
    case 4: printf("Cetiri\n"); break;
    default: printf("Broj je deljiv sa 5\n");
}
```

Primer 7.26. *Programeri koriste funkcije da bi smanjili dupliranje koda i olakšali održavanje. Funkcije omogućavaju lakšu organizaciju koda, ponovnu upotrebu i modularnost.*

Primer 7.27. *Programeri nekada koriste rekurziju (funkcija koja poziva samu sebe) umesto iteracije. Rekurzija je često prirodniji izbor za probleme koji se mogu podeliti na manje podprobleme, kao što su pretraga stabala ili rešavanje problema poput rekurzivno definisanih matematičkih funkcija. Iteracija se bira kada se traži efikasnije i manje memorijski zahtevno rešenje, jer rekurzija može izazvati prekomernu upotrebu memorije.*

```
int faktorijelRekurzivno(int n) {
    if (n == 0) {
        return 1;  // Bazni slučaj: faktorijel od 0 je 1
    } else {
        return n * faktorijelRekurzivno(n - 1);  // Rekurzivni poziv
    }
}
```

```
int faktorijelIterativno(int n) {
    int rezultat = 1;
    for (int i = 1; i <= n; i++) {
        rezultat *= i;  // Množi rezultat sa svakim sledećim brojem
    }
    return rezultat;
}
```

Pragmatika programskih jezika može se razmatrati i u širem kontekstu u kojem obuhvata najpre izbor jezika i tehnologije koji će se koristiti za rešavanje nekog konkretnog problema, a zatim i razumevanje koncepata dizajna programskih jezika. Pragmatika jezika se, između ostalog, bavi i sledećim pitanjima dizajna programskih jezika: izbor i karakteristike tipova podataka koje jezik obezbeđuje kao i načinima da se kreiraju kompleksni tipovi podataka, načini na koji se može ostvariti kontrola toka izvršavanja u programima, upotreba i karakteristike potprograma, modularnost i načini omogućavanja podele koda i modularnog programiranja, pristup i upravljanje memorijom i slično.

Pitanja i zadaci za vežbu

Pitanje 7.21.

Pitanje 7.22.

Pitanje 7.23. *Koje su najznačajnije programske paradigme? U koju paradigmu spada programski jezik C? Šta su to proceduralni, a šta su to deklarativni jezici?*

Pitanje 7.24.

Pitanje 7.25.

Pitanje 7.26.

Pitanje 7.27.

Pitanje 7.28.

Pitanje 7.29.

Pitanje 7.30.

Pitanje 7.31.

Pitanje 7.32.

Pitanje 7.33.

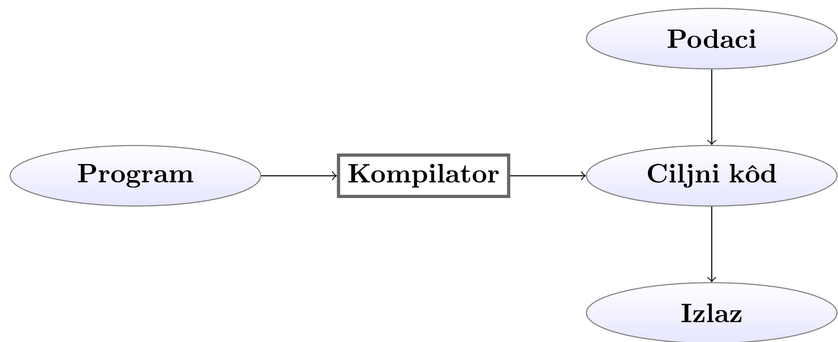
7.3 Jezički procesori

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.

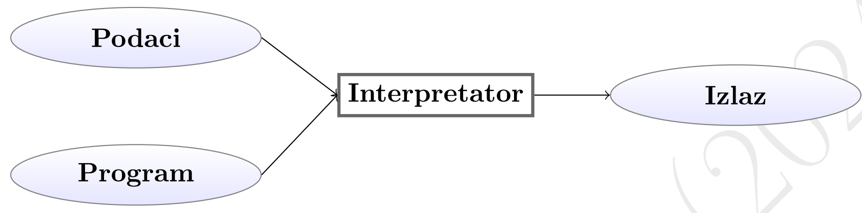
U zavisnosti od toga da li se ceo program analizira i transformiše u mašinski kôd pre nego što može da se izvrši, ili se analiza i izvršavanje programa obavljaju naizmenično deo po deo programa (na primer, naredba po naredba), jezički procesori se dele u dve grupe: *kompilatore* i *interpretatore*. Pored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

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 pri čemu samo prevođenje zahteva značajne, pre svega vremenske resurse. S druge strane, kompilirani

[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.



Slika 7.1: Kompilacija i izvršavanje



Slika 7.2: Interpretacija i izvršavanje

Elektronska verzija (2024)

programi su obično veoma efikasni, značajno efikasniji nego kada se program izvršava ne neki od narednih načina.

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.

Kombinacija kompilacije i interpretacije. 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 apstraktne virtuelne mašine), a zatim se vrši interpretacija ili kompilacija u vreme izvršavanja ovog međujezika i njegovo izvršavanje na konkretnom računaru.

Neke platforme i programski jezici zasnovani su na ideji da se programi kompiliraju u specifičan „poluprevedeni kôd" (često se naziva bajtkod, engl. *bytecode*) a zatim se taj kôd prilikom izvršavanja interpretira ili kompilira na neki specifičan način. Bajtkod se može shvatiti kao asemblerski tj. mašinski jezik neke *virtuelne mašine* koji je mnogo nižeg nivoa nego originalni program na višem programskom jeziku, ali koji za razliku od realnog asemblera ne pokriva detalje konkretne arhitekture na kojoj će se program izvršavati. Veoma popularna je i takozvana JIT kompilacija (engl. *just in time compilation*), koja podrazumeva da se bajtkod izvršava tako što se često izvršavane naredbe programa prevode u mašinske instrukcije za ciljnu mašinu, koje se onda čuvaju i direktno izvršavaju (umesto da se interpretiraju). Moguća je i takozvana AOT kompilacija (engl. *ahead of time compilation*) koja podrazumeva da se pre svog izvršavanja ceo bajtkod prevede na mašinski jezik ciljnog računara. I JIT i AOT kompilacijom se dobija mnogo brže izvršavanje programa nego kod klasičnog interpretiranja bajtkoda, a zadržavaju se prednosti koje prevođenje na bajtkod donosi. To je pre svega jednostavna prenosivost programa prevedenih na bajtkod na različite platforme — za svaku novu platformu potrebno je izgraditi samo interpreter, JIT ili AOT kompilator koji obrađuju bajtkod relativno blizak mašinskom jeziku, a to je mnogo jednostavnije nego razviti kompilator za polazni izvorni jezik visokog nivoa (prevođenje do nivoa bajtkoda može da bude izrazito složen proces, koji uključuje kako različite analize, tako i različite napredne optimizacije izvornog programa).

Među najznačajnijim jezicima koji danas koriste bajtkod su Java i C#. Java programi mogu da se kompiliraju na takozvani JAVA bajtkod, koji se onda može interpretirati ili JIT kompilirati na bilo kakvom računaru (koji ima raspoloživu takozvanu Java virtualnu mašinu — JVM). U toku je i aktivni razvoj AOT kompilatora za Javu. Postoje prevodioci i za druge programske jezike koji koriste ovaj pristup i generišu Java bajtkod (na primer, programski jezici Scala i Kotlin). C# je deo Microsoft-ove .NET platforme i on

se, kao i Visual Basic i F#, prevodi na bajtkod platforme .NET. Iako dominantno vezana za operativni sistem Windows, platforma .NET je podržana i na drugim operativnim sistemima (na primer, Mono je implementacija otvorenog koda platforme .NET koja može da se koristi i na Linux sistemima).

Naglasimo da programski jezik sâm ne određuje da li će programi na njemu biti prevođeni na jedan, drugi ili neki treći način. Zaista, za mnoge jezike postoji raspoloživo više raznorodnih sistema za prevođenje i izvršavanje. Intepretator se često 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.

Kako bi se izvršila standardizacija i olakšala izgradnja jezičkih prevodioca potrebno je precizno definisati šta su ispravne sintaksičke konstrukcije programskog jezika. Opisi na govornom, prirodnom jeziku, iako mogući, obično nisu dovoljno precizni i potrebno je korišćenje preciznijih formalizama. Ovi formalizmi se nazivaju *metajezici* dok se jezik koji se opisuje korišćenjem metajezika naziva *objektni jezik*. Metajezici obično rezervišu neke simbole kao specijalne, obično za svoj zapis koriste samo ASCII karaktere i imaju svoju posebnu sintaksu pogodnu za jednostavnu obradu na računaru. U okviru ove glave biće ukratko opisano i nekoliko metajezika.

7.3.1 Struktura kompilatora

Prevođenje (kompilacija) programskog jezika je transformisanje teksta programa na jednom računarskom jeziku u tekst programa na drugom jeziku. Obično je polazni jezik programski jezik visokog nivoa, a ciljni jezik je asembler ili mašinski jezik, ili jezik neke „virtualne mašine“. *Kompilatori* prevode čitav program na jeziku višeg nivoa u program na mašinskom ili nekom drugom ciljnom jeziku. Ukoliko je ciljni jezik mašinski jezik, onda, očigledno, kompilacija mora zavisiti od mašine na kojoj će se program izvršavati. Kako bi bilo moguće prevođenje programa u odgovarajuće programe na mašinskom jeziku nekog konkretnog računara, neophodno je precizno definisati sintaksu i semantiku programskih jezika.

Današnji kompilatori obično imaju tri ključne komponente:

Prednji sloj (eng. *front-end*) čita program zapisan na višem programskom jeziku, obrađuje ga i pohranjuje u obliku interne reprezentacije tj. međukoda. Sastoji se od sledećih komponenti (kojima u nekim slučajevima prethodi pretprocesor):

- Leksički analizator;
- Sintaksički analizator;
- Semantički analizator;
- Generator međukoda.

Srednji sloj (eng. *middle-end*) optimizije međukod i priprema ga za prevođenje na ciljni jezik. Čini ga jedna komponenta:

- Optimizator međukoda.

Iako se srednji sloj bavi samo jednim poslom, tj optimizacijom međukoda, ovaj sloj je najkompleksniji i najvažniji sloj modernih kompilatora jer od njega najviše zavisi efikasnost izvršivog koda.

Zadnji sloj (eng. *back-end*) prevodi internu reprezentaciju (međukod) u ciljni jezik (često, ali ne nužno mašinski jezik). Čini ga naredne komponente:

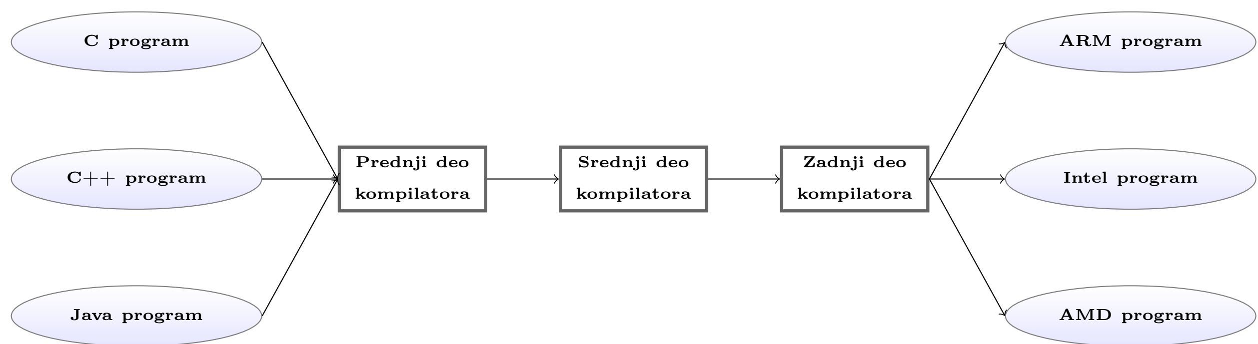
- Generator i optimizator ciljnog koda.

U preocesu finalnog prevođenja u ciljni jezik, generator i optimizator često rade naizmenično i isprepletano.

Razlaganje kompilatora na navedene komponente omogućava kombinovanje različitih prednjih slojeva (koji omogućavaju prevođenje različitih programskih jezika na međukod) i različitih zadnjih slojeva (koji su prilagođeni različitim ciljnim arhitekturama) sa jednim srednjim slojem, koji je ujedno i najkompleksniji sloj kompilatora. To je prikazano na slici 7.3.

U nastavku su ukratko opisane sve navedne faze kompilacije.

Elektronska verzija (2024)



Slika 7.3: Različiti slojevi kompilatora

7.3.2 Leksička analiza

Leksička analiza je proces izdvajanja *tokena*, osnovnih jezičkih elemenata, iz niza ulaznih karaktera (na primer, karaktera koji čine program). U analogiji sa prirodnim jezikom, leksička analiza bi odgovarala podeli rečenice na reči i određivanju vrste svake reči. Deo kompilacije koji se bavi leksičkom analizom naziva se ‘leksički analizator ili *lekser*.

Primer 7.28. *Razmotrimo naredni kôd napisan u programskom jeziku C++:*

```
if (starost >= 18)
    dopuna = 0;
else
    dopuna = 18 - starost;
```

Navedeni kôd je, zapravo, samo niz karaktera (`\n` *označava znak za novi red, a* `\t` *označava tabulator):*

```
if (starost >= 18)\n\tdopnua=0;\nelse\n\tdopuna = 18 - starost;
```

Zadatak leksera je da razloži ovaj niz na tokene kao što su identifikator, celobrojna vrednost, matematički operator, itd.

Token je sintaksička klasa, kategorija, kao što su u prirodnom jeziku kategorije, na primer, imenice, glagoli ili prilozi. Leksema je konkretan primerak, konkretna instanca jednog tokena.

Primer 7.29. *U prirodnim jezicima, jedna imenica je* pesma*, a jedan glagol je* pevati*. Slično, u programskim jezicima, za token* `IDENTIFIER`*, primeri leksema bi mogli da budu* `starost` *i* `dopuna` *iz primera 7.28, a za token* `OPERATOR` *primer lekseme mogao bi da bude* `-`*.*

Pored izdvajanja tokena, odnosno *tokenizacije*, leksički analizator može imati i druge zadatke, kao što je, na primer, eliminacija komentara (ako nema pretprocesora). Tokom leksičke analize u specijalnu tabelu koja se naziva *tabela simbola*, upisuju se prepoznati identifikatori i pridružuju im se određene relevantne informacije (na primer, vrsta i kolona u kodu gde je taj identifikator pronađen). Ova tabela dopunjuje se tokom narednih faza kompilacije (na primer, informacijama o tipovima).

Leksička analiza može da otkrije neke (jednostavne) vrste grešaka u kodu.

Primer 7.30. *U programskom jeziku C, prilikom kompilacije narednog koda, biće prijavljenje greške koje su date u komentarima. Ove greške prijavljuje leksički analizator.*

```
int a = 09;  /* error: invalid digit "9" in octal constant */
printf("Hi); /* error: missing terminating " character */
```

Leksički analizatori obično prosleđuju spisak izdvojenih leksema (i tokena kojima pripadaju) drugom programu koji nastavlja analizu teksta programa.

Elektronska verzija (2024)

Regularni izrazi kao metajezik za opis leksike. Token, kao skup svih mogućih leksema, opisuje se formalno pogodnim obrascima koji mogu da uključuju cifre, slova, specijalne simbole i slično. Za te obrasce za opisivanje tokena (tj. za opis leksike programskog jezika) kao metajezik se obično koristi formalizam *regularnih izraza (engl. regular expressions).*

Osnovne konstrukcije koje se koriste prilikom zapisa regularnih izraza su:

karakterske klase: navode se između `[` i `]` i označavaju jedan od navedenih karaktera. Na primer, klasa `[0-9]` označava cifru;

alternacija: navodi se sa `|` i označava alternativne mogućnosti. Na primer, `a|b` označava slovo `a` ili slovo `b`;

opciono pojavljivanje: navodi se sa `?`. Npr. `a?` označava da slovo `a` može, a ne mora da se javi;

ponavljanje: navodi se sa `*` i označava da se nešto javlja nula ili više puta. Npr. `a*` označava niz od nula ili više slova `a`;

pozitivno ponavljanje: navodi se sa `+` i označava da se nešto javlja jedan ili više puta. Npr. `[0-9]+` označava neprazan niz cifara.

Primer 7.31. *Razmotrimo identifikatore u programskom jeziku C. Govornim jezikom, identifikatore je moguće opisati kao*

„neprazne niske koje se sastoje od slova, cifara i podvlaka, pri čemu ne mogu da počnu cifrom“.

Ovo znači da je prvi karakter identifikatora slovo ([a-zA-Z]) ili podvlaka (_), za čim sledi nula ili više (*) slova ([a-zA-Z]), cifara ([0-9]) ili podvlaka (_). Na osnovu ovoga, moguće je napisati regularni izraz kojim se opisuju identifikatori:

```
([a-zA-Z]|_)([a-zA-Z]|[0-9]|_)*
```

Ovaj izraz je moguće zapisati jednostavnije kao:

```
[a-zA-Z_][a-zA-Z_0-9]*
```

Algoritam za izdvajanje leksema iz ulaznog teksta zasniva se na takozvanim konačnim automatima. Postoje programi koji na osnovu opisa tokena u vidu regularnih izraza generišu leksere na izabranom programskom jeziku, na primer, na jeziku C. Primer takvog programa je `lex`.

Formalizam regularnih izraza je obično dovoljan da se opišu leksički elementi programskog jezika (na primer, skup svih identifikatora, skup brojevnih literala, i slično). Međutim nije moguće konstruisati regularne izraze kojim bi se opisale neke konstrukcije koje se javljaju u programskim jezicima. Na primer, nije moguće napisati regularni izraz kojim bi se opisali svi ispravni aritmetički izrazi, tj. skup $\{a,\ a{+}a,\ a{*}a,\ a{+}a{*}a,\ a{*}(a{+}a),\ \ldots\}$.

7.3.3 Sintaksička analiza

Sintaksička analiza, poznata i kao *parsiranje*, je proces organizovanja leksema izdvojenih u fazi leksičke analize u ispravnu jezičku konstrukciju. U programskim jezicima ispravne jezičke konstrukcije mogu da uključuju dodele, petlje, uslovne naredbe itd. U analogiji sa prirodnim jezikom, sintaksička analiza odgovara proveri da li su reči u rečenici složene u skladu sa gramatičkim pravilima jezika, kao i određivanju gramatičke strukture rečenice (određivanje subjekta, predikata, itd).

Primer 7.32. *Rečenica* Jelena ide u školu. *je sintaskički ispravna, dok rečenica* Jelena u ide školu. *nije ispravna. Primetimo da obe rečenice sadrže identične reči. Slično, kôd*

```
dopuna = 18 - starost;
```

jeste ispravan, dok naredni kôd

```
18 - starost = dopuna;
```

nije ispravan. Primetimo da i u ovom slučaju, oba primera sadrže identične lekseme.

Rezultat sintaksičke analize za ispravnu ulaznu jezičku konstrukciju je *sintaksičko stablo* (ili *stablo parsiranja*). Programi koji vrše parsiranje zovu se *sintaksički analizatori* ili *parseri*.

Sintaksička analiza može da otkrije raznovrsne greške u kodu.

Elektronska verzija (2024)

Primer 7.33. *U programskom jeziku C++, prilikom kompilacije narednog koda, biće prijavljenje greške koje su date u komentarima, pri čemu će umesto* ln1 *i* ln2 *biti istaknuti odgovarajući brojevi linija u kojima se ovaj kôd nalazi. Ove greške prijavljuje sintaksički analizator.*

```
18 - starost = dopuna; /* error: lvalue required as left operand of assignment
                                   ln1 |     18 - starost = dopuna;
                                       |     ~~~^~~~~~~~~
                         */

if starost >= 18  /* error: expected ‘(’ before ‘starost’
                                   ln2 |     if starost >= 18
                                       |        ^~~~~~~
                                       |        (
                         */
```

Načini opisa sintakse programskih jezika. Za opisivanje sintakse jezika obično se koriste kontekstno-slobodne gramatike (jer izražajna snaga regularnih izraza nije dovoljno velika za to) i drugi odgovarajući meta-jezici: *BNF (Bakus-Naurova forma)*, *EBNF (proširena Bakus-Naurova forma)* i *sintaksički dijagrami*. BNF je pogodna notacija za zapis kontekstno-slobodnih gramatika, EBNF proširuje BNF operacijama regularnih izraza čime se dobija pogodniji zapis, dok sintaksički dijagrami predstavljaju slikovni metajezik za predstavljanje sintakse. Dok je BNF veoma jednostavan metajezik, precizna definicija EBNF zahteva više truda i ona je data kroz ISO 14977 standard.

Kontekstno-slobodne gramatike. Sintaksa jezika se obično opisuje gramatikama. U slučaju prirodnih jezika, gramatički opisi se obično zadaju neformalno, koristeći govorni jezik kao metajezik u okviru kojega se opisuju ispravne konstrukcije, dok se u slučaju programskih jezika, koriste znatno precizniji i formalniji opisi. Za opis sintaksičkih konstrukcija programskih jezika koriste se uglavnom kontekstno-slobodne gramatike (engl. *context free grammars*).

Kontekstno-slobodne gramatike su izražajniji formalizam od regularnih izraza. Sve što je moguće opisati regularnim izrazima, moguće je opisati i kontekstno-slobodnim gramatikama, tako da je kontekstno-slobodne gramatike moguće koristiti i za opis leksičkih konstrukcija jezika (doduše regularni izrazi obično daju koncizniji opis).

Kontekstno-slobodne gramatike su određene skupom pravila. Sa leve strane pravila nalaze se takozvani pomoćni simboli (neterminali), dok se sa desne strane nalaze niske u kojima mogu da se javljaju bilo pomoćni simboli bilo takozvani završni simboli (terminali). Svakom pomoćnom simbolu pridružena je neka sintaksička kategorija. Jedan od pomoćnih simbola se smatra istaknutim, naziva se početnim simbolom (ili aksiomom). Niska je opisana gramatikom ako ju je moguće dobiti krenuvši od početnog simbola, zamenjujući u svakom koraku pomoćne simbole desnim stranama pravila.

Primer 7.34. *Neka je gramatika zadata sledećim pravilima:*

$$
\begin{array}{lcl}
E & \rightarrow & E+T \mid T \\
T & \rightarrow & T*F \mid F \\
F & \rightarrow & (E) \mid a
\end{array}
$$

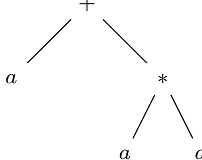
Neterminal E odgovara izrazima, neterminal T sabircima (termima), a neterminal F činiocima (faktorima).

I ova gramatika opisuje ispravne aritmetičke izraze. Na primer, niska $a+a*a$ se može izvesti na sledeći način:

$$
E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F \Rightarrow
$$
$$
a+F*F \Rightarrow a+a*F \Rightarrow a+a*a.
$$

Ovom izvođenju odgovara naredno stablo izvođenja.

Elektronska verzija (2024)



Kontekstno-slobodne gramatike čine samo jednu specijalnu vrstu formalnih gramatika. U kontekstno-slobodnim gramatikama, sa leve strane pravila uvek se nalazi tačno jedan neterminalni simbol a sa desne strane pravila može se, u opštem slučaju, nalaziti proizvoljan niz terminalnih i neterminalnih simbola.

BNF. BNF je metajezik pogodan za zapis pravila kontekstno-slobodnih gramatika. U BNF notaciji, sintaksa objektnog jezika se opisuje pomoću konačnog skupa *metalingvističkih formula (MLF)* koje direktno odgovaraju pravilima kontekstno-slobodnih gramatika.

Svaka metalingvistička formula se sastoji iz leve i desne strane razdvojene specijalnim, takozvanim univerzalnim metasimbolom (simbolom metajezika koji se koristi u svim MLF) `::=` koji se čita „po definiciji je", tj. MLF je oblika `A ::= a`, gde je `A` metalingvistička promenljiva, a `a` metalingvistički izraz. Metalingvističke promenljive su fraze prirodnog jezika u uglastim zagradama (<, >), i one predstavljaju pojmove, tj. sintaksičke kategorije objektnog jezika. Ove promenljive odgovaraju pomoćnim (neterminalnim) simbolima formalnih gramatika. U programskom jeziku, sintaksičke kategorije su, na primer, <program>, <ceo broj> i <identifikator>. Metalingvističke promenljive ne pripadaju objektnom jeziku.

S druge strane, metalingvističke konstante su simboli objektnog jezika. To su, na primer, `0`, `1`, `2`, `+`, `-`, ali i rezervisane reči programskog jezika, na primer, `if`, `then`, `begin`, `for`, itd. Dakle, uglaste zagrade razlikuju neterminalne simbole tj. imena sintaksičkih kategorija od terminalnih simbola objektnog jezika koji se navode tačno onako kakvi su u objektnom jeziku.

Metalingvistički izrazi se grade od metalingvističkih promenljivih i metalingvističkih konstanti primenom operacija konkatenacije i alternacije (`|`).

Primer 7.35. *Jezik celih brojeva u dekadnom brojnom sistemu može se opisati sledećim skupom MLF:*

```
<ceo broj>           ::= <neoznacen ceo broj> |
                         <znak broja><neoznacen ceo broj>
<neoznacen ceo broj> ::= <cifra> |
                         <neoznacen ceo broj><cifra>
<cifra>              ::= 0|1|2|3|4|5|6|7|8|9
<znak broja>         ::= +|-
```

Primer 7.36. *Gramatika aritmetičkih izraza može u BNF da se zapiše kao:*

```
<izraz>  ::= <izraz> + <term> | <term>
<term>   ::= <term> * <faktor> | <faktor>
<faktor> ::= ( <izraz> ) | <ceo broj>
```

EBNF. EBNF proširuje BNF nekim elementima regularnih izraza:

- Završni simboli objektnog jezika se navode pod navodnicima kako bi se svaki karakter, uključujući i one koji se koriste kao metasimboli u okviru EBNF, mogli koristiti kao simboli objektnog jezika.
- Pravougaone zagrade `[` i `]` ukazuju na opciono pojavljivanje.
- Vitičaste zagrade `{` i `}` ukazuju na ponavljanje.
- Svako pravilo se završava eksplicitnom oznakom kraja pravila.
- Obične male zagrade se koriste za grupisanje.

Izražajnost i metajezika BNF i metajezika EBNF jednaka je izražajnosti kontekstno-slobodnih gramatika, tj. sva tri ova formalizma mogu da opišu isti skup jezika. EBNF nudi koncizniji zapis u odnosu na BNF.

Primer 7.37. *Korišćenjem EBNF identifikatori programskog jezika C se mogu definisati sa:*

Elektronska verzija (2024)

```
<identifikator> ::= <slovo ili _> { <slovo ili _> | <cifra> }
<slovo ili _>   ::= "a"|...|"z"|"A"|...|"Z"|"_"
<cifra>         ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Primer 7.38. *Korišćenjem EBNF, jezik celih brojeva u dekadnom brojnom sistemu može se opisati sledećim skupom MLF:*

```
<ceo broj> ::= ["+"|"-"]<cifra>{<cifra>}
<cifra>    ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Primer 7.39. *Gramatika aritmetičkih izraza može u EBNF da se zapiše kao:*

```
<izraz>  ::= <term> {"+" <term>}
<term>   ::= <faktor> {"*" <faktor>}
<faktor> ::= "(" <izraz> ")" | <ceo broj>
```

Primer 7.40. *Naredba grananja* `if` *sa opcionim pojavljivanjem* `else` *grane može se opisati sa:*

```
<if_naredba>  ::= if "(" <bulovski_izraz> ")"
                     <niz_naredbi>
                  [ else
                     <niz_naredbi> ]
<niz_naredbi> ::= "{" <naredba> ";" { <naredba> ";" } "}"
```

Sintaksička analiza na osnovu zadate sintakse jezika zasniva se na takozvanim potisnim automatima. Postoje programi koji na osnovu opisa sintakse jezika (na primer, u vidu EBNF-a) generišu parsere na izabranom programskom jeziku, na primer, na jeziku C. Primer takvog programa je program `Yacc` (skraćeno od *Yet another compiler compiler*).

7.3.4 Semantička analiza

Semantička analiza je proces u kojem se proveravaju semantički uslovi i primenjuju pravila koja nije pogodno opisati sintaksičkim pravilima ni primenjivati u fazi sintaksičke analize. Semantička analiza vrši se nad sintaksičkim stablom izgrađenim tokom faze sintaksičke analize i neznatno ga modifikuje. Semantička analiza obično uključuje proveru tipova, implicitne konverzije, kao i provere koje se odnose na vidljivost promenljivih, to jest na njihov domen (jer kontekstno-slobodne gramatike nisu dovoljno izražajne da opišu pravila domena).

Semantička analiza može da otkrije raznovrsne greške u kodu. Takođe, kao rezultat semantičke analize mogu biti prijavljena i upozorenja. Upozorenja u kodu treba ozbiljno razmotriti i eliminisati.

Primer 7.41. *U programskom jeziku C++, prilikom kompilacije narednog koda, biće prijavljenja greška i upozorenje koji su dati u komentarima, pri čemu će umesto* ln1 *i* ln2 *biti istaknuti odgovarajući brojevi linija u kojima se ovaj kôd nalazi. Ova greška, odnosno upozorenje, rezultat su semantičke analize koda.*

```
int starost, starost;  /* error: redeclaration of ‘int starost’
                          ln1 |     int starost, starost;
                              |                  ~~~~~~~
                               note: ‘int starost’ previously declared here
                          ln1 |     int starost, starost;
                              |         ~~~~~~~
                       */

if (starost = 18)      /* warning: suggest parentheses around assignment used
                          as truth value [-Wparentheses]
                          ln2 |  if (starost = 18)
                              |      ~~~~~~~~~^~~~
                       */
```

Načini opisa semantike programskih jezika Dok većina savremenih jezika ima precizno i formalno definisanu sintaksu, formalna definicija semantike postoji samo za neke programske jezike. U ostalim slučajevima,

semantika programskog jezika se 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.

Primer 7.42. *Semantika koda iz primera 7.28 se neformalno može opisati na sledeći način:*

> *Ako je vrednost promenljive* starost *veća ili jednaka 18, onda vrednost promenljive* dopuna *treba da postane jednaka nuli. U suprotnom, izračunaj vrednost promenljive* dopuna *oduzimanjem od broja 18 vrednosti promenljive* starost.

Ovo je primer semantike konkretnog programa. Ova semnatika je opisana prirodnim jezikom na osnovu opisa semantike samog jezika, koja definiše značenje za naredbu if, *za naredbu dodele i za operator* -.

Formalno, semantika programskih jezika se zadaje na neki od naredna tri načina:

denotaciona sematika — programima se pridružuju matematički objekti (na primer funkcije koje preslikavaju ulaz u izlaz).

operaciona semantika — zadaju se korak-po-korak objašnjenja kako će se naredbe programa izvršavati na stvarnoj ili apstraktnoj mašini.

aksiomatska semantika — opisuje se efekat programa na tvrđenja (logičke formule) koja opisuju stanje programa. Najpoznatiji primer aksiomatske semantike je *Horova logika*.

Primer 7.43. *Semantika UR mašina definiše značenje programa operaciono, dejstvom svake instrukcije na stanje registara apstraktne UR mašine kao što je navedeno u tabeli 7.1.*

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 7.1: Tabela URM instrukcija

7.3.5 Generisanje međukoda

Generisanje međukoda je proces u okviru kojeg se izlaz iz faze semantičke analize (u vidu označenog sintaksičkog stabla) prevodi u linearnu reprezentaciju, nezavisnu od konkretnih mašina. Na primer, složeni izrazi mogu se svoditi u međukodu samo na pojedinačne operacije sa po dva argumenta, koje lako mogu da se prevedu na asembler. To svojstvo je očuvano i tokom optimizacije međukoda. Time je omogućeno da se (mnogi) izrazi u međukodu mogu čuvati kao nizovi jednostavnih četvorki koje čine operator, dva argumenta i rezultat.

Primer 7.44.

Algoritam generisanja međukoda je definisan za sve sintaksno i semantički ispravne programe. Kako je ulaz u fazu generisanja međukoda program za koji je utvrđeno da je sintaksno i semantički ispravan, nakon ove faze nije moguće da se pojave grešake i/ili upozorenja koji bi se saopštili korisniku. To važi sve do faze linkovanja, kada ponovo može da dođe do greške.

7.3.6 Optimizacija međukoda

Optimizacija međukoda je proces u okviru kojeg se na generisani međukod primenjuju raznovrsne optimizacije u cilju dobijanja efikasnijeg i kvalitetnijeg ciljnog koda, a bez promene njegovog vidljivog ponašanja tj. uz čuvanje *semantike programa*. Te optimizacije mogu da uključuju eliminisanje koda koji se ne koristi, propagaciju konstanti, promenu poretka naredbi, transformaciju petlji i slično.

Optimizacije mogu da budu lokalne i da se odnose na delove programa koji imaju jednostavnu, sekvencijalnu formu. Komplikovanije su optimizacije koje se odnose na razgranate delove programa i petlje, a najkomplikovanije optimizacije koje uključuju zavisnosti između različitih funkcija.

Primer 7.45. *Optimizacija može da naredne komande na međujeziku*

Elektronska verzija (2024)

```
x := x + 0  // može se eliminisati: mrtav kod (kod bez efekta)
x := x * 1  // može se eliminisati: mrtav kod (kod bez efekta)
x := x * 0  // može se uprostiti: vrednost izraza je 0
z := x + u  // moguća je primena propagacije vrednosti x
y := 2 * x  // može se eliminisati: mrtav kod (kod bez efekta)
y := 2 * z  // može se ubrzati: množenje brojem 2 je ekvivalentno
            // šiftovanju za jedno mesto u levo, pri čemu je
            // šiftovanje značajno efikasnija operacija od množenja
```

zameni narednim komandama (na osnovu analize koja je data kao komentar uz svaku naredbu):

```
x := 0
z := 0 + u;
y := z << 1
```

Ovako izmenjen kôd je bolji jer sadrži manji broj efikasnijih naredbi.

Optimizacija međukoda može da se izvršava sa različitim ciljevima. Tri osnovna cilja su smanjenje (i) vremena izvšavanja, (ii) veličine izvršivog programa i (iii) energije koju program troši prilikom izvršavanja. U primeru 7.45 ostvarena su sva tri cilja. Međutim, često su ovi ciljevi međusobno suprotstavljeni i to tako što ostvarenje jednog cilja negativno utiče na bar jedan od druga dva cilja. Na primer, smanjenje vremena izvršavanja često povećava veličinu izvršivog programa.

Proces optimizacije mora da uzme u obzir željene karakteristike programa. Najčešće se optimizacije vrše sa ciljem poboljšanja vremenskih performansi programa, pri čemu se vodi računa da povećanje veličine izvršivog programa bude razumno. U kontekstu uređaja sa ugrađenim računarom koji imaju ograničene resurse, može da bude važnija veličina izvršivog programa pa se u tom slučaju sprovodi drugačiji skup optimizacija. U kontekstu uređaja koji imaju ograničene izvore energije (na primer, rade na baterije: mobilni telefoni, pametni satovi i slično) veoma je bitna energetska efikasnost programa. Takođe, pored željenih karaktestika izvršivog programa koji kompilator treba da generiše, proces optimizacije mora da uzme u obzir i ukupno vreme kompilacije jer se nekada ne isplati primenjivati najkompleksnije optimizacije.

U oblasti prevođenja programskih jezika, faza optimizacije međukoda je najčešći predmet inovacija i istraživanja. Prethodne faze se već dugo sprovode na suštinski iste načine.

7.3.7 Generisanje i optimizacija ciljnog koda

Generisanje i optimizacija ciljnog koda je proces prevođenja međukoda na ciljni jezik, često jezik prilagođen nekoj konkretnoj računarskoj arhitekturi, kao što su konkretni mašinski jezici. Taj proces uključuje baratanje resursima niskog nivoa (na primer, da li će neka promenljiva da bude čuvana u registrima ili u memoriji). U okviru optimizacije koda, nizovi instrukcija na ciljnom jeziku (na primer, mašinskih instrukcija) mogu biti zamenjeni jednostavnijim fragmentima koda ili njihov poredak može biti promenjen radi kvalitetnijeg korišćenja procesora i registara.

Ilustracija sprovođenja faza kompilacije

Primer 7.46. *Sve navedene faze ilustrovane su narednim pojednostavljenim primerom.*

Izvorni kôd: *naredba dodele i složeni aritmetički izraz*

```
cur_time = start_time + cycles * 60
```

Leksička analiza: *izdvajanje reči i njihovih kategorija (tri identifikatora, dodela, sabiranje, množenje i celobrojna konstanta)*

```
ID(1) ASSIGN ID(2) ADD ID(3) MULT INT(60)
```

Sintaksička analiza: *izgradnja sintaksičkog stabla*

```
    ASSIGN
    /     \
ID(1)      ADD
```

Elektronska verzija (2024)

```
         /    \
      ID(2)   MULT
             /   \
          ID(3) INT(60)
```

Semantička analiza: *dodavanje čvora konverizije u sinstaksno stablo, usled nepoklapanja tipova*

```
    ASSIGN
    /     \
ID(1)      ADD
         /    \
      ID(2)   MULT
             /   \
          ID(3) int2real
                   |
                INT(60)
```

Generisanje međukoda: *uvođenje privremenih promenljivih kako bi svaka naredba dodele imala najviše dva argumenta*

```
temp1 = int2real(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimizacija međukoda:

Korak 1: zamena celobrojne promenljive 60 sa realnom promenljivom 60.0

```
temp1 = 60.0
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Korak 2: prenos vrednosti promenljive temp1 *u izraz u kojem* temp1 *učestvuje, prenos vrednosti promenljive* temp3 *u izraz u kojem* temp3 *učestvuje*

```
temp1 = 60.0
temp2 = id3 * 60.0
temp3 = id2 + temp2
id1 = id2 + temp2
```

Korak 3: eliminacija mrtvog koda (nepotrebne promenljive temp1 *i* temp3*)*

```
temp2 = id3 * 60.0
id1 = id2 + temp2
```

Generisanje koda na ciljnom jeziku: *odabir registara i instrukcija ciljne arhitekture*

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Pitanja i zadaci za vežbu

Pitanje 7.34.

Pitanje 7.35.

Pitanje 7.36.

Pitanje 7.37.

Elektronska verzija (2024)

Elektronska verzija (2024)

Glava 8

Proces razvoja softvera

Pod *razvojem softvera* često se ne misli samo na neposredno pisanje programa, već i na procese koji mu prethode i slede. U tom, širem smislu, razvoj softvera naziva se i *životni ciklus razvoja softvera*. Razvoj softvera razlikuje se od slučaja do slučaja, ali u nekoj formi obično ima sledeće faze i podfaze:

Planiranje: Ova faza obuhvata prikupljanje i analizu zahteva od naručioca softvera, razrešavanje nepotpunih, višesmislenih ili kontradiktornih zahteva i kreiranje precizne specifikacije problema i dizajna softverskog rešenja. Podfaze ove faze, opisane u poglavlju 8.1, su:

- Analiza i specifikovanje problema;
- Modelovanje rešenja;
- Dizajn softverskog rešenja.

Realizacija: Ova faza obuhvata implementiranje dizajniranog softverskog rešenja u nekom konkretnom programskom jeziku. Implementacija treba da sledi opšte preporuke, kao i preporuke specifične za realizatora ili za konkretan projekat. Analizom efikasnosti i ispravnosti proverava se pouzdanost i upotrebljivost softverskog proizvoda, a za naručioca se priprema i dokumentacija. Podfaze ove faze su:

- Implementiranje (kodiranje, pisanje programa) (o nekim aspektima ove podfaze govori glava **??**);
- Evaluacija – analiza ispravnosti i analiza efikasnosti (o nekim aspektima ovih podfaza govore redom glava **??** i glava **??**);
- Izrada dokumentacije (obično korisničke dokumentacije – koja opisuje korišćenje programa i tehničke dokumentacije — koja opisuje izvorni kôd);

Eksploatacija: Ova faza počinje nakon što je ispravnost softvera adekvatno proverena i nakon što je softver odobren za upotrebu. Puštanje u rad uključuje instaliranje, podešavanja u skladu sa specifičnim potrebama i zahtevima korisnika, ali i testiranje u realnom okuženju i sveukupnu evaluaciju sistema u stvarnim uslovima korišćenja. Organizuje se obuka za osnovne i napredne korisnike i obezbeđuje održavanje kroz koje se ispravljaju greške ili dodaju nove manje funkcionalnosti. U održavanje se obično uloži više od tri četvrtine ukupnog rada u čitavom životnom ciklusu softvera. Podfaze ove faze su:

- Obuka i tehnička podrška;
- Puštanje u rad;
- Održavanje.

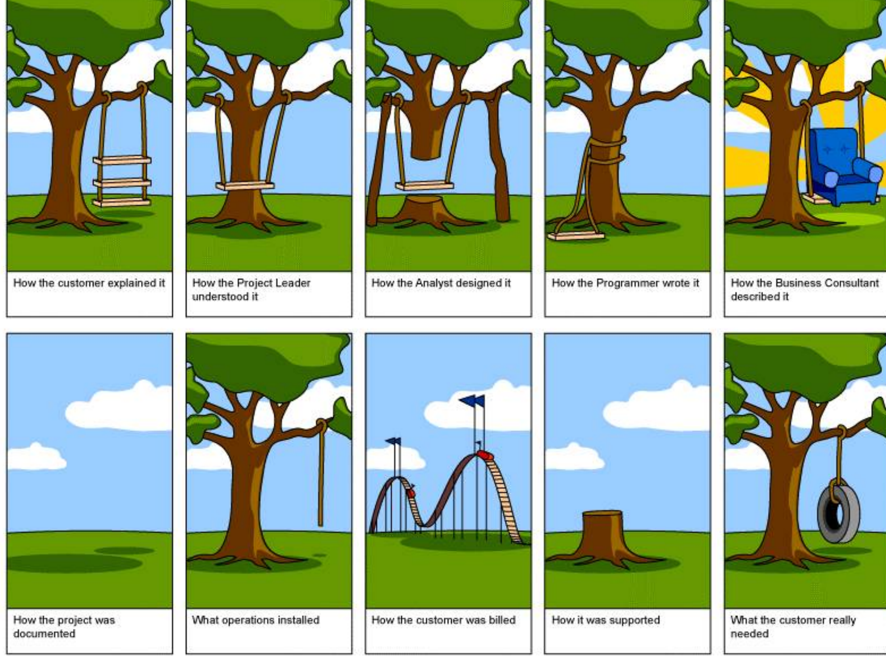
Postoje međunarodni standardi, kao što su ISO/IEC 12207 i ISO/IEC 15504, koji opisuju životni ciklus softvera kroz precizno opisane postupke izbora, implementacije i nadgledanja razvoja softvera. Kvalitet razvijenog softvera često se ocenjuje prema nivou usklađenosti sa ovim standardima.

Kontrola kvaliteta softvera (eng. software quality assurance, SQA) pokriva kompletan proces razvoja softvera i sve njegove faze i podfaze. Proces kontrole kvaliteta, takođe opisan standardom ISO/IEC 15504, treba da osigura nezavisnu potvrdu da su svi proizvodi, aktivnosti i procesi u skladu sa predefinisanim planovima i standardima.

Faze razvoja softvera i moguće probleme na šaljiv način ilustruje čuvena karikatura prikazana na slici 8.1.

Za razvoj softvera relevantni su i procesi istraživanja tržišta, nabavke softvera, naručivanja softvera, tenderi, razmatranje ponuda i slični, ali u ovom tekstu neće biti reči o njima.

Elektronska verzija (2024)



Slika 8.1: Faze razvoja softvera ilustrovane na šaljiv način

8.1 Planiranje

Poslovna analiza u fazi planiranja bavi se, pre svega, preciznom postavkom i specifikovanjem zahteva, dok se modelovanje i dizajn bave razradom projekta koji je definisan u fazi analize. U fazi planiranja često se koriste različite dijagramske tehnike i specijalizovani alati koji podržavaju kreiranje ovakvih dijagrama (takozvani CASE alati, engl. Computer Aided Software Engineering). Procesom planiranja strateški rukovodi arhitekta čitavog sistema (engl. enterprise architect, EA). Njegov zadatak je da napravi opšti, apstraktan plan svih procesa koji treba da budu softverski podržani.

8.1.1 Analiza i specifikovanje problema

Proces analize i specifikovanja problema obično sprovodi *poslovni analitičar* (engl. business analyst, BA), koji nije nužno informatičar, ali mora da poznaje relevantne poslovne ili druge procese. Kada se softver pravi po narudžbini, za poznatog kupca, u procesu analize i specifikovanja problema vrši se intenzivna komunikacija poslovnog analitičara sa naručiocima, krajnjim korisnicima ili njihovim predstavnicima. Kada se softver pravi za nepoznatog kupca, često u kompanijama ulogu naručioca preuzimaju radnici zaposleni u odeljenju prodaje ili marketinga (koji imaju ideju kakav proizvod bi kasnije mogli da prodaju).

U komunikaciji poslovnog analitičara sa naručiocima, često se najpre vrši analiza postojećih rešenja (na primer, postojećeg poslovnog procesa u kompaniji koja uvodi informacioni sistem) i razmatraju se mogućnosti njihovog unapređenja uvođenjem novog softvera. Naručioci često nemaju informatičko obrazovanje, pa njihovi zahtevi koje softver treba da zadovolji mogu da budu neprecizni ili čak i kontradiktorni. Zadatak poslovnog analitičara je da, u saradnji sa naručiocima, zahteve precizira i uobliči. Rezultat analize je opšta specifikacija problema koja opisuje problem (na primer, poslovne procese) i željene funkcionalnosti programa, ali i potrebnu efikasnost i druga svojstva.

Pored precizne analize zahteva, zadatak poslovne analize je i da proceni: obim posla[1] koji treba da bude urađen (potrebno je precizno definisati šta projekat treba da obuhvati, a šta ne); rizike koji postoje (i da definiše odgovarajuće reakcije u slučaju da nešto pođe drugačije nego što je planirano); potrebne resurse (ljudske i materijalne); očekivanu cenu realizacije projekta i njegovih delova; plan rada (po fazama) koji je neophodno poštovati i slično.

Kada je problem precizno specifikovan, prelazi se na sledeće faze u kojima se modeluje i dizajnira rešenje specifikovanog problema.

[1]Obim posla često se izražava u terminima broja potrebnih čovek-meseci (jedan čovek-mesec podrazumeva da jedan čovek na projektu radi mesec dana).

8.1.2 Modelovanje rešenja

Modelovanje rešenja obično sprovodi *arhitekta rešenja* (engl. solution architect, SA), koji mora da razume specifikaciju zahteva i da je u stanju da izradi matematičke modele problema i da izabere adekvatna softverska rešenja, na primer, programski jezik, bazu podataka, relevantne biblioteke, strukture podataka, algoritamska rešenja, itd.

8.1.3 Dizajn softverskog rešenja

– OVDE IDU ONI UUI DIJAGRAMI I SLICNO

U procesu dizajniranja, *arhitekta softvera* (engl. software architect) vrši preciziranje rešenja i opisuje *arhitekturu softvera* (engl. software architecture) – celokupnu strukturu softvera i načine na koje ta struktura obezbeđuje integritet sistema i željeni ishod projekta (ispravan softver, dobre performanse, poštovanje rokova i uklapanje u planirane troškove). Dizajn razrađuje i pojmove koji su u ranijim fazama bili opisani nezavisno od konkretnih tehnologija, dajući opšti plan kako sistem da bude izgrađen na konkretnoj hardverskoj i softverskoj platformi. Tokom dizajna često se koriste neki unapred ponuđeni obrasci (engl. design patterns) za koje je praksa pokazala da predstavljaju kvalitetna rešenja za određenu klasu problema.

U jednostavnijim slučajevima (na primer kada softver treba da radi autonomno, bez korisnika i korisničkog interfejsa), dizajn može biti dat i u neformalnom tekstualnom obliku ili u vidu jednostavnog dijagrama toka podataka tj. tokovnika (engl. data flow diagram)[2]. U kompleksnijim slučajevima, koriste se standardizovane grafičke notacije (kaže se i *grafički jezici*), poput UML (Unified Modeling Language), koji omogućavaju modelovanje podataka, modelovanje poslovnih procesa i modelovanje softverskih komponenti.

Neke od osnovnih tema koje se razmatraju u okviru dizajna softvera su:

- Apstrahovanje (engl. abstraction) – apstrahovanje je proces generalizacije kojim se odbacuju nebitne informacije tokom modelovanja nekog entiteta ili procesa i zadržavaju samo one informacije koje su bitne za sâm softver. Na primer, apstrahovanjem se uočava da boja očiju studenta nema nikakvog značaja u informacionom sistemu fakulteta i ta informacija se onda odbacuje prilikom predstavljanja studenta u sistemu.

- Profinjavanje (engl. refinement) – profinjavanje je proces razvoja programa odozgo-naniže. Nerazrađeni koraci se tokom profinjavanja sve više preciziraju dok se na samom kraju ne dođe do sasvim preciznog opisa u obliku funkcionalnog programskog koda. U svakom koraku jedan zadatak razlaže se na sitnije zadatke. Na primer, u nekoj situaciji, zadatak koji obavlja funkcija `obradi_podatke_iz_datoteke()` razlože se na zadatke koje obavljaju funkcije `otvori_datoteku()`, `procitaj_podatke()`, `obradi_podatke()`, `zatvori_datoteku()`, itd. Apstrahovanje i profinjavanje međusobno su suprotni procesi.

- Dekompozicija (engl. decomposition) – cilj dekompozicije je razlaganje na komponente koje je lakše razumeti, realizovati i održavati. Njen proizvod nije implementacija, već opis arhitekture softverskog rešenja. Postoje različiti pristupi dekompoziciji, obično u skladu sa programskom paradigmom koja će se koristiti (na primer, objektno-orijentisana, funkcionalna, itd). Većina pristupa teži razlaganju na komponente tako da se što više smanje njihove zavisnosti (tako što unutrašnje informacije jednog modula nisu dostupne iz drugih) i da se poveća kohezija (jaka unutrašnja povezanost) pojedinačnih komponenti. Na primer, u funkcijski-orijentisanom dizajnu, svaka funkcija odgovorna je samo za jedan zadatak i sprovodi ga sa minimalnim uticajem na druge funkcije. Rezultat dekompozicije često se prikazuje grafički, u vidu strukturnog modela sistema koji opisuje veze između komponenti i njihovu hijerarhiju (na svakom nivou hijerarhije, svakom čvoru koji nije list, odgovara nekoliko, obično između dva i sedam, podređenih čvorova).

- Modularnost (engl. modularity) – softver se deli na komponente koje se nazivaju moduli. Svaki modul ima precizno definisanu funkcionalnost i poželjno je da moduli što manje zavise jedni od drugih kako bi mogli da se koriste i u drugim programima.

8.2 Metodologije razvoja softvera

I u teoriji i u praksi postoje mnoge metodologije razvoja softvera. U praksi su one često pomešane i često je teško striktno razvrstati stvarne projekte u postojeće metodologije. U nastavku je opisano nekoliko često korišćenih metodologija i ključnih ideja na kojima su zasnovane.

[2]Ovi dijagrami ilustruju kako podaci teku kroz sistem i kako se izlaz izvodi iz ulaza kroz niz funkcionalnih transformacija, ali ne opisuju kako ih treba implementirati. Notacija koja se koristi u tokovnicima nije standardizovana, ali različite notacije su često veoma slične i intuitivne.

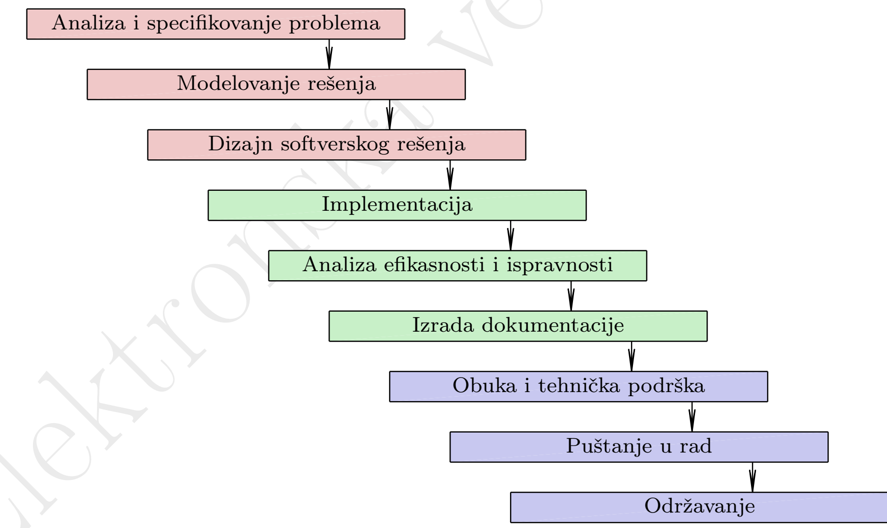
Elektronska verzija (2024)

Metodologija vodopada. Metodologija vodopada je najstarija metodologija. Prvi put formalno je opisana 1970. godine, iako je praksa postojala i ranije, još od pedesetih godina prošlog veka. Vodopad metodologija je bila standard u industriji softvera tokom 70-ih i 80-ih godina posebno u velikim organizacijama kao što su vlade i velike korporacije. U strogoj varijanti ove tradicionalne metodologije na sledeću fazu u razvoju softvera prelazi se tek kada je jedna potpuno završena (slika 8.2):

- *zahtevi:* Skupljanje svih korisničkih zahteva, njihovo dokumentovanje i odobravanje od strane relevantnih strana;
- *dizajn:* Razrada tehničke arhitekture i specifikacija koje će se koristiti tokom implementacije;
- *implementacija:* Pisanje koda prema specificiranom dizajnu i zahtevima;
- *testiranje:* Verifikacija i validacija da sistem funkcioniše prema zahtevima;
- *integracija:* Kombinovanje različitih modula u jedinstven sistem i verifikacija njihove međusobne kompatibilnosti;
- *održavanje:* Faza nakon isporuke gde se ispravljaju greške i dodaju potrebne nadogradnje.

Ova metodologija se smatra primenljivom ako su ispunjeni sledeći uslovi:

- svi zahtevi poznati su unapred i njihova priroda ne menja se bitno u toku razvoja;
- zahtevi su u skladu sa očekivanjima svih relevantnih strana (investitori, korisnici, realizatori, itd.);
- zahtevi nemaju nerazrešene, potencijalno rizične faktore (na primer, rizike koji se odnose na cenu, tempo rada, efikasnost, bezbednost, itd);
- pogodna arhitektura rešenja može biti opisana i podrobno shvaćena;
- na raspolaganju je dovoljno vremena za rad u etapama.



Slika 8.2: Ilustracija za metodologiju vodopada

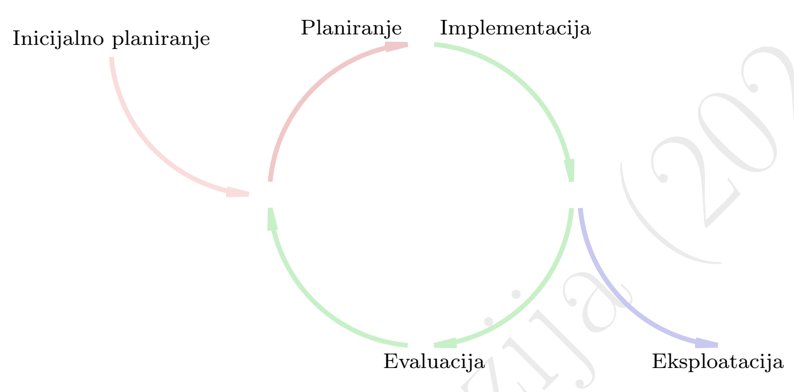
Ova metodologija koristi se obično u veoma velikim timovima. Detaljna dokumentacija za sve faze je neophodna što je korisno u velikim i kompleksnim projektima. Prednosti ove metodologije su jasna struktura jer osigurava da se svaki aspekt detaljno analizira pre nego što se krene sa sledećom fazom.

Metodologija ne predviđa modifikovanje prethodnih faza jednom kada su završene i ova osobina, krutost metodologije, predmet je najčešćih kritika. Često, klijentu je veoma teško da eksplicitno na početku projekta odredi sve zahteve. Izrada aplikacija često traje toliko dugo da se zahtevi promene u međuvremenu i završni proizvod više nije sasvim adekvatan a ponekad ni uopšte upotrebljiv. Korisnici (klijenti) se obično uključuju samo na početku (za zahteve) i na kraju (za prihvatanje), što može dovesti do neslaganja između korisničkih očekivanja i isporučenog proizvoda. Dodatno, postoje problemi i u fazi razvoja kada članovi tima moraju da čekaju izradu zadataka od kojih njihov rad zavisi. Vreme potrošeno čekajući da se zadatak odblokira nekada bude veće od produktivnog vremena.

Elektronska verzija (2024)

Metodologija vodopada se danas retko koristi, mada može biti korisna kod projekta sa veoma jasnim i nepromenljivim zahtevima kao što su sistemi sa visokim nivom rigidnosti (na primer, medicinski uređaji ili avijacija). Poznato je da NASA koristi metodologiju vodopada u okviru izrade softvera za svemirski šatl, što je opravdano imajući na umu da su zahtevi i analiza za ovakav softver unapred striktno i jasno definisani. Postoje varijante metodologije vodopada (na primer, V-metodolologija) koje se češće koriste.

Metodologija iterativnog i inkrementalnog razvoja. U ovoj metodologiji, opisanoj prvi put šezdesetih i sedamdesetih godina prošlog veka (ilustrovanoj slikom 8.3), razvoj se sprovodi u iteracijama i projekat se gradi inkrementalno. Iteracije obično donose više detalja i funkcionalnosti, a inkrementalnost podrazumeva dodavanje jednog po jednog modula, pri čemu i oni mogu biti modifikovani ubuduće. U jednom trenutku, više različitih faza životnog ciklusa softvera može biti u toku. U ovoj metodologiji vraćanje unazad je moguće.



Slika 8.3: Ilustracija za iterativnu metodologiju

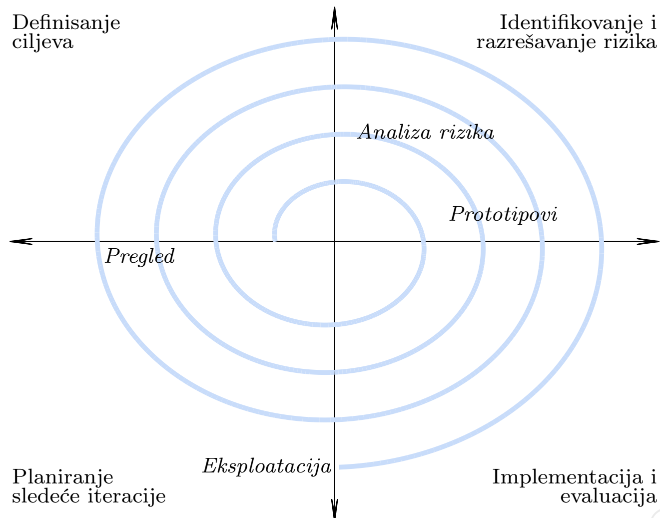
Metodologija rapidnog razvoja. U ovoj metodologiji (engl. rapid application development), opisanoj sedamdesetih i osamdesetih godina prošlog veka, faza planiranja svedena je na minimum zarad brzog dobijanja prototipova u iteracijama. Faza planiranja preklapa se sa fazom implementacije što olakšava izmene zahteva u hodu. Proces razvoja kreće sa razvojem preliminarnog modela podataka i algoritama, razvijaju se prototipovi na osnovu kojih se definišu, preciziraju ili potvrđuju zahtevi naručioca ili korisnika. Ovaj postupak ponavlja se iterativno, sve do završnog proizvoda. Aplikaciju prati vrlo ograničena dokumentacija.

Ova metodologija ponekad može dovesti do niza prototipova koji nikada ne dostižu do zadovoljavajuće finalne aplikacije. Čest izvor takvih problema su grafički korisnički interfejsi (engl. graphical user interface; GUI). Naime, korisnici napredak u razvoju aplikacije doživljavaju prvenstveno kroz napredak grafičkog korisničkog interfejsa. To podstiče programere, pa i vođe projekata, da se u prototipovima usredsređuju na detalje grafičkog interfejsa umesto na druge segmente aplikacije (kao što su, na primer, poslovni procesi i obrada podataka). Čak i mnogi razvojni alati privilegovano mesto u razvoju softvera daju razvoju grafičkih interfejsa. Ovakav razvoj aplikacije često dovodi do niza prototipova sa razrađenim korisničkim interfejsom, ali bez adekvatnih obrada koje stoje iza njega.

Ova metodologija pogodna je za razvoj softvera za sopstvene potrebe ili za potrebe ograničenog broja korisnika.

Spiralna metodologija. Ova metodologija (opisana prvi put krajem osamdesetih godina prošlog veka) kombinuje analizu rizika sa drugim metodologijama kao što su metodologija vodopada i metodologije iterativnog razvoja. Spirala koja ilustruje ovu metodologiju (prikazana na slici 8.4) prolazi više puta kroz faze kao što su planiranje, implementacija i evaluacija tekućeg verzije, kao i analiza rizika. Različite faze ne sprovode se istovremeno, već jedna za drugom. Prvi prototip pravi se na osnovu preliminarnog, pojednostavljenog dizajna i predstavlja samo aproksimaciju finalnog proizvoda. Na kraju svake iteracije, prototip se evaluira, analiziraju se njegove dobre i loše strane, profinjuje specifikacija za sledeću iteraciju i analiziraju se rizici (rizici koji se odnose na bagove, na cenu, tempo rada, efikasnost, bezbednost, itd). Na primer, planiranje dodatnog testiranja smanjuje rizik od neispravnog proizvoda, ali može da uveća cenu ili da nosi rizik zakasnelog izlaska na tržište. Ako neki rizik ne može biti eliminisan, naručilac mora da odluči da li se sa projektom nastavlja ili ne. Ukoliko se sa projektom nastavlja, ulazi se u sledeću iteraciju.

Elektronska verzija (2024)



Slika 8.4: Ilustracija za spiralnu metodologiju

Agilna metodologija razvoja. Agilne metodologije su upotrebi od devedesetih godina prošlog veka a trenutno su verovatno najpopularnije. Ova metodologija stavlja fokus na zadovoljstvo korisnika i zato se podstiče rana i inkremetalna isporuka softvera u vidu iteracija sa minimalnim dodavanjem funkcionalnosti u kratkim vremenskim intervalima (obično od jedne do četiri nedelje). Na ovaj način se teži minimizovanju rizika, kao što su bagovi, prekoračenje budžeta ili izmena zahteva. Dodatne smernice za razvoj daju prioritet isporuci naspram analize i dizajna (iako ove aktivnosti nisu obeshrabrene).

Radi se u malim, visokomotivisanim timovima koji su samoorganizovani i imaju kontrolu nad odlukama o projektu. Agilne metodologije zahtevaju permanentnu komunikaciju, poželjno uživo (zbog čega, međutim, ne ostaje mnogo pisanog traga o progresu niti pisane dokumentacije).

Agilne metode su razvijene u nastojanju da se prevaziđu uočene slabosti konvencionalnog razvoja softvera. Agilni razvoj može doneti važne prednosti, ali nije primenjiv na sve projekte, sve proizvode, sve ljude i sve situacije. U današnjem vremenu često je teško ili nemoguće predvideti kako će se softver razvijati kako vreme prolazi. Potrebe krajnjih korisnika se menjaju, a novi konkurentski proizvodi i rešenja pojavljuju se nekada iznenada, bez upozorenja. Zato, u mnogim situacijama nije moguće u potpunosti definisati specifikacije pre početka projekta i razvoj softvera mora biti dovoljno agilan da bi se prilagodio novim, promenjenim zahtevima. Sa druge strane, promene su skupe. Jedna od najprivlačnijih karakteristika agilnog pristupa je njegova sposobnost da smanji troškove promena tokom celog softverskog procesa.

Manifest agilne metodologije je osnovni dokument koji opisuje principe i vrednosti agilnog razvoja softvera. Kreiran je 2001. godine od strane grupe programera i ima dvanaest jednostavnih principa, kao što su: glavna mera napretka je upotrebljivost raspoloživog softvera, održivi razvoj, neprekidna usredsređenost na dobar dizajn, pojedinci i interakcije pre procesa i alata, funkcionalan softver pre obimne dokumentacije, odgovor na promene pre nego praćenje plana, itd.

Jedan od ciljeva ove metodologije je u ranom otkrivanju i ispravljanju propusta i neusklađenih očekivanja. Svaka iteracija odnosi se na minijaturni softverski proizvod sa svim uobičajenim fazama razvoja (koje se izvršavaju istovremeno). Svaku iteraciju potrebno je završiti na vreme i dobiti saglasnost naručioca. Za razliku od rapidne metodologije, u okviru koje se, u iteracijama, razvijaju nekompletni prototipovi, u agilnoj metodologiji, nakon nekih iteracija softver može biti isporučen naručiocu (ili na tržište) iako nema upotpunjenu funkcionalnost.

Agilna metodologija u mnogim je aspektima razvoja softvera uopštena, te postoji više vidova ove metodologije koji preciziraju neke njene aspekte, uključujući skram i ekstremno programiranje.

Skram (engl. scrum) je vid agilne metodologije u kojem se neposredna, praktična iskustva koriste u upravljanju izazovima i rizicima. Skram rayvoj se sastoji od jednog ili više Skram timova, pri čemu svaki Skram tim čine tri uloge: vlasnik proizvoda, skram master i razvojni tim (Slika ??).

Vlasnik proizvoda (engl. product owner) je osoba sa vizijom i autoritetom koja usmerava članove tima. Vlasnih proizvoda ima ključnu ulogu u osiguravanju sveukupnog uspeha rešenja koje se razvija ili održava.

Elektronska verzija (2024)

Njegova osnovna odgovornost je da odredi *šta* će biti razvijeno i *kojim redosledom*. U tom smislu, vlasnik proizvoda nadgleda, a to uključuje kreiranje, redefinisanje, procenu i prioritizaciju spiska zahteva (eng. scrum backlog). Time vlasnik proizvoda osigurava da se donose dobre finansijske odluke na svim nivoima – od planiranja verzije proizvoda koja će biti objavljena, ali i sprinta pa do samog spiska zahteva proizvoda. Takođe, na kraju svakog sprinta, vlasnik proizvoda je odgovoran za odluku o tome da li će se finansirati sledeći sprint.

Kako bi ispunio ove odgovornosti, vlasnik proizvoda mora balansirati dve ključne uloge. S jedne strane, on predstavlja naručioce – kupce i korisnike i njegova uloga je da razume njihove potrebe i prioritete. Sa druge strane, on komunicira i sa razvojnim timom. On definiše kriterijume za prihvatanje funkcionalnosti koje se razvijaju i osigurava da se sprovode testovi kako bi se ti kriterijumi proverili. Tako da je on delom i poslovni analitičar, ali i tester.

Ključno je da vlasnik proizvoda proverava kriterijume za prihvatanje proizvoda tokom izvođenja sprinta, umesto da čeka na kraj sprinta. Funkcionalnosti se testiraju čim su završene i vlasnik proizvoda može brzo da identifikuje greške i nesporazume, i tako da omogućavajući timu da te probleme reši pre kraja sprinta.

Skram master je osoba koja olakšava komunikaciju između vlasnika proizvoda i tima. Skram master pomaže timu i organizaciji da se pridržavaju skram vrednosti i principa. Oni vode tim kroz rešavanje problema i uklanjanje prepreka, radi poboljšanja skram procesa i štite tim od spoljašnjih ometanja. Skram master nije tradicionalan menadžer tima i ne vrši kontrolu.

Razvojni tim je samoorganizovana, multidisciplinarna grupa odgovorna za dizajn, izgradnju i testiranje proizvoda. Tim obično ima od pet do devet članova, uključujući osobe sa različitim veštinama kao što su programiranje, testiranje i dizajn korisničkog interfejsa. Ova raznolikost osigurava da tim poseduje sve potrebne veštine da isporuči visokokvalitetan, funkcionalan softver, bez potrebe da se oslanja na druge timove ili da prenosi posao, što često vodi do nesporazuma i kašnjenja.

Tokom izvršenja sprinta, tim sarađuje kako bi stavke iz liste zahteva pretvorio u potencijalno isporučive funkcionalnosti. Oni sami organizuju svoj rad, planiraju, upravljaju i realizuju zadatke, uz održavanje redovne komunikacije kroz dnevne sastanke. Ovi dnevni sastanci omogućavaju timu da pregleda napredak, prilagodi planove i osigura usklađenost sa ciljem sprinta. Na početku svakog sprinta, učestvuju u planiranju sprinta sa vlasnikom proizvoda i skram masterom kako bi odredili najvažnije stavke liste zahteva koje treba obraditi, osiguravajući usklađenost tima sa ukupnim ciljevima projekta.

Razvojni tim takođe igra ključnu ulogu u pregledima sprinta i retrospektivama, gde ceo skrum tim i zainteresovane strane procenjuju obavljeni posao i identifikuju oblasti za poboljšanje. Radom u kratkim, iterativnim ciklusima, tim može kontinuirano unapređivati proizvod, kao i svoje procese, osiguravajući stalno poboljšanje i prilagođavanje.

Svaki član razvojnog tima mora biti potpuno posvećen i fokusiran na postizanje ciljeva sprinta. Tim mora raditi transparentno, otvoreno deliti informacije i sarađivati u rešavanju problema čim se pojave. Tim uspeva ili ne uspeva kao celina, sa naglaskom na kolektivno vlasništvo nad poslom. U skladu sa skramovim principom održivog tempa, tim osigurava da isporučuje visokokvalitetne proizvode bez preopterećenja, što omogućava zdravu, produktivnu i prijatnu radnu atmosferu.

Softverski proizvod sve vreme se održava u (integrisanom i testiranom) stanju koje se potencijalno može isporučiti. Vreme je podeljeno u kratke intervale, „sprintove", obično duge samo jedan mesec ili kraće i na kraju svakog sprinta svi akteri i članovi tima sastaju se da razmotre stanje projekta i planiraju dalje korake. Skram ima jednostavan skup pravila, zaduženja i sastanaka koji se, zarad jednostavnosti i predvidivosti, nikad ne menjaju. Postoje sastanci na početku i kraju svakog sprinta, ali i kratki, petnaestominutni dnevni sastanci („dnevni skram").

SLIKA AKTIVNOSTI

Ekstremno programiranje je vid agilne metodologije u kojem su posebno važne jednostavnost, motivacija i kvalitetni odnosi unutar tima. Programeri rade u parovima (dok jedan programer piše kôd, drugi pokušava da pronađe i ukaže na eventualne greške i nedostatke) ili u većim grupama, na kodu jednostavnog dizajna koji se temeljno testira i unapređuje tako da odgovara tekućim zahtevima. U ekstremnom programiranju, sistem je integrisan i radi sve vreme (iako svesno nema potpunu funkcionalnost). Svi članovi tima upoznati su sa čitavim projektom i pišu kôd na konzistentan način, te svako može da razume kompletan kôd i da radi na svakom delu koda. U ekstremnom programiranju, faze se sprovode u veoma malim koracima i prva iteracija može da dovede, do svesno nepotpune ali funkcionalne celine, već za jedan dan ili nedelju. Zahtevi se obično ne mogu u potpunosti utvrditi na samom početku, menjaju se tokom vremena, te naručilac treba da konstantno bude uključen u razvojni tim. Naručiocu se ne prikazuju samo planovi i dokumenti, već

konstantno i (nekompletni, nesavršeni, ali funkcionalni) softver. Dokumentacija mora da postoji, ali se izbegava preobimna dokumentacija.

8.3 Ekspoloatacija

Ovde nesto reci o: - Deployment - Post-Deployment Monitoring - End-of-Life Management - Continuous integration and continuous delivery (CI/CD) - Availability and reliability and security

8.4 Alati i tehnike korišćeni u razvoju softvera

8.4.1 – project management tools (JIRA, Trello)

8.4.2 Sistemi za kontrolu verzija

Sistemi za kontrolu verzija (eng. *version control systems – VCS*) su ključni alati u razvoju softvera za praćenje promena, saradnju i upravljanje istorijom projekta. Među njima, Git je najpopularniji i najšire korišćen. Ovi sistemi su poznati i pod drugim imenima, recimo kao menadžer izvornog koda (eng. *Source Code Management – SCM*) ili kao sistem za kontrolu revizija (eng. *Revision Control System – RCS*). Bez obzira na terminologiju, cilj ostaje isti: uvati sadržaj, beležiti sve promene i omogućiti pristup različitim verzijama.

Prilikom rada na projektu veoma je važno osigurati čuvanje rezervne kopije projekta jer može doći do gubitka podataka usled greške u kodu ili kvara diska. Održiva i pouzdana strategija čuvanja rezervene kopije projekta obično uključuje i kontrolu verzija omogućavajući programerima praćenje i upravljanje revizijama. **Repozitorijum** je centralizovano skladište gde se čuva sav izvorni kod projekta zajedno sa istorijom svih izmena. Repozitorijum sadrži:

Verzije datoteka – Sve datoteke i dokumenti projekta, ali i sve njihove različite verzije obeležene i datumom i vremenom kada je neka verzija napravljena.

Istorija izmena – Evidencija o svakoj promeni koja je napravljena u kodu, uključujući ko je napravio promenu, kada je promena napravljena, kao i opis promene. Ove izmene ujedno predstavljaju i formalan način komunikacije između članova tima koji rade na istom projektu.

Grane – Različite verzije koda koje se mogu razvijati paralelno i na taj način je omogućeno istovremeno razvijanje različitih funkcionalnosti ili verzija projekta.

Repozitorijum omogućava timovima da efikasno sarađuju, prate promene i vrše integraciju različitih delova koda na kontrolisan način.

U početku, upravljanje skladištem je bila ključna funkcija sistema za kontrolu verzija, smanjujući prostor na disku potreban za održavanje svih verzija. Kada se kreira nova verzija, čuva se samo razlika između nje i prethodne verzije (*delta*). U okviru repozitorijuma se čuvaju sve delte (izmene) nad projektom. Te delte, kada se primene na osnovnu verziju, ponovo kreiraju ciljnu verziju. Obično i na repozitorijumu i na lokalnom računaru na kom programer radi na projektu se čuva najnovija, poslednja verzija, a korišćenjem delti je moguće kreirati ranije verzije.

U timskom razvoju softvera, različiti članovi tima često rade na istoj komponenti istovremeno. U ovakvim okolnostima važno je izbeći sukobe između njihovih promena. Sistemi za kontrolu verzija koriste model javnog repozitorijuma i privatnog radnog prostora. Programeri preuzimaju komponente iz repozitorijuma u svoj privatni radni prostor, prave promene, a zatim ih vraćaju nazad. Ako više ljudi radi na jednoj komponenti, sistem upozorava ostale i osigurava da izmenjene komponente dobiju različite identifikatore verzija, kao i integraciju svih izmena u jedinstven kod. Dodatno, programerima je omogućeno da dodaju opise svojih izmena, tako da svi članovi tima mogu da razumeju zbog čega je neka promena nastala. Ovo je često poželjno, a zapravo u mnogim organizacijama i obavezan korak prilikom unosa izmena u kod.

Dodatno, sistemi za kontrolu verzija često imaju pridružene i dodatne alate za analizu koda. Na primer, alati koji formatiraju kod, alati za statičku analizu koda koji otkrivaju potencijalne greške, i alati za kontinualnu integraciju koji automatski prevode i testiraju kod. Kod koji ne može da se prevede ili ne prolazi sve testove obično se ne može ni postaviti na glavni ili javni repozitorijum.

Veoma često, deo sistema za kontrolu verzija je i proces revizije koda (eng. *code review*). Pre nego što se izmene unesu u glavni repozitorijum, drugi programeri, članovi tima, ili supervizori pregledaju predložene izmene. Tokom ove revizije, oni analiziraju kvalitet, efikasnost, sigurnost i održivost koda. Nakon što se utvrdi da je kod u skladu sa standardima projekta i da nema kritičnih grešaka, odobrava se integracija izmena u glavni repozitorijum.

Kao što je već rečeno, najpoznatiji i najšire korišćen sistem za kontrolu verzija je **Git**. Git je *distribuirani sistem* za kontrolu verzija, što znači da svaki korisnik ima punu kopiju istorije repozitorijuma, što omogućava

rad van mreže i veću otpornost na greške. Pored Gita, postoje i drugi sistemi za kontrolu verzija, kao što su Subversion (SVN) i Mercurial. SVN je centralizovani sistem, što znači da postoji jedan centralni repozitorijum kojem korisnici pristupaju. Ovo olakšava centralizovanu kontrolu i nadzor, ali može biti manje fleksibilno u poređenju sa distribuiranim sistemima kao što je Git. Mercurial je takođe distribuirani sistem, sličan Gitu, ali je Git poznat po jednostavnosti korišćenja i brzini.

Linux kernel je razvijan pomoću BitKeepera, komercijalnog alata za kontrolu verzija. Godine 2005. kompanija koja poseduje BitKeeper odlučila je da više ne dozvoljava besplatno korišćenje alata. Linux zajednica je morala da pronađe drugačije rešenje. Linus Torvalds je tražio besplatan alat koji bi zadovoljio sve potrebe za razvoj Linux kernela, pa je osmislio i razvio Git zajedno sa grupom programera. Git je morao da zadovolji nekoliko ključnih zahteva:

- **Distribuiran razvoj:** Omogućiti paralelan i nezavisan razvoj u privatnim repozitorijumima bez stalne potrebe za sinhronizacijom sa centralnim repozitorijumom. Programeri mogu raditi na različitim lokacijama, čak i van mreže, uz istovremeno omogućavanje hiljadama programera da rade na istom projektu. Svaki repozitorijum ima kompletnu istoriju svih promena.
- **Brzina i efikasnost:** Da bi se uštedelo na prostoru i skratilo vreme prenosa, korišćene su kompresije i đelta"tehnika. Distribuirani model umesto centralizovanog modela osigurao je da kašnjenje mreže ne ometa svakodnevni razvoj.
- **Pouzdanost:** Pošto je Git distribuirani sistem za kontrolu revizija, važno je imati apsolutnu sigurnost da je integritet podataka očuvan. Git koristi kriptografsku hash funkciju SHA1 (eng. *Secure Hash Function*) za imenovanje i identifikaciju objekata u svojoj bazi podataka, što osigurava integritet i poverenje u distribuirane repozitorijume.
- **Preuzimanje odgovornosti:** Ključni aspekt sistema za kontrolu verzija je znati ko je promenio datoteke i, ako je moguće, zašto. Git nameće vođenje evidencije o izmenama pri svakom menjanju datoteke.
- **Nepromenljivost:** Git-ova baza podataka repozitorijuma sadrži objekte koji su nepromenljivi. To znači da, kada su kreirani i smešteni u bazu podataka, ne mogu biti izmenjeni. Dizajn Git baze podataka znači da je cela istorija koja se nalazi unutar baze podataka za kontrolu verzija takođe nepromenljiva.
- **Atomske transakcije:** Niz promena koje treba da se obave se obavljaju sve zajedno ili se uopšte ne obavljaju. To znači da ako prilikom unosa izmena se desi da mrežna veza se prekine ili server prestane da radi, izmene neće biti delimično primenjene i na taj način ostaviti datoteku u neispravnom stanju.
- **Grane:** Omogućiti paralelno razvoj različitih grana u okviru kojih se mogu razvijati različite funkcionalnosti. Omogućiti i spajanje grana u jednu.
- **Slobodan za korišćenje.**

Ima mnogo različitih načina za korišćenje Gita. U ovoj knjizi Git se koristi putem komandne linije. Komandna linija je jedino mesto gde možete pokrenuti sve Git komande – većina GUI-ja implementira samo delimičan skup Git funkcionalnosti radi jednostavnosti i izbor grafičkog klijenta je stvar ličnog ukusa.

8.4.3 – continuous integration tools (Jenkins)

8.5 Novi trendovi i tehnologije

– DevOps – Microservices – Cloud Computing – AI

8.6 Dodatno

– Distributed software engineering – Web development – Service-oriented architecture – Embedded software – Component-Based Software Engineering – UML diagrams

– TESTIRANJE

Pitanja i zadaci za vežbu

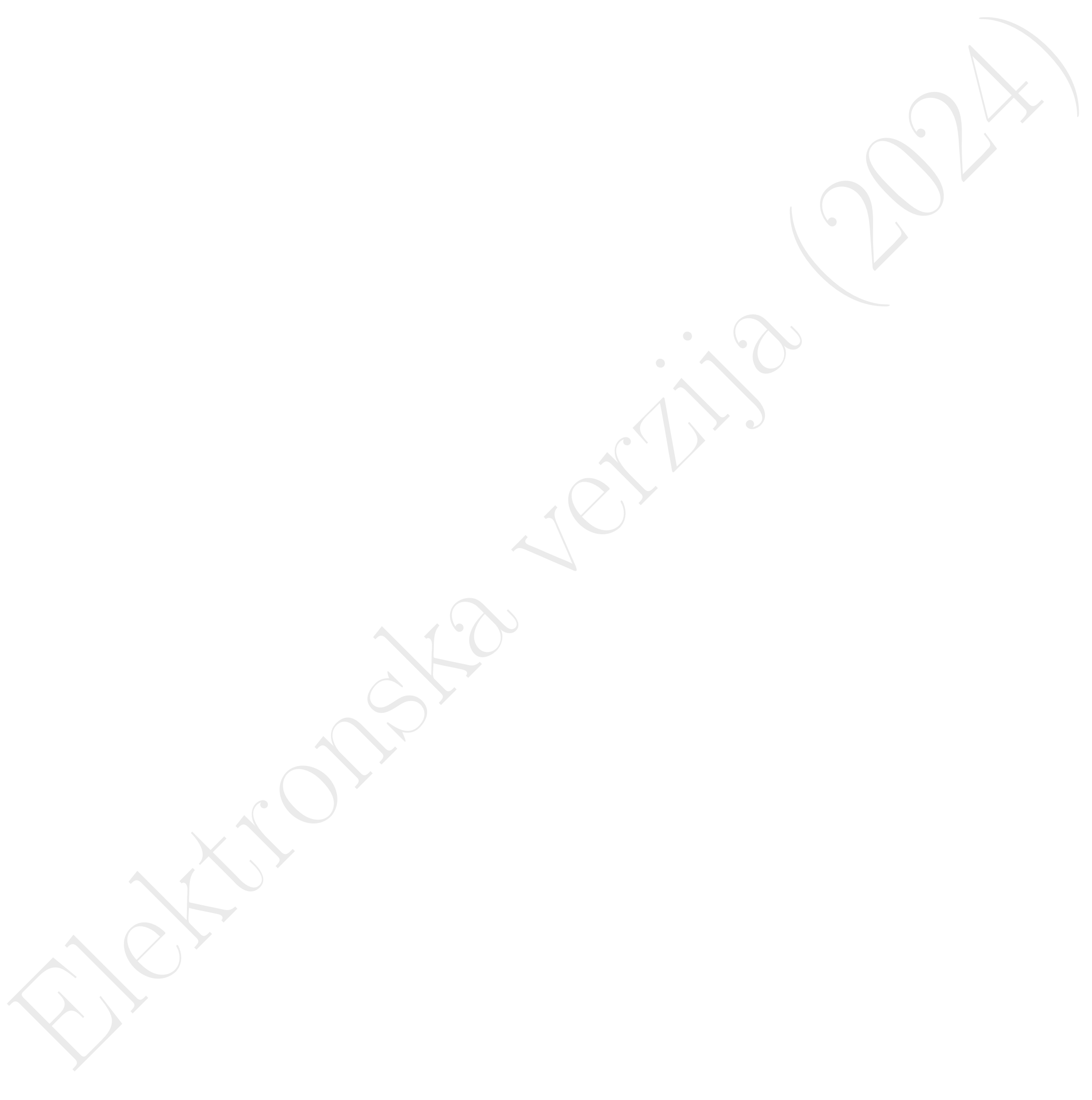
Pitanje 8.1. *Šta su sličnosti a koje razlike između projekata u građevinarstvu i informacionim tehnologijama?*

Pitanje 8.2. *Navesti faze razvoja softvera i ko ih obično sprovodi.*

Elektronska verzija (2024)

Pitanje 8.3. *Koje dve vrste dokumentacije treba da postoje?*

Pitanje 8.4. *Nabrojati najznačajnije metodologije razvoja softvera. Istraži na internetu koje su metodoloje razvoja softvera danas najpopularnije.*


Elektronska verzija (2024)

Glava 9

Baze podataka

9.1 Uvod u baze podataka

9.1.1 Šta je baza podataka?

Baza podataka predstavlja organizovani sistem za skladištenje i upravljanje informacijama. Njena primarna svrha je da omogući efikasno kreiranje, pretragu, ažuriranje i brisanje podataka. Za razliku od tradicionalnih metoda skladištenja podataka, kao što su fajl sistemi, baze podataka nude složenije strukture koje podržavaju različite upite i analize. Moderni sistemi za baze podataka omogućavaju integraciju sa različitim aplikacijama i korisnicima, što ih čini ključnim delom IT infrastrukture. Zbog svoje sposobnosti da upravlja velikim količinama podataka i obezbedi visok nivo integriteta, baze podataka su neophodne za mnoge organizacije i aplikacije.

9.1.2 Tipovi baza podataka

Postoji nekoliko vrsta baza podataka, od kojih su najčešće relacione i NoSQL baze podataka. Relacione baze podataka (RDBMS) organizuju podatke u tabelama sa jasno definisanim relacijama između njih. NoSQL baze podataka, s druge strane, nude fleksibilnije modele podataka, kao što su dokumenti, grafovi i ključ-vrednost parovi. Hibridne baze podataka kombinuju karakteristike oba pristupa, omogućavajući kombinaciju fleksibilnosti i strukture. Razumevanje različitih tipova baza podataka pomaže u odabiru najprikladnijeg sistema za specifične zahteve i aplikacije.

9.2 Relacione baze podataka

9.2.1 Osnovni pojmovi

Relacione baze podataka koriste tabelarni format za organizaciju podataka. Tabele se sastoje od redova i kolona, gde svaki red predstavlja jedinstven zapis, a svaka kolona predstavlja atribut ili karakteristiku podataka. Ključne komponente uključuju primarni ključ, koji jedinstveno identifikuje svaki red u tabeli, i strani ključ, koji uspostavlja veze između tabela. Ova struktura omogućava lako pretraživanje i manipulaciju podacima, kao i održavanje integriteta podataka kroz relacije. Relacione baze podataka omogućavaju kompleksne upite koristeći SQL jezik, što ih čini veoma moćnim alatima za analizu i upravljanje podacima.

9.2.2 Normalizacija

Normalizacija je proces organizovanja podataka u bazi kako bi se smanjila redudantnost i poboljšalo integritet podataka. Proces normalizacije se sastoji od nekoliko koraka, poznatih kao forme normalizacije. Prva normalna forma (1NF) osigurava da su svi atributi u tabeli atomski i da ne sadrže ponovljene grupe. Druga normalna forma (2NF) eliminiše delimičnu zavisnost između atributa, dok treća normalna forma (3NF) uklanja tranzitive zavisnosti. Cilj normalizacije je poboljšanje efikasnosti baze podataka i smanjenje mogućnosti za anomalije prilikom ažuriranja podataka.

9.2.3 SQL

Structured Query Language (SQL) je standardni jezik za upravljanje relacijskim bazama podataka. SQL omogućava korisnicima da kreiraju, pretražuju, ažuriraju i brišu podatke u bazi. Osnovne SQL komande uključuju

Elektronska verzija (2024)

SELECT za preuzimanje podataka, INSERT za dodavanje novih redova, UPDATE za modifikaciju postojećih podataka i DELETE za brisanje podataka. JOIN operacije omogućavaju kombinovanje podataka iz više tabela na osnovu zajedničkih atributa, omogućavajući složene analize i izveštaje. SQL je ključni alat za rad sa relacijskim bazama podataka i pomaže u optimizaciji i upravljanju velikim količinama podataka.

9.3 NoSQL baze podataka

9.3.1 Šta su NoSQL baze podataka?

NoSQL baze podataka predstavljaju alternativu tradicionalnim relacionim bazama podataka i dizajnirane su za rad sa velikim količinama neuređenih ili polu-uređenih podataka. One pružaju fleksibilnost u pogledu strukture podataka, omogućavajući korisnicima da biraju između različitih modela, kao što su dokumenti, grafovi i ključ-vrednost parovi. NoSQL baze su često optimizovane za velike obime podataka i visoke brzine upita, što ih čini pogodnim za aplikacije kao što su društvene mreže i sistemi za analizu u realnom vremenu. Ove baze obično ne koriste tradicionalne SQL upite i transakcije, već nude alternativne metode za upravljanje podacima. Kao rezultat, NoSQL baze pružaju veću fleksibilnost i skalabilnost u određenim situacijama.

9.3.2 Vrste NoSQL baza podataka

Postoji nekoliko vrsta NoSQL baza podataka, svaka sa svojim specifičnostima i primenama. Dokument-orijentisane baze, kao što je MongoDB, čuvaju podatke u formatima kao što su JSON ili BSON, što omogućava dinamične i složene strukture podataka. Ključ-vrednost baze, poput Redis, skladište podatke kao parove ključ-vrednost, omogućavajući brzi pristup i jednostavnu manipulaciju podacima. Kolon-orijentisane baze, kao što je Cassandra, optimizovane su za rad sa velikim količinama podataka u kolona, što poboljšava performanse upita. Graf baze podataka, poput Neo4j, omogućavaju modeliranje i pretraživanje kompleksnih mreža i veza između podataka. Svaka od ovih vrsta ima svoje specifične prednosti i primene u zavisnosti od tipa aplikacija i potreba korisnika.

9.3.3 Kada koristiti NoSQL?

NoSQL baze podataka su idealne za aplikacije koje zahtevaju visoku skalabilnost i fleksibilnost u radu sa podacima. One su posebno korisne za aplikacije koje upravljaju nestrukturiranim ili polu-strukturiranim podacima, kao što su logovi ili društvene mreže. NoSQL baze nude prednosti u okruženjima sa velikim obimima podataka i visokim brzinama upita, omogućavajući brzu i efikasnu obradu. Ako aplikacija zahteva visoku dostupnost i otpornost na greške, NoSQL može biti bolji izbor od tradicionalnih RDBMS. Ipak, za aplikacije koje zahtevaju složene transakcije i visok nivo integriteta podataka, relacione baze podataka često su pogodnije.

9.4 Arhitektura baza podataka

9.4.1 Komponente baze podataka

Arhitektura baze podataka uključuje nekoliko ključnih komponenti koje rade zajedno kako bi omogućile efikasno skladištenje i manipulaciju podacima. Fizičko skladište podataka može biti na disku ili u memoriji, u zavisnosti od zahteva performansi. Upravljački sistem baze podataka (DBMS) je softver koji upravlja svim operacijama nad podacima, uključujući kreiranje, ažuriranje i pretragu podataka. Interfejsi za korisnike i aplikacije omogućavaju komunikaciju između korisnika i baze podataka, često putem SQL upita ili API-ja. Sve ove komponente zajedno čine osnovu za efikasan rad baze podataka.

9.4.2 Pitanja performansi

Indeksi su ključni za poboljšanje performansi pretrage podataka u tabelama. Oni omogućavaju brži pristup podacima, ali dodavanje indeksa može usporiti operacije umetanja i ažuriranja. Optimizacija upita uključuje analizu i prilagođavanje SQL upita kako bi se minimizirali troškovi izvođenja i poboljšala brzina. Normalizacija i denormalizacija tabela mogu uticati na performanse upita, zavisno od vrste operacija koje se izvode. Redovno praćenje performansi baze podataka i prilagođavanje konfiguracija može pomoći u održavanju efikasnosti sistema.

Elektronska verzija (2024)

9.5 Sigurnost i backup

9.5.1 Sigurnost baza podataka

Sigurnost baza podataka je ključna za zaštitu podataka od neovlašćenog pristupa i napada. Kontrola pristupa obuhvata autentifikaciju korisnika i autorizaciju, osiguravajući da samo ovlašćeni korisnici imaju pristup određenim podacima. Enkripcija podataka štiti podatke kako u mirovanju, tako i tokom prenosa, čineći ih nečitljivim za neovlašćene osobe. Implementacija sigurnosnih politika i procedura, uključujući redovno praćenje i reviziju pristupa, pomaže u prevenciji sigurnosnih incidenata. Takođe, redovno ažuriranje softverskih komponenti i primena sigurnosnih zakrpa su ključni za zaštitu sistema od poznatih ranjivosti.

9.5.2 Backup i oporavak

Strategije za backup uključuju pune, inkrementalne i diferencijalne kopije podataka, svaka sa svojim prednostima i izazovima. Pune kopije podataka prave rezervnu kopiju celokupne baze, dok inkrementalne kopije beleže samo promene od poslednjeg backupa. Planiranje oporavka od katastrofa uključuje pripremu strategija za brzo vraćanje podataka u slučaju gubitka. Testiranje procedura za oporavak pomaže u osiguravanju efikasnosti alata i tehnika za vraćanje podataka. Redovno pravljenje backup-a i čuvanje kopija na različitim lokacijama poboljšava otpornost na gubitak podataka i omogućava brzo obnavljanje sistema.

9.6 Trendovi i budućnost baza podataka

9.6.1 Cloud baze podataka

Cloud baze podataka nude brojne prednosti, uključujući skalabilnost, fleksibilnost i smanjene troškove održavanja. Korisnici mogu lako prilagođavati kapacitet i resurse baze podataka prema potrebama aplikacije. Većina cloud provajdera nudi automatizovane funkcionalnosti za backup i sigurnost, što olakšava upravljanje podacima. Međutim, korišćenje cloud baza može izazvati zabrinutosti u vezi sa privatnošću i kontrolom podataka. Odluka o prelasku na cloud bazu zavisi od specifičnih potreba organizacije i zahteva za bezbednost i performanse.

9.6.2 Big Data i baze podataka

Tehnologije za analizu velikih podataka, kao što su Hadoop i Spark, omogućavaju obradu ogromnih količina podataka koje tradicionalne baze ne mogu efikasno obraditi. Integracija sa ovim tehnologijama pomaže u izvođenju kompleksnih analiza i uvida iz velikih skupova podataka. Big Data baze često nude distribuirano skladištenje i sposobnosti obrade velike količine podataka, što omogućava rad sa velikim obimima podataka i visoke brzine upita. Ove tehnologije omogućavaju analizu u realnom vremenu i mogu značajno poboljšati donošenje odluka na osnovu podataka. U budućnosti, očekuje se da će razvoj tehnologija u ovoj oblasti dodatno unaprediti efikasnost i kapacitet baza podataka.

9.6.3 Veštačka inteligencija i baze podataka

Veštačka inteligencija (AI) se koristi za unapređenje performansi baza podataka kroz automatsko optimizovanje upita i predviđanje potreba za resursima. AI može pomoći u identifikaciji obrazaca i anomalija u podacima, što poboljšava sigurnost i analize. Uvođenje AI u baze podataka omogućava bolje upravljanje podacima i donošenje odluka na osnovu složenih analiza. AI takođe može unaprediti korisničko iskustvo kroz personalizovane preporuke i automatizaciju zadataka. Razvoj AI tehnologija može dovesti do pametnijih i efikasnijih baza podataka, koje će bolje odgovoriti na potrebe modernih aplikacija i korisnika.

9.7 Pitanja i odgovori

Elektronska verzija (2024)

Glava 10

Matematika i informatika

Elektronska verzija (2024)

Matematika i računarstvo i informatika su duboko povezane naučne oblasti. Savremeno računarstvo je utemeljeno na matematičkim teorijama (poput, na primer, matematičke logike), ali često i na matematičkim metodama koje podrazumevaju strogo i precizno opisivanje računarskih sistema i deduktivno rezovanje i izvođenje pouzdanih zaključaka (pre svega u domenu izgradnje korektnih i pouzdanih računarskih sistema). Sa druge strane, računarstvo često kombinuje ovaj pristup sa inženjerskim pristupom u kom se teži izgradnji što efikasnijih sistema, koji mogu da budu zasnovani i na različitim heurističkim komponentama čije funkcionisanje ne mora biti u potpunosti razjašnjeno.

Računarstvo matematici pruža alat (računarske programe) koji olakšava rešavanje praktičnih matematičkih zadataka.

10.1 Primene matematike u računarstvu i informatici

U nastavku ćemo navesti kratak pregled koji pokazuje kako se različite oblasti matematike koriste u računarstvu. U računarstvu se koriste kako tehnike diskretne matematike (matematičke logike, teorije skupova, kombinatorike, teorije grafova, teorije brojeva), tako i tehnike kontinualne matematike (pre svega numeričke matematike i verovatnoće i statistike).

10.1.1 Matematička logika

Matematička logika je grana matematike koja se bavi formalnim sistemima zaključivanja i proučavanjem logičkih struktura. Osnovni cilj matematičke logike je formalno analiziranje i razjašnjavanje osnova matematike, uključujući načine na koje matematički argumenti mogu biti provereni i dokazani. Ova disciplina obuhvata proučavanje formalnih jezika, teorije dokaza, teorije modela, teorije skupova i slično. Matematička logika igra ključnu ulogu u razumevanju fundamentalnih principa na kojima počiva matematika i pruža alate za rešavanje logičkih i matematičkih problema.

U računarstvu, matematička logika ima ključnu ulogu u razvoju teorijskih osnova, algoritama i praktičnih primena. Razumevanje matematičke logike omogućava dizajn korektnih i efikasnih računarskih sistema, verifikaciju softvera i hardvera, te razvoj inteligentnih sistema.

Iskazna logika proučava formule dobijene primenom logičkih veznika $\wedge$, $\vee$, $\neg$ i slično na iskazna slova. Na primer,

$$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q.$$

Iskazna logika predstavlja osnovu dizajna logičkih kola (hardverskih komponenti). Procesor direktno može da izvršava logičke operacije nad bitovima digitalno zapisanih podataka, dok se aritmetičke operacije nad brojevima takođe svode na primitivne logičke operacije. Za verifikaciju korektnosti hardvera (na primer, za dokazivanje ekvivalentnosti neoptimizovane i optimizovane varijante nekog logičkog kola) koriste se *SAT rešavači* — programi koji veoma efikasno mogu da provere da li je data logička formula zadovoljiva (pri čemu ta formula može biti ogromna i sadržati i stotine hiljada promenljivih).

Predikatska logika[1] umesto iskaznih slova razmatra i unutrašnju strukturu iskaza — svaki iskaz se dobija primenom neke relacije (unarne, binarne, ternarne itd.) na termove dobijene primenom funkcija na promenljive i konstante. Pri tom se prilikom izgradnje formula pored iskaznih veznika dopušta i primena kvantifikatora ($\forall$ i $\exists$). Na primer,

[1] Najčešće se razmatra tzv. predikatska logika prvog reda.

$$\neg(\forall x)(P(x)) \Leftrightarrow (\exists x)(\neg(P(x))).$$

Predikatska logika se koristi za formalnu specifikaciju i verifikaciju softvera. Postoje tesne veze između upitnih jezika baza podataka i predikatske logike prvog reda. Ova logika predstavlja i teorijsku osnovu logičkog programiranja, koje se realizuje programskim jezicima kakav je Prolog.

Matematička logika se u veštačkoj inteligenciji koristi za reprezentaciju znanja, izvođenje zaključaka i izgradnju sistema zasnovanih na znanju i rezonovanju. Oblast veštačke inteligencije koja se bavi razvojem algoritama i sistema sposobnih da automatski donose zaključke na osnovu unapred definisanih pravila i podataka naziva se automatsko rezonovanje.

Računari danas igraju važnu ulogu u matematičkoj logici, pomažući u rešavanju složenih logičkih problema. Upotreba računara u ovoj oblasti značajno je unapredila istraživanja i omogućila primene koje su ranije bile nezamislive. Računari se koriste za automatsko i interaktivno dokazivanje teorema i analizi različitih logičkih sistema i aksiomatskih teorija.

10.1.2 Linearna algebra

Linearna algebra je grana matematike koja se proučava vektore i vektorske (tj. linearne) prostore, matrice, linearne transformacije, sisteme linearnih jednačina i slično.

Vektor je n-torka (obično realnih) brojeva koji predstavljaju tačku u prostoru (brojevi su koordinate tačke). Vektori se mogu množiti skalarom tako što se svaki element pomnoži skalarom. Dva vektora se sabiraju tako što im se odgovarajuće koordinate saberu. Vektorima se predstavljaju različiti podaci u računarstvu. Jasno je da se svaka tačka u ravni i ili tačka u prostoru u sistemima za računarsku grafiku predstavlja dvodimenzionalnim ili trodimenzionalnim vektorom (na primer, vektor $[2,3]$ predstavlja koordinate tačke u ravni tj. u dvodimenzionalnom prostoru). Vektorima se predstavljaju i komplikovaniji podaci. Na primer, sve zaključne ocene učenika se mogu predstaviti vektorom, u sistemu za automatsku klasifikaciju cvetova, svaki cvet se može opisati vektorom koji sadrži dužinu čašice, širinu čašice, dužinu latice i širinu latice.

Vektorski (linearni) prostori su algebarske strukture koje sadrže skup vektora koji se se mogu sabirati i množiti skalarom, uz uobičajene algebarske zakone koje ove operacije zadovoljavaju. Osnovni primeri su prostor vektora u geometriji i prostor vektora predstavljenih n-torkama brojeva (on se naziva $\mathbb{R}^n$), međutim, razmatraju se i mnogi drugi oblici vektorskih prostora (na primer, vektorski prostor formiran od polinoma ili od realnih funkcija).

Matrica je pravougaona tabela brojeva raspoređenih u redove i kolone. Na primer, rasterske slike se predstavljaju matricama piksela – crno bele slike se predstavljaju jednom matricom, a slike u boji sa tri matrice (po jednom za crvenu, zelenu i plavu boju, u skladu sa modelom RGB). Sve zaključne ocene svih učenika neke škole se mogu predstaviti matricom (matricu tada posmatramo kao niz vektora).

Linearne transformacije su funkcije koje slikaju vektore u nove vektore, tako da se je slika linearne kombinacije vektora linearna kombinacija njihovih slika tj. tako da je $f(k_1\vec{a} + k_2\vec{b}) = k_1 f(\vec{a}) + k_2 f(\vec{b})$. Na primer, skaliranje vektora i rotacija vektora su linearne transformacije. Svaka linearna transformacija se može predstaviti kao proizvod matrice (koja odgovara linearnoj transformaciji) i vektora. Tako se, na primer, pomeranja objekata ili kamere u sistemima za računarsku grafiku (na primer, video igrama) implementiraju množenjem matrica i vektora koordinata (što hardver vrši veoma efikasno). Razne analize slika (na primer, zamućivanje ili detekcija ivica) se vrše primenom linearnih transformacija na delove slike.

Metode linearne algebre se primenjuju i na rešavanje sistema linearnih jednačina. Naime, sistemi linearnih jednačina se mogu predstaviti u obliku $Ax = b$, gde je A matrica sistema, b vektor slobodnih članova, a x vektor koji sadrži nepoznate. Rešavanje velikih sistema (koji sadrže veliki broj nepoznatih i jednačina) je praktično nezamisliva bez korišćenja računara. Prilikom rešavanja ovih sistema potrebno je koristiti metode numeričke linearne algebre koje osiguravaju taykozvanu stabilnost tj. koje osiguravaju da se usled nepreciznosti zapisa brojeva u računaru ne dobiju rešenja sistema koja previše odstupaju od stvarnih.

I naprednije tehnike linearne algebre se koriste u računarstvu. Na primer, tehnika *singularne dekompozicija* (engl. singular value decomposition) se koristi za kompresiju slika. U mašinskom učenju se koristi *analiza glavnih komponenata* (engl. principle component analysis) u cilju smanjenja dimenzionalnosti podataka. Tom metodom se uočavaju grupe međusobno povezanih podataka (na primer, veličina i težina jabuke) i skupovi podataka se mogu redukovati tako da se ne izgube informacije koje su njima predstavljene, a da se njihova analiza olakša.

Računari su neizostavan alat u linearnoj algebri, omogućavajući efikasno rešavanje velikih i složenih problema koji uključuju vektore, matrice i linearne sisteme jednačina. Korišćenje računara u ovoj oblasti nije samo ograničeno na ubrzavanje računanja, već omogućava analize koje bi bile nemoguće ili previše zahtevne za ručno izvođenje. Na primer, zahvaljujući računarima moguće je rešiti sisteme sa stotinama jednačina i promenljivih.

Elektronska verzija (2024)

10.1.3 Geometrija

Geometrija je grana matematike koja se bavi proučavanjem oblika, veličina, relativnih položaja figura i svojstava prostora. Osnovni cilj geometrije je istraživanje i razumevanje prostornih odnosa i osobina geometrijskih objekata kao što su tačke, prave, površine i zapremine. Euklidska geometrija proučava ravne i trodimenzionalne prostore prema principima koje je postavio Euklid. Neeuklidska geometrija istražuje prostore sa drugačijim osobinama od onih u euklidskom prostoru, kao što su hiperbolička i eliptična geometrija. Analitička geometrija koristi algebarske metode za proučavanje geometrijskih figura, povezujući geometriju sa algebrom kroz korišćenje koordinatnih sistema. Ona omogućava precizno opisivanje geometrijskih oblika koristeći jednačine i funkcije. Diferencijalna geometrija koristi tehnike diferencijalnog računa i algebre za proučavanje glatkih oblika, krivina i površina. Ona igra ključnu ulogu u razumevanju zakrivljenih prostora i nalazi primenu u fizici. Geometrija je fundamentalna disciplina koja povezuje mnoge oblasti matematike i pruža alate za rešavanje širokog spektra problema u nauci i inženjerskim disciplinama.

Najznačajniju primenu u računarstvu ima analitička geometrija. Analitička geometrija predstavlja osnovu računarske grafike. Tačke se predstavljaju koordinatama, a vektori i operacije nad vektorima (sabiranje, oduzimanje, množenje skalarom, skalarni i vektorski proizvod) se koriste u sklopu raznih geometrijskih primitiva (na primer, proveru da li su tačke kolinearne, izračunavanje površine, određivanje preseka duži, određivanje orijentacije trojke tačaka) koje se dalje koriste za izgradnju složenijih algoritama (na primer, određivanje da li tačka pripada mnogouglu, određivanje preseka mnogouglova, određivanje konveksnog omotača skupa tačaka i slično).

Umesto klasičnih Dekartovih koordinata uvode se i koriste *homogene koordinate*, koje predstavljaju osnovu analitičke projektivne geometrije. Umesto koordinata (x, y) u ravni koriste se koordinate (kx, ky, k), za $k \neq 0$. To znači da su tačke klase ekvivalencije međusobno proporcionalnih trojki koordinata. Na primer, tačka $(1, 3)$ se može predstaviti koordinatama $(1, 3, 1)$, ali i $(2, 6, 2)$ ili $(-0,5, -1,5, -0,5)$. Obično se ovako dobijene koordinate normalizuju na oblik $(x, y, 1)$, deljenjem poslednjom koordinatom. Izuzetak predstavljaju tačke oblika $(x, y, 0)$, koje se ne mogu normalizovati na ovaj način i koje ne odgovaraju običnim tačkama. Smatra se da su ovo koordinate beskonačno dalekih tačaka. Korišćenjem homogenih koordinata izbegava se analiza specijalnih slučajeva (na primer, prilikom određivanja preseka dve prave, ako su prave paralelne, presek neće postojati, tj. biće beskonačno daleka tačka). Takođe se postiže da se uobičajene geometrijske transformacije objekata (translacija, rotacija i slično) mogu predstaviti množenjem koordinata matricama, što je operacija koju hardver (grafički procesori) izvršavaju veoma brzo. Na primer, rotacija za ugao θ oko koordinatnog početka se može postići množenjem homogenih koordinata matricom

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

dok se translacija za vektor (v_x, v_y) može postići množenjem homogenih koordinata matricom

$$\begin{pmatrix} 1 & 0 & v_x \\ 0 & 1 & v_y \\ 0 & 0 & 1 \end{pmatrix}.$$

Osim u softveru za računarsku grafiku i video igre, geometrija se koristi u softveru za 2D i 3D modeliranje (computer aided design, CAD), softveru za 3D animacije, geografskim informacionim sistemima (GIS) i slično.

Dakle, računari se koriste za kreiranje geometrijskih modela u različitim oblastima (na primer, trodimenzionalnih modela u arhitekturi i mašinstvu). Vizualizacija (naročito u trodimenzionalnom prostoru) je nezamisliva bez primene računara. Računarska geometrija je oblast koja se bavi razvojem algoritama za rešavanje problema u geometriji (na primer, algoritama za triangulaciju kojom se složene površi razbijaju na trouglove, što je osnova za vizualizaciju i druge oblika analize površi). Geometrijski algoritmi se koriste i za prepoznavanje oblika (na primer, pravih linija ili kružnica na slici).

10.1.4 Teorija brojeva

Teorija brojeva je grana matematike koja proučava svojstva i odnose između celih brojeva. To je jedna od najstarijih i oblasti matematike, koja se fokusira na strukture i obrasce koji se javljaju u skupu celih brojeva. Posebno značajan je pojam deljivosti brojeva i klasifikacija brojeva na proste i složene. Teorija brojeva razmatra diofantske jednačine (ispituje se postojanje celobrojnih rešenja polinomskih jednačina). Po metodama proučavanja razlikuju se algebarska i analitička teorija brojeva. Algebarska teorija brojeva se bavi proučavanjem brojeva kroz algebarske strukture, kao što su brojevna polja i prstenovi i ona omogućava dublje razumevanje

kako se brojevi ponašaju u različitim matematičkim strukturama. Analitička teorija brojeva koristi alate iz analize, kao što su beskonačni redovi i integrali, za rešavanje problema vezanih za brojeve. Na primer, ζ-funkcija (zeta-funkcija) koju je uveo Riman koristi se za proučavanje prostih brojeva[2].

U računarstvu, teorija brojeva ima značajnu ulogu u razvoju algoritama, kriptografiji, generisanju slučajnih brojeva i mnogim drugim oblastima.

Kriptografija je verovatno najpoznatija oblast primene teorije brojeva u računarstvu. Mnoge moderne kriptografske tehnike oslanjaju se na teškoće određenih problema u teoriji brojeva. Na primer, RSA je jedan od najpopularnijih javnih kriptografskih sistema koji se oslanja na teškoću faktorizacije velikih prirodnih brojeva (naime, nije pronađen algoritam koji može u razumnom vremenu da pronađe proste činioce broja koji je proizvod dva prosta broja sa po oko 1000 cifara). Pošto su prosti brojevi ključni za mnoge kriptografske sisteme, postoje algoritmi kao što su Miller-Rabinov algoritam ili algoritam AKS koji se koriste za efikasnu proveru da li je broj prost.

Teorija brojeva se koristi u algoritmima za generisanje pseudoslučajnih (engl. pseudo-random) brojeva, koji su ključni za simulacije, kriptografiju i igre.

Teorija brojeva igra ulogu u algoritmima za kompresiju podataka i kodiranje. Na primer, kodovi zasnovani nad polinomima u konačnim poljima (Galoaova polja) se koriste za detekciju i ispravljanje grešaka u skladištenju i prenosu podataka.

Računari su postali neophodan alat u teoriji brojeva, omogućavajući matematičarima da rešavaju složene probleme, eksperimentišu sa velikim brojevima, proveravaju hipoteze, i razvijaju nove teoreme. Računaje sa velikim brojevima je nemoguće bez primena računara. Računari koriste metode za simulaciju i eksperimentisanje sa različitim raspodelama brojeva, što može pružiti statističke dokaze ili indikacije za matematičke tvrdnje. Neki dokazi u teoriji brojeva, zahtevaju značajne računske operacije koje su izvedene uz pomoć računara. Računari omogućavaju proveru velikog broja slučajeva koji bi bili previše kompleksni za ručno računanje.

10.1.5 Kombinatorika i teorija grafova

Kombinatorika je grana matematike koja proučava načine na koje se objekti mogu kombinovati, rasporediti ili organizovati. Najčešće se razmatra određivanje broja različitih mogućnosti (na primer, na koliko načina se iz bubnja od 39 loptica za loto može izvući 7 loptica) ili sistematično nabrajanje tih mogućnosti (na primer, nabrojati sve moguće načine da se 5 poslova rasporedi na 5 procesora tako da svaki procesor izvršava jedan posao). Najšeće se razmatraju kombinacije (odabir nekoliko elemenata skupa, bez obzira na redosled), varijacije (odabir nekoliko elemenata skupa i njihovo raspoređivanje u određeni redosled) i permutacije (raspoređivanje elemenata skupa u određeni redosled).

U računarstvu, kombinatorika igra ključnu ulogu u rešavanju problema koji uključuju grafove, drveta i slične kombinatorne strukture.

Teorija raspodele i kombinatornog dizajna koristi se za rešavanje problema raspodele resursa i dizajniranje algoritama sa specifičnim karakteristikama. Takvi su, na primer, algoritmi raspodelu poslova u paralelnim i distribuiranim sistemima. Kombinatorika se koristi i u testiranju softvera tako što se teži generisanju testova koji pokrivaju sva moguća izvršavanja programa.

Čest oblik optimizacije je tzv. kombinatorna optimizacija koja podrazumeva da se među mnogim kobinatornim objektima koji zadovoljavaju neka svojstva pronađu oni koji imaju najveću ili najmanju vrednost neke pridružene statistike. Na primer, problem trgovačkog putnika zahteva da se od svih mogućih redosleda obilazaka nekog skupa gradova (svih permutacija tog skupa) pronađe onaj u kom trgovački putnik prelazi najmanji put.

Drveta i grafovi su kombinatorni objekti koji imaju ogromne primene u računarstvu. Drveta se često koriste za efikasno smeštanje podataka (na primer, pretraživačka drveta se koriste u programima za predstavljanje skupova i mapa, B-drveta se koriste u bazama podataka za efikasnu pretragu). Drveta se, na primer, koriste u kompilatorima da bi se pročitani program predstavio u memoriji na način koji se dalje može efikasno obrađivati u cilju generisanje izvršivog mašinskog programa.

Grafovi su kombinatorni objekti koji se sastoje od čvorova međusobno povezanih granama (u usmerenim grafovima, grane su usmerene tj. jednosmerne, dok su u neusmerenim grafovima grane neusmerene tj. dvosmerne). Mnogi problemi se mogu modelovati grafovima. Na primer, grafovi modeliraju računarske mreže gde čvorovi predstavljaju uređaje (računare, rutere) a grane mrežne veze između njih. Algoritmi za rutiranje, kao što su Dijkstrin i Bellman-Fordov, koriste se za pronalaženje najkraćih puteva između čvorova. Takvi algoritmi za pronalaženje najkraćih puteva se koriste i u drugim srodnim problema. Na primer, u veštačkoj inteligenciji se algoritam A* koristi se za navigaciju robota ili vozila. Algoritmi za analizu grafova koriste se za otkrivanje

[2] Rimanova hipoteza se bavi ispitivanjem nula Rimanove ζ-funkcije definisane u kompleksnoj ravni analitičkim produženjem funkcije $\zeta s = \sum_{n=1}^{+\infty}$ i smatra se najznačajnijim problemom savremene matematike. Poznata anegdota kaže je da je čuveni matematičar David Hibert izjavio da bi mu prvo pitanja ako bi ga digli iz groba za nekoliko stotina godina bilo da li je u međuvremenu rešena Rimanova hipoteza.

Elektronska verzija (2024)

struktura i obrazaca unutar podataka. Algoritmi za otkrivanje zajednica se koriste se za grupisanje čvorova u društvenim mrežama. Grafovi modeliraju strukturu interneta gde čvorovi predstavljaju veb-stranice, a grane hiperveze između njih. Algoritam PageRank, koji su razvili osnivači kompanije Google, koristi grafove za rangiranje veb-stranica na osnovu njihove povezanosti. U kompilatorima se grafovima modelira tok izvršavanja programa gde čvorovi predstavljaju osnovne blokove koda (blokove naredbi koje se sekvencijalno izvršavaju), a grane prelaze između blokova (grane nastaju usled korišćenja grananja i petlji u programima). Grafovima se modeliraju i zavisnosti između različitih delova programa, što omogućava bezbednu optimizaciju.

Računari omogućavaju rešavanje složenih kombinatornih problema, koji bi inače bili izuzetno teško ili nemoguće rešivi ručno. Koriste se za nabrajanje i prebrojavanje određenih kombinatornih objekata, za pronalaženje predstavnika klasa neizomorfnih objekata i slično. Ovo može pomoći u testiranju i formulisanju hipoteza, pa čak i dokazivanju nekih teorema metodama iscrpnog kombinatornog nabrajanja (na primer, teorema o obojivosti planarnih grafova sa 4 boje je skoro 200 godina predstavljala otvoren matematički problem, dok nije dokazana pomoću računara). Računari pružaju softver za vizualizaciju grafova, mreža i drugih kombinatornih struktura, što pomaže u analizi i interpretaciji podataka.

10.1.6 Verovatnoća i statistika

Verovatnoća i statistika su dve povezane grane matematike koje se bave proučavanjem nasumičnih događaja, analize podataka, i donošenja zaključaka na osnovu tih podataka. Iako su međusobno povezane, svaka od ovih disciplina ima svoje specifične ciljeve i metode. Verovatnoća se bavi proučavanjem i kvantifikacijom slučajnih događaja. Na primer, moguće je proceniti koliko je verovatno da se bacanjem kokcice za jamb dva puta uzastopno dobije broj 1 ili da se izborom nasumičnog elementa niza brojeva odabere baš najmanji element niza). Statistika se bavi prikupljanjem, analizom, interpretacijom i prezentacijom podataka. Statistika je od ključne važnosti za donošenje zaključaka o velikim populacijama na osnovu uzoraka podataka (na primer, na osnovu efekta nekog leka na uzorku testiranih pacijenata, donose se odluke o široj upotrebi tog leka).

Verovatnoća i statistika igraju ključnu ulogu u mnogim oblastima računarstva, pružajući alate za modelovanje i analizu podataka. Verovatnoća i statistika su osnova mnogih algoritama mašinskog učenja, gde se koriste za analizu podataka, pravljenje predviđanja i donošenje odluka.

Regresija je statistička tehnika koja se koristi za modelovanje odnosa između zavisne i jedne ili više nezavisnih promenljivih. Na primer, linearna regresija se može upotrebiti za predviđanje cene nekretnine na osnovu kvadrature i lokacije. *Klasifikacija* je tehnika kojom se ulazni podaci svrstavaju u jednu ili više kategorija. Na primer, naivni Bajesov klasifikator koristi Bajesovu teoremu iz oblasti verovatnoće i statistike za klasifikaciju neželjene pošte (engl. spam). *Klasterovanje* je tehnika kojom se slični podaci grupišu, bez unapred definisanih kategorija. Na primer, klasterovanjem se mogu detektovati grupe ljudi sa sličnim ponašanjem na društvenim mrežama.

Prilikom analize podataka koriste se deskriptivne statistike (na primer, prosek, medijana, standardna devijacija). Metode za testiranje hipoteza pokazuju da li postoji dovoljno dokaza u uzorku podataka da se podrži određena hipoteza o populaciji. Na primer, t-test se može upotrebiti za poređenje rezultata dve različite grupe studenata na istom testu i ovim testom se može proceniti kolika je verovatnoća da je razlika u rezultatu rezultat slučajnosti, a koliko da je jedna grupa zaista bolja od druge.

Važnu klasu algoritama čine *probabilistički algoritmi*, zasnovani na generisanju nasumičnih (pseudo-slučajnih) brojeva. *Las Vegas* algoritmi uvek dovode do tačnog rezultata, ali njihovo vreme izvršavanja i može varirati u zavisnosti od izabranih brojeva. Na primer, jedan od najčuvenijih algoritama uopšte je algoritam brzog sortiranja (engl. quick sort) koji nasumično bira vrednost na osnovu koje će podeliti niz na elemente koji su manji od nje i veći od nje i zatim će sortirati svaki od dva tako dobijena dela niza. Ako se prilikom svakog izbora ta vrednost nalazi blizu sredine niza, sortiranje će biti efikasno (složenost će mu biti $O(n \log n)$), ali ako se često događa da nasumično određene vrednosti dele niz na neravnomerne delove, sortiranje može biti veoma neefikasno (složenost će mu biti $O(n^2)$).

Monte Karlo algoritmi imaju garantovano vreme izvršavanja, međutim, ne postoji garancija da će krajnji rezultat biti tačan (iako se obično sa velikom verovatnoćom može tvrditi da hoće, postoji mala verovatnoća da će postojati netrivijalno odstupanje između dobijenog i tačnog rezultata). Na primer, Fermaov test za ispitivanje da li je broj prost je zasnovan na maloj Fermaovoj teoremi koja tvrdi da je za bilo koji prost broj p i ceo broj a koji nije deljiv sa p broj $a^{p-1} - 1$ deljiv sa p. Test za dati broj p proverava određen broj vrednosti a. Ako se za neko a dobije da $a^{p-1} - 1$ nije deljiv sa p, tada p sigurno nije prost. Ako se ispita određen broj vrednosti a i uvek se dobije da je $a^{p-1} - 1$ deljiv sa p, sa jako velikom verovatnoćom se može tvrditi da je p prost broj.

Verovatnoća se koristi i za analizu sigurnosti kriptografskih algoritama i protokola. U računarskoj grafici koriste se stohastički algoritmi koriste za simulaciju prirodnih fenomena kao što su vatrometi, voda i oblaci.

Računari igraju ključnu ulogu u statistici, omogućavajući analizu velikih količina podataka, primenu složenih statističkih metoda i vizualizaciju rezultata. Statistički softver omogućava primenu širokog spektra statističkih

tehnika za analizu podataka na velikim količinama podataka.

10.1.7 Numerička matematika i optimizacija

Numerička matematika se bavi razvojem i analizom algoritama za rešavanje matematičkih problema koji se ne mogu lako rešiti analitički, tj. bez numeričkih metoda. Fokus je na pružanju približnih rešenja za probleme koji su previše složeni za tačno rešenje ili gde su tačni rezultati nepraktični.

Razmatraju se metode za približno rešavanje jednačina (na primer, jednačina $\cos x = x$ se ne može rešiti analitički, ali se numeričkim metodama može pronaći približno rešenje $x = 0.739085\ldots$).

Numerička matematika omogućava približno izračunavanje određenih integrala (čak i kada nije moguće u analitičkom obliku odrediti primitivnu funkciju funkcije koja se integrali), izračunavanje vrednosti izvoda funkcija, približno rešavanje linearnih i nelinearnih jednačina (tj. određivanje nula funkcija) i numeričko rešavanje diferencijalnih jednačina (tj. određivanje vrednosti rezultujuće funkcije u nekoj tački ili na datom skupu tačaka).

Numeričkim metodama se vrši i interpolacija i aproksimacija funkcija tj. određivanje funkcija (često polinoma) na osnovu poznatih vrednosti $f(x_i)$ na nekom skupu tačaka x_i (interpolacijom se određuje funkcija iz date klase funkcija koja ima baš zadate vrednosti, dok se aproksimacijom određuje funkcija koja ima vrednosti koje su približne zadatim (ali globalno gledano bolje opisuje pojavu koja se modeluje).

Numeričke metode se primenjuju i za optimizaciju tj. za pronalaženje minimuma ili maksimuma funkcija. Mnogi problemi iz realnog života se formulišu i rešavaju kao optimizacioni problemi (na primer, pronalaženje rasporeda časova ili pronalaženje optimalne rute dostavnih vozila).

Jedan od važnih zadataka numeričke matematike je da obezbedi metode koje će biti numerički stabilne tj. koje će obezbediti da se u uslovima zapisa brojeva samo sa određenom preciznošću (u računaru se brojevi obično zapisuju u pokretnom zarezu, sa fiksiranim brojem bitova) dobiju rezultati koji su dovoljno precizni da bi bili praktično upotrebljivi.

Računari se primenjuju za mnoga izračunavanja u tehnici i nauci (engl. scientific computing) i u svim tim primenama veoma važnu ulogu ima numerička matematika. Sa druge strane, od svog samog početka numerička matematika je nezamisliva bez primene računara.

10.1.8 Teorijsko računarstvo

Teorijsko računarstvo je grana računarstva koja se bavi proučavanjem osnovnih principa i granica računarskih sistema. Njegov cilj je da razvije formalne modele i metode za analizu računarskih algoritama, računskih problema, kao i mogućnosti i ograničenja računara. Teorijsko računarstvo kombinuje matematičke metode i logiku kako bi dalo odgovore na ključna pitanja kao što su:

- Koji se problemi mogu rešiti uz pomoć računara?
- Koliko resursa (vremena, memorije) je potrebno za rešavanje tih problema?

Metode i principi teorijskog računarstva omogućavaju istraživačima i inženjerima da razvijaju efikasne algoritme, razumeju složenost problema i istražuju granice onoga što računari mogu postići. Kroz ovu disciplinu, računarstvo se razvija kao naučna oblast, povezujući matematiku, logiku i praktične aspekte tehnologije.

Da bi se ustanovilo šta računari mogu, a šta ne mogu izračunati, uvode se formalni modeli izračunavanja (na primer, Tjuringove mašine, URM mašine i slično). Pored ovih opštih modela izračunavanja koji mogu da realizuju svaki algoritam, koriste se i specijalizovani modeli koji imaju manju izražajnost, ali se koriste u nekim drugim specijalizovanim domenima. Na primer, konačni automati (engl. finite automata) se uspešno koriste u leksičkoj analizi teksta, pretrazi teksta u modelovanju hardvera i slično. Potisni automati (engl. pushdown automata) proširuju konačne automate stek-memorijom i čine osnovu sintaksičke analize (engl. parsing) računarskih programa, koriste se u verifikaciji programa i slično.

10.1.9 Algoritmi i složenost izračunavanja

Računarski programi su zasnovani na algoritmima — preciznim opisima postupaka izračunavanja tj. transformacije podataka predstavljenih brojevima. Algoritmi su fundamentalni koncepti u računarstvu i koriste se u svim aspektima razvoja softvera (od osnovnih algoritama, poput sortiranja, pa do naprednih algoritama, koji se koriste u savremenim računarskim sistemima za rešavanje specifičnih problema). Međutim, matematika se hiljadama godina bavi izučavanjem takvih postupaka. Na primer, još su stari Grci opisali i koristili čuveni Euklidov algoritam za određivanje najvećeg zajedničkog delioca dva broja. Zato se slobodno može smatrati da konstrukcija i analiza algoritama objedinjavaju matematička i računarska znanja.

Razumevanje algoritama omogućava efikasno rešavanje problema i razvoj naprednih softverskih sistema. Matematičke tehnike pružaju alatke za analizu i razumevanje algoritama, što je ključno za napredak u oblasti računarstva.

Ispitivanje i dokazivanje *korektnosti algoritama*, kao i verifikacija softvera se vrše tradicionalnim matematičkim metodama. Bez obzira da li se analiza vrši neformalno ili formalno, potrebno je da se precizno specifikuje problem koji se rešava, da se da precizan opis algoritma na odgovarajućem nivou apstrakcije, da se precizno formulišu teoreme koje garantuju korektnost i da se one dokažu. Formalna analiza korektnosti podrazumeva da se sve ovo izvrši u nekom formalnom logičkom okviru (na primer, u Horovoj logici), a često se za proveru korektnosti dokaza koriste specijalizovani računarski programi (tzv. interaktivni dokazivači teorema), čime se garantuje maksimalna pouzdanost.

Analiza *složenosti algoritama* proučava efikasnost algoritama u smislu resursa, tj. vremena i prostora (memorije) potrebnog za njihovo izvršavanje. Matematički pojmovi kao što su asimptotska notacija (O-notacija, Ω-notacija, Θ-notacija) koriste se za izražavanje složenosti algoritama. Na primer, ako algoritam koji sortira niz dužine n može to da uradi uz najviše $7n^2 + 4n + 5$ koraka, bez obzira na to kakvi su elementi niza, reći ćemo da je da ovaj algoritam kvadratne složenosti tj. da pripada klasi $O(n^2)$. Formalno, za niz $f(n)$ kažemo da pripada klasi $O(g(n))$ ako postoje konstante C i n_0 tako da za svako $n \geq n_0$ važi da je $f(n) \leq C \cdot g(n)$ — počevši od neke vrednosti n_0 niz f je odozgo ograničen nizom g (eventualno skaliranim nekim faktorom C).

Problemi se klasifikuju prema resursima potrebnim za njihovo rešavanje. Na primer, problemi u klasi P su oni za koje su pronađeni efikasni algoritmi, čije vreme izvršavanja polinomski zavisi od veličine ulaza, *NP*-kompletni problemi su oni čija se rešenja mogu proveriti u polinomskom vremenu, ali još nije pronađen algoritam polinomske složenosti za pronalaženje tih rešenja, niti je dokazano da takav algoritam ne postoji.

10.2 Matematički softver

Matematika pruža čvrste teorijske i metodološke osnove računarstvu. Sa druge strane, kao što smo već videli, računari matematici pružaju efikasan alat za vršenje numeričkih, ali i simboličkih izračunavanja. U novije vreme računari se koriste i za dokazivanje matematičkih teorema i za proveru matematičkih dokaza, što doprinosi njihovoj pouzdanosti i približava nas hiljadugodišnjim idealima generacija i generacija matematičara.

10.2.1 Sistemi za računarsku algebru

Sistemi za računarsku algebru (engl. computer algebra systems, CAS) su softverski alati dizajnirani za automatsko rešavanje problema iz simboličke matematike. Oni omogućavaju manipulisanje matematičkim izrazima na simbolički način, za razliku od numeričkog rešavanja. U stanju su da izvode operacije nad polinomima i algebarskim izrazima, kao što su faktorizacija, pojednostavljivanje i razvijanje izraza. rešavanje jednačina. Ovi sistemi mogu rešavati linearne i nelinearne jednačine i sisteme jednačina simbolički, što znači da rezultat može biti izraz, a ne samo broj. Mogu se koristiti za izračunavanje izvoda i pronalaženje određenih i neodređenih integrala. Sistemi za računarsku algebru omogućavaju rešavanje običnih i parcijalnih diferencijalnih jednačina na simbolički način.

Najpopularniji sistemi za računarsu algebru su Wolfram Mathematica, Maple, SageMath, Maxima itd.

Ovi sistemi imaju primenu u obrazovanju, u naučnim istraživanjima u matematici, fizici i inženjerskim disciplinama, kao i u industriji.

10.2.2 Softver za numerička izračunavanja

Sistemi za numerička izračunavanja su softverski alati dizajnirani za izvođenje matematičkih proračuna koji zahtevaju približna rešenja, posebno za probleme koji se ne mogu rešiti analitički (simbolički). Ovi sistemi su ključni za primenu numeričkih metoda u inženjerskim disciplinama, fizici, ekonomiji i drugim naučnim disciplinama. Numeričkim metodama mogu približno da rešavaju linearne i nelinearne jednačine, mogu da pronalaze približne vrednosti izvoda i određenih integrala, kao i približna rešenja diferencijalnih jednačina, da približno rešavaju optimizacione probleme (pronalaze minimume i maksimume) i slično. Ovi sistemi često koriste rad sa matricama i podržavaju efikasno izvršavanje različitih operacija nad matricama (množenje, pronalaženje inverzne matrice, determinante, sopstvenih vrednosti i slično). Često imaju određene mogućnosti i 2d i 3d vizualizacije.

Najpopularniji sistemi za numerička izračunavanja su MATLAB, GNU Octave, Julia itd.

10.2.3 Statistički softver

Statistički softver je specijalizovan softver za analizu podataka, primenu statističkih metoda i vizualizaciju rezultata. Ovaj softver omogućava istraživačima, analitičarima i profesionalcima u raznim oblastima da efikasno

Elektronska verzija (2024)

obrađuju velike količine podataka i donose zaključke zasnovane na tim analizama. Softver podržava izračunavanje osnovnih statističkih mera kao što su srednje vrednosti, medijana, standardna devijacija, varijansa i percentili, primenu statističkih testova, linearnu i nelinearnu regresiju, analizu vremenskih serija itd. Softver podržava i pravljenje različitih grafikona, histograma i drugih oblika vizualizacije podataka i statistika.

Najpopularniji statistički softver danas je R, SPSS, SAS, Stata itd.

10.2.4 Softver za analizu i vizualizaciju podataka

Softver za analizu i vizualizaciju podataka omogućava korisnicima da obrađuju, analiziraju i interpretiraju velike količine podataka, kao i da rezultate tih analiza prikažu na vizualno privlačan i razumljiv način. Primećujemo da postoje određene sličnosti između ovog tipa softvera i statističkog softvera, ali svaki ima svoje karakteristične domene primene. Ovi alati su ključni za donošenje informisanih odluka u različitim oblastima, uključujući poslovanje, nauku, zdravstvo, inženjerske discipline i slično. Ovaj softver ima alate za uklanjanje nepotpunih ili netačnih podataka, normalizaciju podataka, i agregaciju, za izračunavanje osnovnih statističkih mera, ali i za naprednu analitiku, korišćenjem metoda mašinskog učenja poput klasifikacije, regresije, i klasterovanja. Softver pruža podrška za analizu i vizualizaciju podataka iz različitih izvora, uključujući relacione baze podataka, Excel tabele, i skladišta u oblaku, a posebno je optimizovan za rad sa velikim količinama podataka, omogućavajući brzo pretraživanje, filtriranje i analizu. Softver poseduje i mogućnost kreiranja i automatskog ažuriranja izveštaja koji prikazuju ključne metrike i trendove promene podataka u realnom vremenu. Važna oblast primene je *poslovna inteligencija* koja podrazumeva praćenje poslovnih performansi, analizu tržišta i donošenje strateških odluka na osnovu podataka.

Najpopularniji alati ovog tipa su Tableau, Power BI, Google Data Studio, ali i jezik Python opremljen posebnim bibliotekama (NumPy, SciPy, Matplotlib, pandas).

10.2.5 Geometrijski i grafički softver

Geometrijski softver služi za kreiranje i analizu geometrijskih oblika i grafičkih prikaza. Softver najčešće nudi alatke za crtanje 2D oblika (kao što su trouglovi, kvadrati, krugovi) i 3D objekata (kao što su kocke, sfere, piramide) i alatke za geometrijske transformacije (rotaciju, translaciju, skaliranje i refleksiju geometrijskih objekata). Ovaj softver omogućava korisnicima da koriste osnovne alatke poput lenjira i šestara za konstrukciju geometrijskih figura na ekranu, što je naročito značajno u matematičkom obrazovanju. Objekti se mogu konstruisati akcijama mišem (geometrijski) ili zadavanjem formula (analitički, tj. algebarski). Softver obično podržava i iscrtavanje grafika funkcija jedne promenljive (u 2D) i dve promenljive (u 3D), kao i crtanje parametarski zadanih krivi i površi. Ovi programi često imaju i određene mogućnosti simboličkog i numeričkog izračunavanja.

Najpoznatiji programi ovog tipa su Geogebra, Desmos, Cabri, Cinderella itd.

10.2.6 Softver za optimizaciju

Softver za optimizaciju se koristi za rešavanje složenih optimizacionih problema u kojima se traži maksimalna vrednost funkcije cilja (na primer, da donesu kompaniji što veći profit), poštujući zadata ograničenja (na primer, koristeći ograničenu količinu resursa). U zavisnosti od toga kakvi su tipovi promenljivih i da li su funkcija cilja i ograničenja linearne ili ne, ovi problemi se rešavaju korišćenjem linearnog programiranja (LP), celobrojnog programiranja (IP), mešovitog celobrojnog programiranje (MIP) ili nelinearnog programiranja (NLP). Ovi alati su ključni za donošenje optimalnih odluka u logistici i transportu, finansijama, energetici i drugim oblastima. Najpoznatiji programski paketi ovog tipa su Gurobi Optimizer, IBM ILOG CPLEX, Mosek i drugi.

10.2.7 Softver za dokazivanje teorema i formalnu verifikaciju programa

SAT (Boolean Satisfiability) i *SMT* (Satisfiability Modulo Theories) rešavači su specijalizovani alati za rešavanje logičkih problema koji se koriste u mnogim oblastima računarstva, kao što su formalna verifikacija, automatsko dokazivanje teorema, optimizacija, i analiza softverskih sistema. SAT rešavači rešavaju problem iskazne zadovoljivosti i proveravaju da li postoji vrednost za promenljive koja čini datu iskaznu formulu tačnom. SAT problem je NP-kompletan, što znači da još nije pronađen efikasan algoritam (polinomske složenosti) koji ga rešava, ali postoje vrlo efikasni heuristički algoritmi za rešavanje mnogih praktičnih instanci. SMT je varijanta SAT problema u kojoj se umesto iskaznih slova koriste formule određenih teorija logike prvog reda. SAT rešavači se više koriste u verifikaciji hardvera, a SMT rešavači u verifikaciji softvera (često se koriste tako što su integrisani u šire sisteme za verifikaciju). Jedan od najpoznatijih SAT rešavača je MiniSAT, a SMT rešavača Microsoft Z3.

Automatsi dokazivači teorema se koristi za rešavanje logičkih problema i pronalaženje formalnih dokaza.

Ovaj softver je posebno dizajniran za rad sa formulama u logici prvog reda (engl. first-order logic) i koristi moćne algoritme za pretragu i zaključivanje, najčešće zasnovane na metodu rezolucije koji proveravaju valjanost datih formula. Koristi se u matematičkim istraživanjima (automatski konstruiše dokaze nekih matematičkih teorema), ali i formalnoj verifikaciji softvera i hardvera (često integrisan u verifikacione sisteme). Najpoznatiji automatski dokazivači teorema su Vampire, E prover, Prover 9.

Interaktivni dokazivači teorema pomažu korisnicima da formalno verifikuju matematičke teoreme ili svojstva softverskih sistema kroz interaktivni proces dokazivanja. Za razliku od potpuno automatskih dokazivača teorema, koji pokušavaju da automatski pronađu dokaz za zadatu teoremu, interaktivni dokazivači teorema omogućavaju korisniku da vodi proces dokazivanja, često koristeći kombinaciju automatizovanih tehnika i ručno zapisanih dokaza. Većina interaktivnih dokazivača teorema podržava logike višeg reda, omogućavajući korisnicima da izraze složene matematičke koncepte, funkcije, i predikate korišćenjem elemenata funkcionalnog programiranja. Iako korisnici vode proces dokazivanja, mnogi interaktivni dokazivači nude integraciju sa automatizovanim alatima (SAT/SMT rešavačima i automatskim dokazivačima za logiku prvog reda), koji olakšavaju izvođenje dokaza automatski dokazujući jednostavnije korake. Interaktivni dokazivači se koriste se za proveru da softverski sistemi ispunjavaju zadate specifikacije, posebno u kritičnim sistemima (na primer, u metro-sistemima, avijaciji, medicini). Takođe, pomažu matematičarima u formalnom dokazivanju teorema, osiguravajući da je svaki dokaz proveren i da u formalizaciji neke teorije ne može da postoji greška. Najpoznatiji interaktivni dokazivači teorema su Lean, Coq, Isbelle/HOL, HOL Light itd.

Elektronska verzija (2024)

Elektronska verzija (2024)

Glava 11

Veštačka inteligencija

Veštačka inteligencija je oblast računarstva koja je danas u velikoj ekspanziji. Skoro da nema oblasti ljudske delatnosti u kojoj veštačka inteligencija ne nalazi primene.

Veštačka inteligencija oslanja se na oblasti algoritmike, matematičke logike, numeričke matematike, matematičke analize, verovatnoće, statistike, itd.

Veštačka inteligencija kao informatička disciplina ustanovljena je na konferenciji *The Dartmouth Summer Research Conference on Artificial Intelligence* u Darmutu (Sjedinjene Američke Države), 1956. godine. Tom prilikom predloženo je, od strane Džona Makartija, i sâmo ime discipline, ne sasvim srećno, jer je to ime često izazivalo nedumice i podozrenje. Tokom godina koje su sledile, bilo je perioda optimizma i intenzivnog razvoja, ali i perioda pesimizma i opadanja interesovanja. Početkom dvadeset prvog veka javio se talas izuzetno uspešnih sistema zasnovanih na „dubokom učenju" – sistema koji su uspešno vršili prepoznavanje lica na fotografijama, prevođenje prirodnih jezika, navigaciju vozila i drugo. Ovaj uzlet praćen je velikim probojima 2020-ih godina u oblastima obrade prirodnog jezika (poput sistema ChatGPT) i računarskog vida (poput sistema DALL-E). Sada, dakle, traje period izuzetnog razvoja veštačke inteligencije i on će sigurno još potrajati.

Ne postoji jedna, opšteprihvaćena definicija veštačke inteligencije, ali se pod njom obično podrazumeva sposobnost mašinskog usvajanja, pamćenja i obrade određenih znanja. Većina metoda veštačke inteligencije usmerena je na uske, konkretne oblasti primene. Ipak, u poslednje vreme javljaju se i podoblasti veštačke inteligencije koje imaju za cilj opšte rasuđivanje u stilu čoveka.

Pomenimo dve, verovatno ključne, teme veštačke inteligencije:

- rešavanje problema u kojima se javlja *kombinatorna eksplozija*, tj. u kojima je broj mogućnosti toliko veliki da se ne može sistematično, tj. iscrpno ispitati u razumnom vremenu. Jedna takva oblast je, na primer, igranje igara kao što je šah.

- *zaključivanje*, kada je cilj iz raspoloživih podataka kreirati neke nove uvide, nova znanja. Jedan vid zaključivanja je *deduktivno zaključivanje* – to je zaključivanje zasnovano na rigoroznom matematičkom rasuđivanju i ide od opšteg ka pojedinačnom. Primer takvog problema[1] je izračunavanje dijagonale nekog pravougaonika – Pitagorina teorema daje opštu vezu između stranica pravouglog trougla i ona se može primeniti za izračunavanje dijagonale bilo kog konkretnog pravougaonika. Drugi vid zaključivanja je *induktivno zaključivanje* – to je zaključivanje zasnovano na mnoštvu raspoloživih pojedinačnih podataka iz kojih se *generalizacijom* mogu kreirati neki novi uvide, nova znanja. Primer takvog problema je prepoznavanje vrste životinje na slici, a na osnovu hiljada slika za koje je već poznato šta prikazuju.

Mnoge metode veštačke inteligencije imaju sličnu opštu strukturu procesa rešavanja problema. Faze rešavanja obično su: modelovanje, tj. opisivanje zadatog problema u strogim, matematičkim terminima; rešavanje problema opisanog u matematičkim terminima; interpretiranje i analiza rešenja. Dublja priroda ovih faza razlikuje se između različitih podoblasti veštačke inteligencije.

11.1 Uska i opšta veštačka inteligencija

Dominantan pravac razvoja veštačke inteligencije dugo je bio razvoj sistema specijalizovanih za konkretne zadatke. To je pravac koji se naziva „uska veštačka inteligencija". Alternativa je „veštačka opšta inteligencija" (eng. *artificial general intelligence*, *AGI*). Njen cilj je razvoj meta-algoritama koji mogu da pokreću sistem

[1] Kada se govori o „problemu", obično se misli na čitavu klasu srodnih zadataka. Pojedinačne zadatke koji pripadaju ovakvim klasama zovemo *instance problema* (ili *primerci problema*).

Elektronska verzija (2024)

sposoban da uči, rasuđuje i rešava sve probleme koje može i čovek. Taj cilj neki istraživači pokušavaju da postignu matematičkim modelovanjem ljudskog mozga, iako mnogi smatraju da je to teško dostižno ili nemoguće. Istraživač i futurista Rej Kercvajl (Raymond Kurzweil, trenutno radi u kompaniji Google), u svojoj uticajnoj knjizi „Singularnost je blizu” iz 2005. godine, tvrdi da nije daleko trenutak kada će računari prevazići čoveka u svim intelektualnim aktivnostima i procenjuje da će računari već 2029. godine moći da prođu Tjuringov test (videti poglavlje 11.2), što će, dalje, oko 2045. godine dovesti do „sveopšteg preokreta u ljudskim mogućnostima“. Skorašnji prodori u ovoj oblasti učvrstili su mnoge u sličnim uverenjima.

11.2 Filozofski i etički aspekti veštačke inteligencije

Neke podoblasti veštačke inteligencije imaju za cilj oponašanje prirodne (ljudske ili životinjske) inteligencije. No, postavlja se pitanje šta je uopšte *inteligencija*. Možemo smatrati da inteligencija podrazumeva sledeće sposobnosti: sposobnost pamćenja, skladištenja znanja i mogućnost njegove obrade, sposobnost učenja – usvajanja novih znanja, sposobnost komunikacije sa drugim inteligentnim bićima ili mašinama, itd. Može se smatrati, dakle, da biće ili mašina imaju atribute inteligentnog, ako imaju navedena svojstva. Alen Tjuring (Alan Turing) formulisao je sledeći znameniti test: ako su u odvojene dve prostorije smeštene jedna ljudska osoba i neki uređaj i ako na identična pitanja pružaju odgovore na osnovu kojih se ne može pogoditi u kojoj sobi je čovek, a u kojoj uređaj, onda možemo smatrati da taj uređaj ima atribute veštačke inteligencije. Tjuring je 1947. godine, govoreći o računarima koji simuliraju čovekovo rasuđivanje, predložio i kreiranje mašina „programiranih da uče i kojima je dopušteno da čine greške” jer, kako kaže, „ako se od mašine očekuje da bude nepogrešiva, onda ona ne može biti inteligentna“. Ovakav pristup, koji dozvoljava greške, karakterističan je za mašinsko učenje. U vezi sa otvorenim pitanjem šta je uopšte ljudska inteligencija (ako želimo da je oponašaju računari), zanimljiv je i stav fon Nojmana: „ako mi preceizno opišete šta je to što ne možе da uradi mašina, onda ću ja moći da napravim mašinu koja će raditi upravo to!“ Ovaj stav odnosi se na to da za svaku obradu podataka koju možemo da opišemo u terminima matematičkih izračunavanja, postoji računarski program koji može da je vrši.

Mnogi problemi veštačke inteligencije mogu se opisati u matematičkim terminima. Pitanje je za koje klase problema postoje opšti načini rešavanja (koje mogu da primene ljudi ili mašine). Rezultati Gedela i Tjuringa iz prve polovine dvadesetog veka pokazali su da postoje matematičke teorije (uključujući i jednostavne teorije kakva je aritmetika) koje su nepotpune i neodlučive. Na primer, postoje tvrđenja o prirodnim brojevima koja su tačna, ali se ne mogu dokazati iz aksioma aritmetike. Štaviše, skup aksioma aritmetike nije moguće proširiti tako da se to promeni. Dodatno, ne postoji algoritam koji može da dokaže svako aritmetičko tvrđenje koje jeste dokazivo. Ovi rezultati govore da za neke probleme ne mogu da postoje mašine koje mogu da reše svaku njihovu instancu, te da preostaje da se njima bave kao i ljudi: da koriste svojstva i zapažanja specifična za konkretne date instance. Drugim rečima, za instance ovakvih problema traganje za rešenjem neće uvek biti isto i neće garantovati uspeh.

Pored filozofskih pitanja o tome gde su granice ljudske i veštačke inteligencije, važna su i etička pitanja koja se odnose na mašine koje mogu da samostalno donose nekakve odluke. Još od ranih dana veštačke inteligencije postoji strah od „mašina koje misle“, a sa njihovim razvojem ta pitanja sve su češća a i ti strahovi jačaju: neki smatraju da ona donosi velike koristi, neki smatraju da od nje prete opasnosti, a neki veruje i u jedno i u drugo. Neke od najčešćih etičkih dilema odnose se na pitanja bezbednosti, cenzure, diskriminacije i privatnosti.

Jednostavn primer pitanja koje se tiče bezbednosti je vezan za autonomnu vožnju. Pitanje je kako treba definisati ponašanje vozila u slučaju da mora da ugrozi jednog od dva učesnika u saobraćaju. Još pre toga, pitanje je da li takve odluke prepustiti samom autonomnom sistemu, bez eksplicitne odluke ugrađene unapred. Ukoliko dođe do ugrožavanja bezbednosti učesnika u saobraćaju, povreda, ili štete, postavljaju se i mnoga dodatna, pravna pitanja, na primer – da li za štetu odgovara proizvođač autonomnog automobila ili čovek koji je u njemu sedeo. Mnoga ovakva pitanja će vrlo uskoro biti od praktičnog značaja, a odgovori na njih tek treba da se formulišu. Pored autonomne vožnje tu su i brojna pitanja vezana za vojne primene veštačke inteligencije.

Sistemi za preporučivanje sadržaja već godinama vrše izbor i filtriranje informacija na internetu i društvenim mrežama koje stižu do korisnika. Imajući u vidu da društvene mreže predstavljaju jedan od bitnih kanala informisanja i upoznavanja mišljenja drugih ljudi, ovakve primene bude podozrivost vezanu za mogućnosti cenzure od strane kompanija koje poseduju te društvene mreže i država koje na njih imaju uticaj. Sistemi za preporučivanje političkih ili društveno relevantnih sadržaja, čak i bez ikakve cenzure ili planskog usmeravanja od strane čoveka, mogu voditi korisnike do jednostranih ili polarizovanih stavova.

Brojni eksperimenti pokazali su da sistemi veštačke inteligencije mogu vršiti diskriminaciju pojedinaca na osnovu njihovog pola, boje kože, etničke pripadnosti i drugih faktora. Naime, sistemi zasnovani na učenju iz podataka, između ostalog uče i mnoštvo ljudskih predrasuda i diskriminativnog ponašanja koje se oslikava u tim podacima. Poznat je primer sistema COMPAS čija je uloga da sudijama u Sjedinjenim Američkim Državama pruža procene verovatoće da će optuženi ponoviti krivično delo ukoliko se bude branio sa slobode. Ustanovljeno je da ovaj sistem precenjuje te verovatnoće u slučaju pripadnika crnačke zajednice.

Tehnike veštačke inteligencije mogu se koristiti i na načine koji mogu ugrožavati privatnost: na primer, za prepoznavanje i označavanje fotografija sa interneta.

11.3 Talasi veštačke inteligencije

Mnoge metode veštačke inteligencije zasnovane su na simboličkoj reprezentaciji: i problem i algoritam rešavanja opisani su eksplicitno, a osobine algoritma mogu se analizirati rigorozno, matematički. I opis problema i algoritam su vrlo specifični, prilagođeni jednom konkretnom zadatku i obično se teško uopštavaju. Ti eksplicitni opisi obično se daju u terminima teorije grafova ili matematičke logike. Za ove metode često se kaže da su GOFAI (od engleskog „Good Old-Fashioned Artificial Intelligence“ – „dobra stara veštačka inteligencija“) i čine takozvani „prvi talas“ veštačke inteligencije. One se najčešće oslanjaju na deduktivno zaključivanje.

Od 2010-ih godina, novi moćni računari omogućili su povratak neuronskih mreža (koje su razvijane još 1950-ih), kroz duboko učenje i ostvarili fantastične rezultate u mnogim poljima, kao što je računarski vid, automatsko prevođenje, automatsko upravljanje vozilima, igranje strateških igara, itd. Ovi rezultati zasnovani su na statistici i mašinskom učenju, tj. na induktivnom zaključivanju. U njima nema eksplicitnog opisivanja procesa rešavanja konkretnih primeraka problema, već se koriste meta-algoritmi (tzv. algoritmi učenja) kojima se, koristeći raspoložive podatke, kreiraju algoritmi za rešavanje konkretnih problema. Ovakvi sistemi obično nisu u stanju da ponude i neko objašnjenje za rešenja koje nude. Ipak, takozvani veliki jezički modeli (poput sistema ChatGPT) imaju i tu mogućnost i koriste je sa većom ili manjom uspešnošću. U razvoju i u primenama ovakvih sistema, uloga čoveka je glavna u fazi modelovanja problema i pripremi podataka za obučavanje. O ovim algoritmima i njihovim osobinama znatno je teže formalno rasuđivati nego o algoritmima koji se zasnivaju na eksplicitnom opisu problema, ali je moguće izvesti statističke ocene njihovog kvaliteta. Sistemi zasnovani na statistici i mašinskom učenju čine takozvani „drugi talas“ veštačke inteligencije.

Rasprostranjeno je uverenje da će u budućnosti dosadašnja dva pristupa morati da se integrišu, vodeći do takozvanog „trećeg talasa“ veštačke inteligencije i to se već dešava. Očekuje se da će u trećem talasu sistemi veštačke inteligencije samostalno kreirati modele koji će moći da objasne kako stvari funkcionišu.

11.4 Znanje i zaključivanje

Za pojam inteligencije suštinske su dve komponente: *znanje* i *zaključivanje*. Raspoloživo znanje nekada je toliko obimno da je potrebno koristiti specijalne tehnike *pretrage* kako bi se došlo do željene informacije (na primer, da li u skupu milion ljudi postoji niz poznanika koji povezuju zadate dve osobe).

Komponenta zaključivanja predstavlja takođe neku vrstu znanja – to je znanje (koje se naziva i meta-znanjem) o procesu izvođenja novog znanja iz raspoloživog znanja. Pod znanjem podrazumevamo i istinite, potvrđene činjenice, ali i hipoteze, nepotpune informacije i informacije date sa određenim verovatnoćama. Zaključivanje može biti deduktivno – zasnovano na rigoroznim opštim pravilima čijom primenom se dobijaju nove konkretne činjenice. Zaključivanje može biti i induktivno – u njemu se na osnovu mnoštva činjenica pokušava izvođenje opštih pravila. Postoje i druge forme i drugi okviri zaključivanja. Jednostavnim primerom ilustrovaćemo nekoliko različitih oblika zaključivanja. Razmotrimo odnos veze

(*i*) $\forall x(P(x) \Rightarrow Q(x))$

i činjenica

(*ii*) $P(a)$

(*iii*) $Q(a)$.

Izvođenje činjenice $Q(a)$ na osnovu (*i*) i (*ii*) odgovara matematičkim principima deduktivnog zaključivanja. Izvođenje veze (*i*) na osnovu niza parova (*ii*) – (*iii*) za različite instance argumenta, odgovara nepotpunoj indukciji, nema matematičku egzaktnost, a u praktičnim primenama pouzdanost mu se povećava sa brojem instanci koje potvrđuju hipotezu. Izvođenje činjenice (*ii*) na osnovu (*i*) i (*ii*) zovemo *abdukcijom*. Ono takođe nije egzaktno, a opravdanje ima u odnosu uzroka i posledice. Na primer, ako za veliki broj osoba znamo da imaju povišenu temperaturu ako imaju grip i ako znamo da neka osoba ima povišenu temperaturu, moguće (ali ne nužno) objašnjenje je da ta osoba ima grip. Abduktivno rasuđivanje zaista se često primenjuje u medicinskim ekspertnim sistemima za utvrđivanje mogućeg uzroka na osnovu poznatih simptoma.

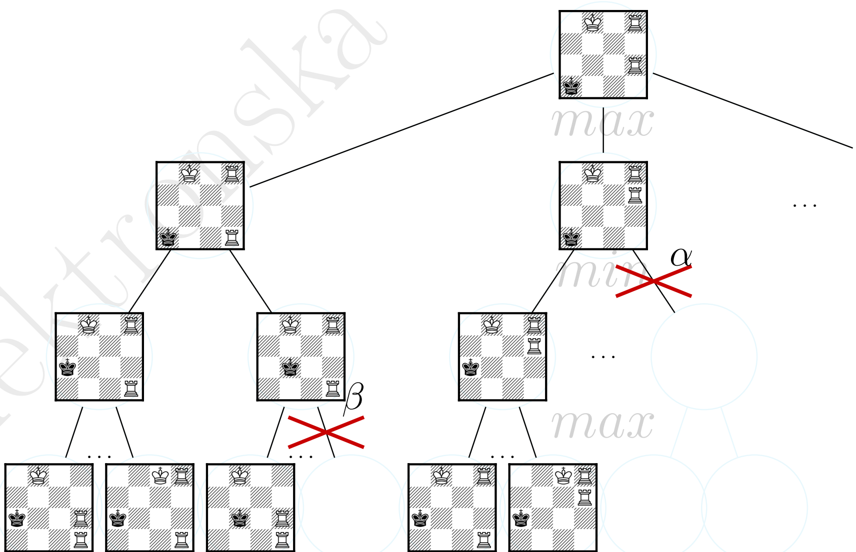
Izbor reprezentacije znanja jedan je od ključnih problema i on je u direktnoj vezi i sa prirodom određenog znanja, ali i sa prirodom mehanizama za zaključivanje. Mehanizmi za zaključivanje moraju biti prilagođeni reprezentaciji znanja i njegovoj prirodi, pa će u jednom slučaju biti zasnovani na klasičnoj logici, a u drugom na modalnoj logici, teoriji verovatnoće, fazi logici itd.

Elektronska verzija (2024)

11.5 Pretraga

U mnogim praktičnim problemima broj postojećih mogućnosti je tako veliki da ne mogu biti sve ispitane sistematično u razumnom vremenu. Na primer, u sistemu za navigaciju, ukoliko je potrebno naći put između neke konkretne adrese u Lisabonu i neke konkretne adrese u Vladivostoku, broj ključnih tačaka koje se mogu razmatrati u traženju najboljeg puta mogao bi da bude prevelik za praktičnu primenu. U takvim situacijama, pretraga se ne vrši na sistematičan način, već se usmerava *heuristikama* — pravilima koja formalizuju smernice za rešavanje nekog problema. Heuristike ne garantuju uvek pronalaženje najboljeg rešenja, ali obično do rešenja dovode mnogo brže nego sistematična pretraga. Usmerena pretraga koristi se u mnogim oblastima računarstva – u pronalaženju najkraćih puteva, u rutiranju, u rešavanju optimizacionih problema, u računarskom igranju strateških igara kao što je šah.

Igranje strateških igara. U igri šah, u sredini partije prosečno ima oko 38 mogućih poteza. Ako ih u nekoj poziciji razmatramo sve - onda je to 38 mogućnosti. Ako razmatramo i sve moguće odgovore protivnika (tj. ako razmatramo dva polupoteza) - onda postoji 38^2 mogućnosti. Ako razmatramo svih mogućih deset polupoteza, onda ima 38^{10} tj. više od $6 \cdot 10^{15}$ mogućnosti, što se ne može obraditi u razumnom vremenu. Ukoliko bi se analizirale sve mogućnosti do samog kraja partije, onda bi taj broj bio još mnogo veći. Sistematična pretraga, dakle, u problemu kao što je ovaj nije primenljiva. Umesto toga, za igru šah najpre se definiše statička funkcija evaluacije koja omogućava nekakvu procenu pozicije iako se nije došlo do kraja partije. Ta funkcija omogućava da se pretraga vrši do neke fiksirane dubine, umesto samo do pozicija u kojima je partija završena. Sledeći ključni korak je onda primena algoritma koji koristi ocene pozicija na nekoj dubini za izbor poteza u tekućem potezu. Jedan od takvih algoritama (minimax sa alfa-beta odsecanjem) često može da odbaci veliki broj mogućnosti bez narušavanja kvaliteta izabranog poteza. Pojednostavljen prikaz ovakvog pristupa igranju igre šah (za pojednostavljenu igru na tabli 4×4 umesto 8×8, i sa svega nekoliko figura) dat je na slici 11.1. Ovakav pristup (opisan ovde pojednostavljeno) korišćen je od strane računara DeepBlue koji je 1997. godine pobedio u šahu Garija Kasparova, tadašnjeg prvaka sveta. Današnji najbolji programi za igranje šaha mogu da za jednu poziciju analiziraju moguće nastavke do dubine od nekoliko desetine polupoteza i ocenjuju po nekoliko miliona pozicija u sekundi. Današnji najbolji programi kombinuju opisani pristup sa savremenim tehnikama mašinskog učenja čime postaju još moćniji.

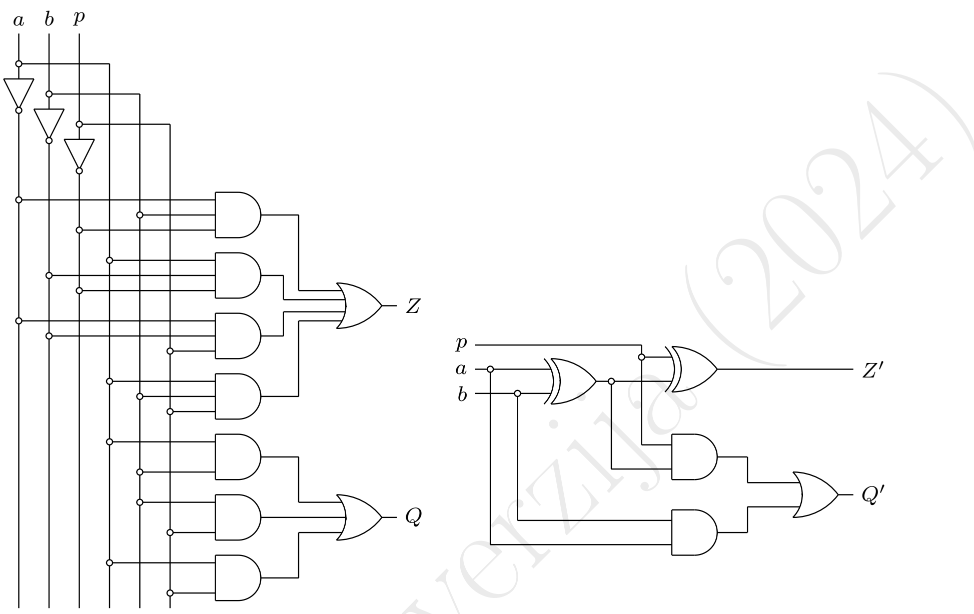


Slika 11.1: Pojednostavljeni prikaz usmeravanja pretrage u igranju igre šah

11.6 Automatsko deduktivno rasuđivanje

U mnogim primenama veštačke inteligencije nije kritično da odgovor koji se dobija od računara bude najbolji mogući i egzaktan. Na primer, nije kritično da li je put između dve tačke najbolji za nekog aktera u nekoj računarskoj igrici. Nije kritično ni da li je neki potez u šahu najbolji mogući, posebno jer je u mnogim situacijama skoro nemoguće utvrditi šta je najbolji mogući potez. U mnogim situacijama prihvatljivo je rešenje koje je „dovoljno dobro“. Takva, dovoljno dobra, a ne nužno tačna, egzaktna i najbolja moguća rešenja obično daju sistemi zasnovani na heuristički usmerenoj pretrazi, kao i sistemi zasnovani na mašinskom učenju i induktivnom

zaključivanju. Međutim, postoje mnoge oblasti primene gde je egzaktnost neophodna. Na primer, sistem metroa mora biti takav da je plan kretanja vozova takav da nikada ne dolazi do sudara. Slično, raspored časova za neku školu mora biti takav da nikad, na primer, jedan nastavnik ne treba da bude u dve učionice istovremeno. U takvim situacijama potrebno je koristiti sisteme koji se zasnivaju na deduktivnom rasuđivanju i koji, u principu, daju rešenja koja su uvek egzaktna. Naravno, to ima svoju cenu, te su takvi sistemi vremenski obično znatno neefikasniji nego sistemi zasnovani na heuristikama i mašinskom učenju. Sistemi zasnovani na automatskom rasuđivaju koriste se u mnogim oblastima računarstva, pa i u mnogim drugim. Na primer, koristeći tu tehnologiju moguće je proveravati ekvivalentnost logičkih kola, napraviti raspored časova za školu ili za fudbalsku ligu, pa čak i vršiti analizu rečenica na prirodnom jeziku.



Slika 11.2: Ilustracija dva ekvivalentna logička kola

Ekvivalentnost logičkih kola. Logička kola su u osnovi svakog savremenog računara. U procesoru računara hardverski su, u vidu logičkih kola, implementirane operacije poput sabiranja, množenja, poređenja i bitovskih operacija. Jednostavnost i efikasnost tih kola je ključna za funkcionisanje procesora, pa i čitavog računara. Nekad neko logičko kolo može biti zamenjeno drugim, jednostavnijim kolom, ali je neophodno najpre se uveriti da su ta dva kola ekvivalentna. Razmotrimo jedan jednostavan primer: neka jedno kolo odgovara logičkoj funkciji $\neg(\neg a \vee \neg b))$ a drugo logičkoj funkciji $a \wedge b$. Da bi se pokazalo da su ova logička kola ekvivalentna potrebno je dokazati da je iskazna formula $\neg(\neg a \vee \neg b)) \Leftrightarrow a \wedge b$ tautologija, tj. da ni za koje vrednosti promenljivih a i b nije netačna. Slika 11.2 ilustruje nešto kompleksniji primer sa dva logička kola za sabiranje: ulazne vrednosti a i b su cifre brojeva koji se sabiraju, p je prethodni prenos, a na izlazu je formula koja opisuje cifru zbira Z i formula koja opisuje novi prenos Q. Da bi se dokazalo da su ova dva kola ekvivalentna potrebno je pokazati da su za sve moguće trojke vrednosti a, b i p izlazi ova dva kola jednaki, što se ponovo može opisati nekom formulom za koju je potrebno utvrditi da je tautologija. U industrijskoj praksi, naravno, javljaju se i daleko kompleksniji zadaci ovog tipa.

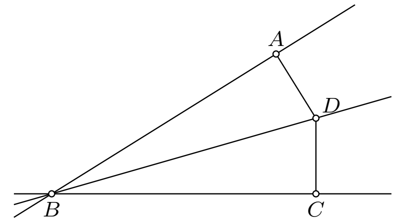
U navedenim primerima, potrebno je imati softversku alatku koja može da proveri da li je data iskazna formula F tautologija. Takvu alatku zovemo *rešavačem.* U prikazanim primerima, postoji svega nekoliko iskaznih promenljivih koje figurišu u formuli, ali savremeni rešavači mogu za svega nekoliko sekundi da ispitaju i neke formule koje imaju na stotine hiljada promenljivih.

Rešavači moraju da daju egzaktan odgovor na pitanje da li je neka formula tautologija ili zadovoljiva ili kontradikcija ili poreciva, ali zbog praktične upotrebljivosti ne mogu sistematično ispituju sve moguće vrednosti za sve promenljive koje se pojavljuju u formuli. Umesto toga, rešavači koriste mnoštvo algoritamskih tehnika koje ubrzavaju ispitivanje. Po tome, rad rešavača sličan je heurističkoj pretrazi, ali uz dodatak da su sve heuristike takve da ne ugrožavaju egzaktnost odgovora rešavača.

Pored alata za rasuđivanje u iskaznoj logici, postoje i alati za automatsko rasuđivanje u drugim, bogatijim logikama. Na primer, takvi alati mogu da rešavaju kompleksne matematičke probleme. Godine 1996. jedan takav alat po prvi put je dokazao teoremu (iz algebre) koju pre toga ljudi nisu uspeli da dokažu.

Elektronska verzija (2024)

Automatsko dokazivanje u geometriji. Jedno od najranijih polja primene automatskog rasuđivanja je geometrija – već sredinom pedesetih godina dvadesetog veka razvijeni su prvi geometrijski dokazivači a jedna od prvih dokazanih netrivijalnih teorema tvrdila je da za proizvoljne međusobno različite tačke A, B, C, D takve da važi: $\angle DB, BA = \angle DB, CB$, $BA \perp AD$, $CB \perp CD$, mora da važi i $AD \cong CD$. Sa raspoloživim aksiomama (koje omogućavaju izvođenje novih zaključaka), savremeni dokazivači bi mogli da proizvedu ovakav dokaz:



1. $\angle BA, AD = \angle CB, CD$
2. $\angle AD, DB = \angle CD, DB$
3. $\angle DB, BA = \angle DB, BC$
4. $\triangle DBA \cong \triangle DBC$
5. $AD \cong CD$

Današnji automatski dokazivači mogu da dokažu daleko složenije teoreme, iz raznih matematičkih disciplina, uključujući mnoge probleme sa međunarodnih matematičkih olimpijada. Automatsko dokazivanje matematičkih teorema nalazi primene ne samo u teorijskoj matematici, već je važno i u mnogim drugim oblastima, na primer u robotici, razvoju softvera, raspoređivanju, itd.

11.7 Mašinsko učenje i induktivno rasuđivanje

Mašinsko učenje, pre svega ono zasnovano na neuronskim mrežama, dalo je nov zalet oblasti veštačke inteligencije u prethodnih 10-15 godina. Pomoću njega napadnuti su neki problemi koji su ranije smatrani zabranom čoveka, a neki dugo otvoreni problemi su u potpunosti rešeni. Najčešće oblasti primene su u računarskom vidu, obradi prirodnih i programskih jezika, autonomnoj vožnji, igranju igara, bioinformatici i šire. Mašinsko učenje omogućava razumevanje saobraćaja od strane autonomnih automobila na osnovu kamera i drugih senzora. Primera radi sa slike je moguće odrediti pozicije pešaka, automobila, semafora i saobraćajnih znakova. Nobelova nagrada za hemiju 2024. godine dodeljena je za rešavanje dugo otvorenog problema savijanja proteina — problem određivanja strukture u trodimenzionalnom prostoru kada je poznat niz aminokiselina koje čine taj protein. Veliki jezički modeli poput ChatGPT-a u stanju su da proizvode visoko kvalitetne prevode između različitih jezika (onih za koje su dostupne velike količine tekstova), popravljaju kvalitet napisanog teksta i da odgovaraju na pitanja i pružaju korisne informacije. Alati poput Kopilota, zasnovani na velikim jezičkim modelima u stanju su da pišu delove programskog koda u skaldu sa zahtevom programera i da tako ubrzaju proces programiranja. OpetnAI o1, veliki jezički model opšte namene rangira se u prvih 12% na problemima takmičarskog programiranja platforme Codeforces. Rešavajući probleme sa Međunarodne matematičke olimpijade, jedan takav sistem specijalizovan za matematičke probleme osvojio je 28 poena — ekvivalent jake srebrne medalje, dok je raspon zlatne medalje počinjao od 29 poena (koje je ostvarilo 58 od 609 takmičara). U igranju strateških igara kao što su šah, go i drugi, neuronske mreže su odavno daleko nadmašile najbolje ljudske igrače.

11.7.1 Mašinsko učenje i generalizacija

Mašinsko učenje može se definisati na više načina. Mašinsko učenje je oblast veštačke inteligencije koja se bavi razumevanjem i formalizacijom induktivnog rasuđivanja, tj. generalizacije. Generalizacija znači formulisanje opštih zakonitosti na osnovu konačnog skupa opažanja. Primera radi, videviši veliki broj labudova od kojih su svi beli, indukcijom se može izvesti zaključak da su svi labudovi beli. Ovakvi zaključci očito mogu biti pogrešni. Teorija mašinskog učenja bavi se razumevanjem uslova pod kojima ovako doneseni zaključci mogu biti visoko pouzdani i metodama koje su u stanju da izvode takve zaključke. Iz perspektive primena, mašinsko učenje se može definisati kao oblast koja se bavi automatskim kreiranjem programa na osnovu datih ulaza i željenih izlaza. Primetimo da i ovo predstavlja vid generalizacije. Naime, ti automatski kreirani programi predstavljaju vid opisa zakonitosti koje važe izmedu njihovih ulaza i izlaza. U nastavku će mašinsko učenje biti posmatrano iz ove praktične perspektive.

Programiranje nije lako. Da bi se uspešno napisao program, potrebno je: precizno opisati problem, detaljno ga razumeti, razložiti na potprobleme koji se lakše rešavaju pojedinačno, smisliti algoritme kojima se svaki od njih rešava i od njihovih rešenja proizvesti rešenje polaznog problema. Nijedan od tih koraka ne mora biti trivijalan, a neki problemi, iako rešivi, za čoveka predstavljaju nepremostiv izazov. Primera radi, neka je potrebno napisati program koji kao ulaz dobija matricu dimenzija $N \times N$ koja sadrži brojeve od 0 do 255 i koja predstavlja sliku u nijansama sive, a na izlazu ispisuje `DA` ili `NE` u zavisnosti od toga da li se na slici nalazi ljudsko lice. Da bi se ovaj problem rešio, potrebno je prvo detaljno razumeti šta su karakteristike ljudskog lica.

Elektronsko izdanje (2024)

Lako je doći do nekih parcijalnih uvida i zaključiti da se lice sastoji od nosa, usta, očiju, ušiju i slično. To samo otvara dalja pitanja vezana za to šta je svaki od ovih pojmova i kakvi odnosi treba da važe među njima da bi oni činili lice. Međutim, suštinska težina ovog posla je u tome da je sve upotrebljene pojmove potrebno definisati u terminima piksela ulazne slike, a ne u bilo kakvim apstraktnijim pojmovima (kao što su zenice, usne, nozdrve, itd.) u kojima su ljudi navikli da razmišljaju i da se izražvaju. Ovo razmatranje sugeriše da je pokušaj pisanja ovakvog programa osuđen na propast već u fazi razumevanja problema i osmišljavanja rešenja. Iako je teško precizno definisati šta je lice, pa samim tim i pristupiti pisanju prethodno pomenutog programa, nije teško sakupiti mnoštvo slika lica sa interneta. Štaviše, moguće je sakupiti skup primera slika koje sadrže lica i za koje odgovor tog programa treba da bude `DA` i skup primera slika koje ne sadrže lica i za koje odgovor programa treba da bude `NE`. Skup primera svakako ne čini preciznu specifikaciju problema, ali može se prihvatiti kao neka neprecizna implicitna specifikacija. Pitanje koje se postavlja u mašinskom učenju je da li program za prepoznavanje lica možemo automatski izvesti procesom *učenja* iz pomenutog skupa primera. U velikom broju praktičnih problema moguće je naučiti takav program koji će *često* za date ulaze davati *dobre* izlaze. Šta u prethodnoj rečenici znači *često* i *dobro* nije precizno definisano, već se u praksi kvalitet izlaza ovakvih programa empirijski evaluira. Primera radi, procenat tačnih identifikacija lica može biti dobra mera kvaliteta ovakvog programa. U uobičajenom programiranju podrazumeva se zahtev da programi budu korektni, tj. da za date ulaze uvek daju dobre izlaze. U mašinskom učenju podrazumeva se da će biti grešaka, tj. da kreirani program za neke ulaze ne daje ispravne izlaze, ali se teži tome da greške budu retke i male. Kako se obično kaže, poželjno je da modeli *dobro generalizuju*. Naglasimo da za dobru generalizaciju nije dovoljno da naučeni program često daje dobre izlaze na podacima iz kojih je naučen. Od presudnog je značaja da to važi i na podacima iz kojih nije učeno. Pritom, greške su očekivane i na jednoj i na drugoj grupi podataka.

11.7.2 Podaci

Proces učenja polazi od podataka. Podaci su opisani nekim numerčkim svojstvima. Primera radi, banka može odlučiti da svoje klijente opiše svojstvima kao što su ime, prezime, godine starosti, godišnja primanja, bračni status, itd. Istorija dnevnih cena nekih akcija na berzi može se opisati nizom vrednosti od kojih svaka odgovara nekom danu u posmatranom vremenskom intervalu i ima vredost cene tih akcija na dati dan. Slika se može opisati matricom trojki intenziteta crvene, zelene i plave. Na ovaj način, svi podaci od interesa mogu se opisati brojevima.

11.7.3 Modeli i njihovo obučavanje

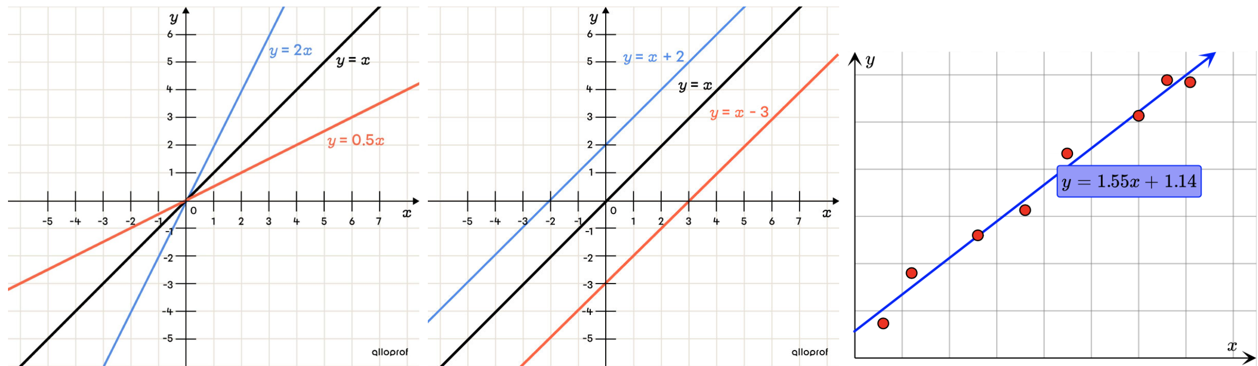
Algoritam koji analizira podatke u potrazi za zakonitostima u njima naziva se *algoritmom učenja*. On kao izlaz daje nekakav matematički opis uočenih zakonitosti. Takvi opisi nazivaju se *modelima*. U razmatranom problemu prepoznavanja lica, program za prepoznavanje lica predstavlja upravo jedan model. Njegov kôd opisuje vezu između ulaza (slika) i izlaza (odgovora `DA` i `NE`). Programi koje pišemo na uobičajeni način obično su čitljivi i pišu se u nekom od standardnih programskih jezika, dok su modeli, tj. programi koji se automatski kreiraju mašinskim učenjem obično matematičke funkcije definisane velikim brojem parametara.

Nameću se mnoga pitanja u vezi sa tim matematičkim funkcijama. Kakve su to funkcije? I kako se pomenuti parametri izračunavaju iz podataka? Jednostavna vrsta parametrizovanih matematičkih funkcija su funkcije oblika $y = ax + b$, gde su y i x promenljive (recimo, telesna masa i visina osobe, tim redom), a a i b nepoznati parametri koje treba odrediti, tako da ova veza dobro opisuje što veći broj ljudi. Ilustracije takvih funkcija prikazane su na slici 11.3. Neka su za određeni broj osoba (recimo 20), izmerene telesna masa i visina. Da li je moguće naći vrednosti parametara a i b tako da model $y = ax+b$ dobro odgovara podacima? Ukoliko je moguće, to znači da je moguće za datu visinu, pomoću ovog modela, predvideti telesnu masu. Ali šta uopšte znači da model dobro odgovara podacima? Prirodno je da im utoliko bolje odgovara, što su razlike između predviđenih i stvarnih vrednosti telesnih masa manje. Recimo, ako su dati parovi visina i masa $\{(x_i, y_i) \mid i = 1, \ldots, 20\}$, želimo da vrednost ukupnog odstupanja

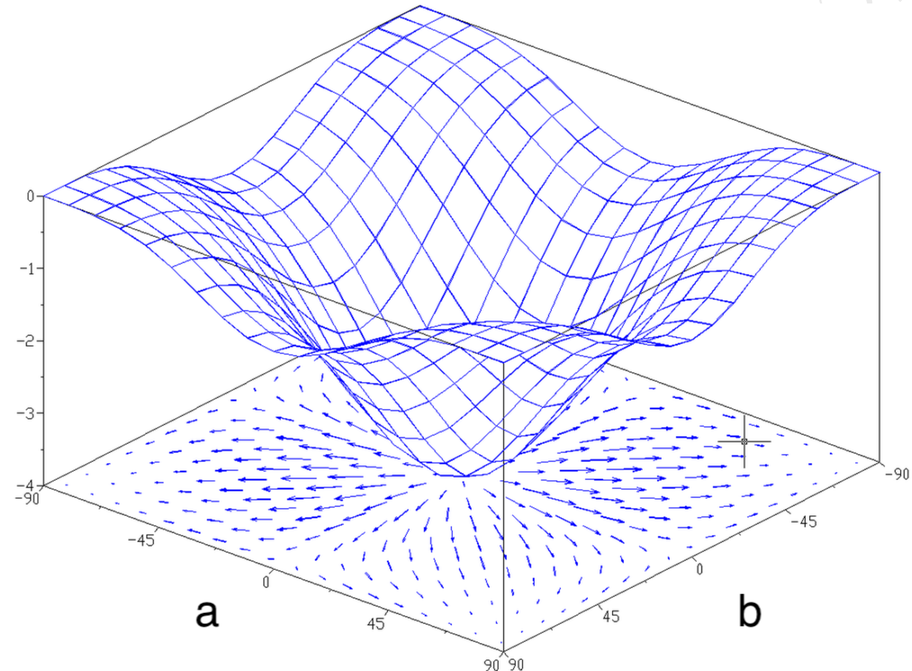
$$L(a, b) = \sum_{i=1}^{20} (y_i - (ax_i + b))^2$$

bude što manja. Ilustracija funkcije koja u ovom smislu dobro odgovara datim podacima data je na slici 11.3.

Očito vrednost $L(a, b)$ zavisi od toga kako izabaremo parametre a i b. Vrednosti u kojima ova funkcija dostiže svoj minimum mogu se dobiti kretanjem nizbrdo pošavši od nekih proizvoljnih vrednosti. Ovaj postupak je moguće izvesti zahvaljujući poznavanju *gradijenta* funkcije u različitim tačkama. Gradijent je vektor koji u svakoj tački pokazuje pravac najbržeg rasta funkcije u okolini te tačke i postoje jednostavne formule za njegovo izračunavanje. Ovaj koncept ilustrovan je slikom 11.4. Ako pođemo od proizvoljne tačke (a, b) i krećemo se malim



Slika 11.3: Prikaz različitih modela oblika $y = ax + b$ pri promeni parametara a (levo) i b (u sredini) i prikaz prave koja „dobro” odgovara nekim datim podacima (desno).

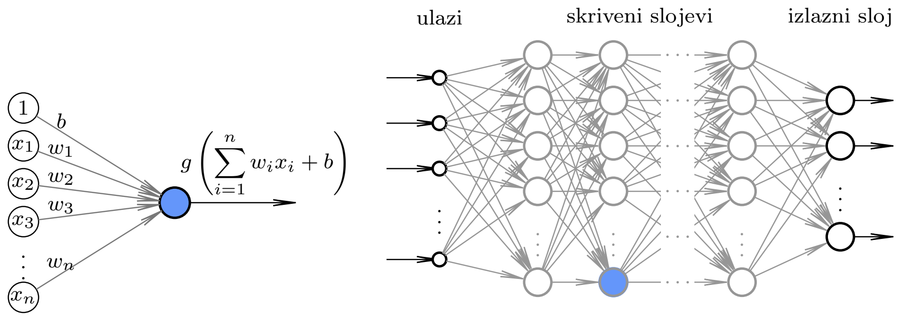


Slika 11.4: Prikaz gradijenta funkcije u različtitim tačkama (a, b).

koracima u smeru suprotnom od gradijenta, nakon dovoljno koraka, naći ćemo se dovoljno blizu tačke u kojoj funkcija dostiže minimalnu vrednost. Funkcija $L(a, b)$ koju posmatramo je jednostavna i ima samo jednu takvu tačku. U slučaju komplikovanijih funkcija, ovaj postupak ne mora uvek naći najbolje vrednosti za parametre funkcije, ali u praksi daje dobre rezultate.

11.7.4 Neuronske mreže

Modeli u vidu pravih kakve smo do sada razmatrali su jednostavni i korisni za rešavanje nekih relativno jednostavnih problema. Postavlja se pitanje postoje li neki komplikovaniji i moćniji modeli. Ideja za njihovu konstrukciju može se pozajmiti iz neurologije. Ljudski mozak sastoji se od relativno jednostavnih gradivnih elemenata – neurona. Neuroni sakupljaju ulazne signale od drugih neurona sa kojima su povezani i u zavisnosti od toga kakve su signale primili, daju nekakav izlazni signal drugim neuronima sa kojima su povezani. Analogno tome, moguće je organizovati izračunavanje u veštačkoj *neuronskoj mreži* prikazanoj na slici 11.5. Svaki od neurona (prikazanih krugovima) predstavlja jednu parametrizovanu funkciju. Neuronska mreža predstavlja složenu kompoziciju neurona, pa je i sâma jedna parametrizovana funkcija (čiji su parametri zapravo parametri svih neurona skupa). Neuroni se mogu definisati na različite načine, ali se obično koristi linearna funkcija ulaza na koju se primenjuje neka jednostavna nelinearna transformacija. Kako se obučava jedna neuronska mreža? Isto kao i jednostavna prava iz prošlog primera! Ukoliko je dostupan skup parova ulaza i njima odgovarajućih izlaza, moguće je izračunati koliko izlazi mreže odstupaju od željenih izlaza za date ulaze. Ponovo je moguće



Slika 11.5: Struktura neurona i arhitektura potpuno povezane neuronske mreže.

smanjivati grešku koju mreža pravi varirajući vrednosti parametara krećući se nizbrdo, u smeru suprotnom od gradijenta.

Ovo objašnjenje neuronskih mreža dato je u vrlo grubim crtama. Postoje mnoge varijacije izloženog principa. Recimo, mreže koje predviđaju neprekidne vrednosti (poput boja na slici) i mreže koje predviđaju diskretne vrednosti (poput reči iz nekog rečnika) imaju nešto drugačije strukture. Načini na koji se mere odstupanja između ispravnih i predviđenih vrednosti takođe se razlikuju u različitim slučajevima. Čak se i cela konstrukcija mreže može značajno razlikovati u slučaju obrade slika, videa, zvuka, teksta ili tabelarnih podataka. Ono što sve te varijacije imaju zajedničko je da sve neuronske mreže predstavljaju kompoziciju velikog broja jednostavnih parametrizovanih funkcija koje se kombinuju u složenu celinu.

11.7.5 Neuronske mreže u praktičnim primenama

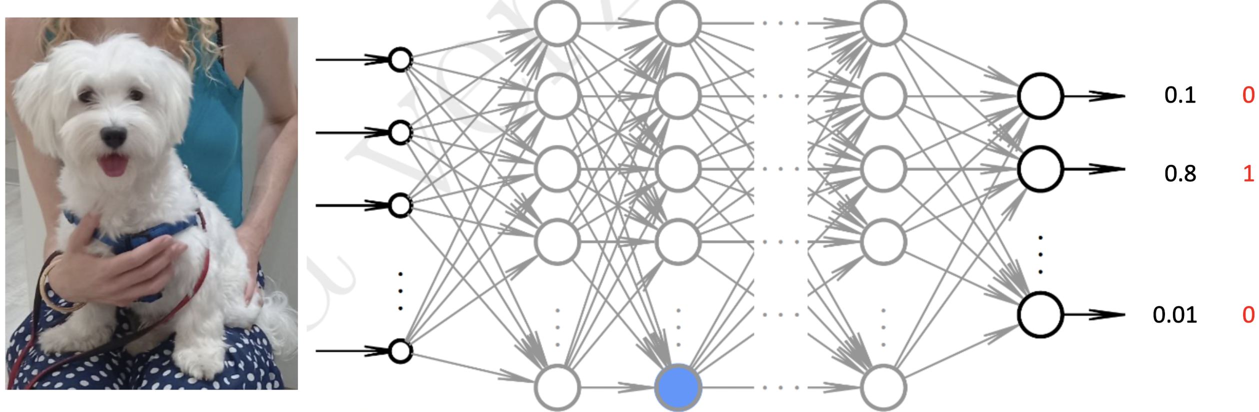
Razmotrimo nekoliko primera primene neuronskih mreža sa fokusom na tome kako se one treniraju u datim kontekstima. Navedene primene tiču se računarskog vida i obrade prirodnog jezika, ali uzbudljivih primena ima i u raznim drugim oblastima.

Razvrstavanje slika. Pretpostavimo da je potrebno prepoznati rasu psa na slici i da je dostupan veliki broj slika za koje je poznato koju rasu prikazuju. Slike se tipično predstavljaju kao matrice piksela od kojih svaki piksel ima pridružena tri broja — intenzitete crvene, zelene i plave boje. Svi ti brojevi, u nekom fiksiranom poretku, predstavljaju ulaze neuronske mreže. Mreža treba da ima onoliko izlaza koliko ima rasa pasa. Ideja je da svaki izlaz odgovara jednoj rasi i da veće vrednosti na izlazu sugerišu veću verovatnoću da slika prikazuje psa te rase. Ovakav dizajn mreže ilustrovan je slikom 11.6. Prirodno je očekivati od mreže da za ulaznu sliku, na izlazu koji odgovara tačnoj rasi, bude vrednost bliska jedinici, a da na izlazima koji odgovaraju pogrešnim rasama budu vrednosti bliske nuli. Ukupno odstupanje od ovih vrednosti za sve raspoložive podatke treba da budu što manje, što se postiže kao i pre — podešavanjem parametara metodom zasnovanom na kretanju nizbrdo. Broj neurona u mreži i njihov raspored obično se određuju eksperimentalno, tako da se na kraju dobiju što bolji rezultati. To eksperimentisanje obično prati neke ustaljene prakse koje ovde nećemo objašnjavati.

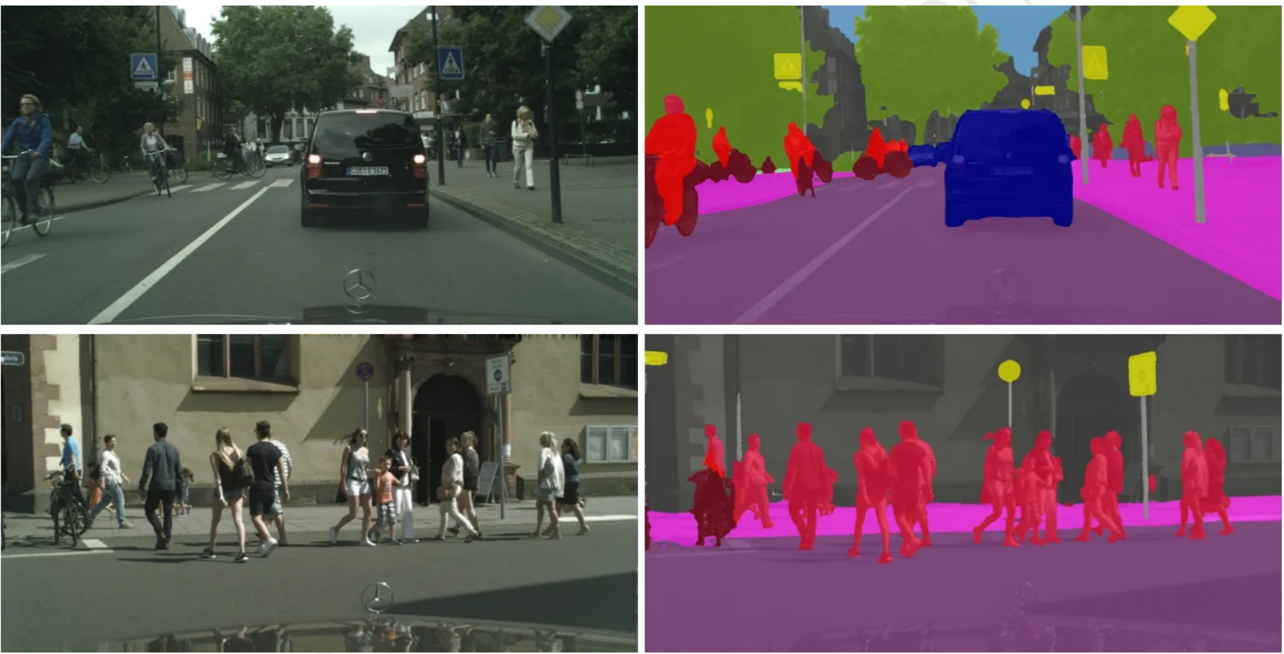
Semantička segmentacija slika. Prethodni primer opisuje problem koji se smatra relativno jednostavnim — razvrstavanje slika u neke unapred definisane kategorije. Druga, izrazito korisna vrsta zadatka koju neuronske mreže mogu da vrše je razvrsavanje pojedinačnih piksela u te unapred definisane kategorije, čime se zapravo vrši segmentacija slike u smislene celine. Primera radi, neki pikseli na slici predstavljaju osobe, neki automobile, neki trotoar i tako dalje. Za neuronsku mrežu koja treba da upravlja autonomnim automobilom od velikog je značaja da razume sliku u takvim terminima i da na osnovu toga razume, recimo, da se pešak nalazi ispred automobila. Prikaz ovakvog zadatka dat je slici 11.7. Neuronska mreža koja bi na ovaj način kategorisala piksele imala bi jednako organizovan ulaz kao i prethodna, ali bi izlaz bio značajno drugačije implementiran. Prethodna mreža bila je u stanju da pomoću N izlaza prepozna jednu od N rasa pasa. Mreža koju sada diskutujemo mora biti u stanju da na sličan način razvrsta svaki od $m \times n$ piksela ulazne slike u neku od N kategorija, tako da mora imati $m \times n \times N$ izlaza. Ukoliko je (najčešće zahvaljujući ljudskom trudu) dostupan skup podataka u kome je za veliki broj slika za svaki piksel poznato kojoj kategoriji pripada, moguće je naučiti neuronsku mrežu da vrši ovaj vid analize slike.

Generisanje slika. U problemu generisanja slika, neuronske mreže dostigle su izvanredne rezultate. Obično se mreži na ulazu pruži tekstualni opis slike koju je potrebno da generiše, kao i neki vektor pseudo-slučajnih brojeva.

Elektronska verzija (2024)

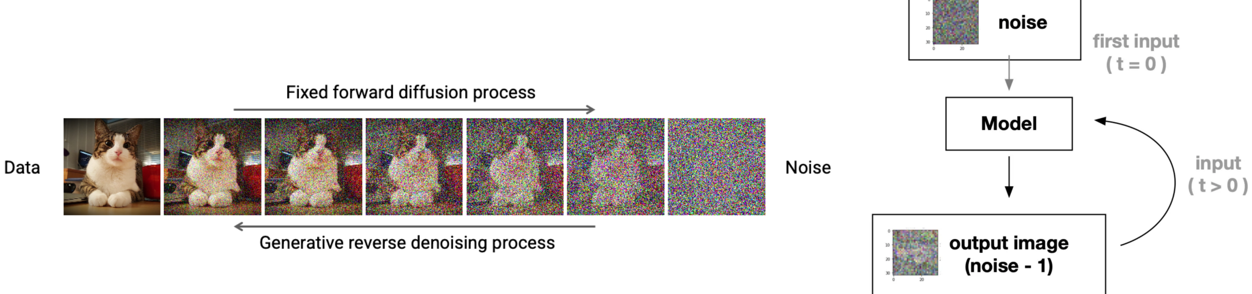


Slika 11.6: Prikaz mreže koja prepoznaje rasu psa. Slika se daje na ulazima mreže, a izlazi treba da odgovaraju stvarnoj rasi psa. Željeni izlazi su prikazani crvenom bojom.



Slika 11.7: Prikaz segmentacije slika iz saobraćaja na kojima se vide jasno označeni učesnici u saobraćaju i saobraćajnij znakovi.

Na osnovu ta dva ulaza, mreža generiše izlaznu sliku. Za isti zadati opis, ali za različite izbore pseudo-slučajnih brojeva, mreža generiše različite slike. Kako ne bismo ulazili u detalje razumevanja teksta, pojednostavićemo problem tako što ćemo pretpostaviti da mreža ne uzima nikakav opis (recimo da je naučena samo da generiše slika pasa i da nije potrebno tražiti od nje više do toga), već da samo na osnovu pseudo-slučajnih brojeva na ulazu generiše sliku. Dobiti kvalitetnu sliku sigurno nije lako. Ali ono što jeste lako, to je pokvariti već postojeću dobru sliku. Recimo, na sledeći način. Pošavši od jedne slike, moguće je na nju kroz iteracije dodavati male količine šuma — pseudoslučajnih brojeva bliskih nuli, sve dok količina šuma u potpunosti ne uništi sadržaj slike i ne završimo sa slikom koja predstavlja samo neke pseudo-slučajne brojeve. Ovaj proces ilustrovan je na slici 11.8. Ovo je lako, ali nije mnogo korisno. Ipak, postavlja se pitanje, da li se ovaj proces može nekako obrnuti: možemo li, krećući se od neke slike popunjene pseudoslučajnim brojevima, doći do kvalitetne slike? Odgovor je pozitivan, i proces se može ugrubo razumeti na sledeći način. Niz slika $I_1, I_2, \ldots, I_n$ koji je generisan dodavanjem šuma može se iskoristiti za dobijanje parova ulaza i izlaza. To bi bili svi mogući parovi (I_i, I_{i-1}) gde prva slika predstavlja ulaz (koji ima više šuma), a druga izlaz (koji ima manje šuma). Za sve te parove, mreža uči da za datu sliku I_i generiše odgovarajuću sliku I_{i-1}, tj. da otklanja šum sa slike. Kada je mreža obučena, proces generisanja slike je iterativan i polazi od slike koja sadrži samo pseudo-slučajne brojeve. U svakoj iteraciji, naučena mreža se primenjuje na tekuću sliku dok se ne dostigne određeni broj iteracija. Taj broj je iskustveno određen tako da proces na kraju rezultuje verodostojnom slikom bez šuma.



Slika 11.8: Niz slika dobijenih dodavanjem šuma na pravu sliku (levo) i shema modela koji uzima sliku sa više šuma i proizvodi sliku sa manje šuma (desno).

Veliki jezički modeli. Trenutno najuzbudljiviju primenu neuronskih mreža predstavljaju veliki jezički modeli. Pojednostavljeno, jezički model je model koji za dati tekst (koji se često naziva kontekst) i svaku potencijalnu narednu reč, procenjuje njenu verovatnoću. Na taj način, jednim jezičkim modelom moguće je generisati verodostojne tekstove na nekom jeziku. Naravno, kao i u slučaju slika, i reči se predstavljaju brojevima. Veliki jezički modeli predstavljaju neuronske mreže obučene na ogromnim korpusima teksta sakupljenim sa čitavog interneta i iz drugih dostupnih izvora. Primetimo da u takvom učenju nije neophodan veliki ljudski trud u pripremi podataka — kada su tekstovi sakupljeni, svaki niz uzastopnih reči iz tog teksta predstavlja jedan kontekst za koji je poznata naredna reč (ako postoji). Iz ogromnog broja situacija, neuronska mreža naučiće ne samo da pogađa najverovatniju sledeću reč, već i verovatnoće mnogih drugih reči kao potencijalnih sledbenika. Naime, ukoliko se u tekstovima iz kojih mreža uči nakon konteksta „Marko igra" pet puta javlja reč fudbal, dva puta reč košarka i tri puta reč tenis i slično za kontekste „Jovana igra", „Petar igra" i druge, mreža će, u nekom obliku, naučiti da razne osobe u približno 50% slučajeva igraju fudbal, u 20% košarku, a u 30% tenis, pa će tako proceniti verovatnoće i u slučaju konteksta „Janko igra" čak i ako Janko nije pomenut u tekstovima iz kojih je mreža učila. Kada je mreža naučena da ocenjuje verovatnoću naredne reči za dati kontekst, obično se dalje doobučava na posebno pripremljenim tekstovima koji predstavljaju parove pitanja i odgovora pošto je to očekivani vid interakcije sa čovekom. Ovakvi sistemi sposobni su da vode smislene konverzacije sa čovekom i da daju korisne informacije, da prevode tekst sa jednog jezika na drugi, da rešavaju matematičke probleme, da pišu delove programskog koda i slično. Neretko i greše, pa je u njihovoj upotrebi i dalje potreban nadzor čoveka. Ipak, dometi njihovog uspeha iznenadili su i mnoge profesionalce u oblasti veštačke inteligencije. Posebno iznenađenje predstavlja činjenica da je njihov uspeh postignut na principu predviđanja naredne reči, što je mnogima delovalo kao previše grub i jednostavan princip za rešavanje sofisticiranih problema. Pokazalo se da je veliki jezički model učeći da korektno niže reč za rečju, zapravo naučio i sistem internih reprezentacija pročitanog teksta, a nad njima i mehanizme za rešavanje najrazličitijih problema. Ipak, kako bi veliki jezički model uspeo da da korektan izlaz nižući reč za rečju, zapravo je neophodno da nauči sistem internih reprezentacija pročitanog teksta, a nad njima mehanizme za rešavanje najrazličitijih problema. To je u nekom smislu prilično slično onome što čovek radi kad rešava probleme - čita njihovu specifikaciju, razume je u svom konceptualnom sistemu koji je oblikovan recimo školovanjem, primenjuje neki mehanizam rešavanja i na osnovu toga ispisuje rešenje. Naravno, dubina ovakvih analogija je otvoreno pitanje, jer u ovom trenutku ni funkcionisanje ljudskog mozga, a ni unutrašnje funkcionisanje velikih jezičkih modela nije dovoljno razjašnjeno.

Opisani primeri kriju kompleksnost procesa treniranja i evaluacije neuronskih mreža i daju samo obrise ideja na kojima su rešenja pomenutih metoda zasnovna. U praksi, kako bi se na ovakvim modelima zasnovali upotrebljivi proizvodi, potrebna je velika količina teorijskog razumevanja i praktičnog iskustva. Pored toga, potrebne su veštine razvoja softvera, razumevanje rada hardvera, operativnih sistema, izračunavanja u oblaku itd. Stoga se ovakvim problemima obično bave raznovrsni timovi inženjera, a nekad i istraživača, uz veliku pomoć ljudi koji pripremaju podatke za obučavanje. Pored raznovrsnog znanja, potreban je i specijalizovani hardver na kojem je moguće vršiti učenje velikih mreža na velikim količinama podataka (koji se nekad broje hiljadama, a nekad milijardama). Otuda je rešavanje praktičnih problema pomoću neuronskih mreža puno izazova, ali shodno tome i uzbuđenja.

Elektronska verzija (2024)

Glava 12

Računarska grafika

Elektronska verzija (2024)

Elektronska verzija (2024)

Glava 13

Računari i društvo

U današnjem društvu računari su postali deo svakodnevnice više nego ikada. Uključeni su u sve sfere života, pa stoga njihov uticaj raste iz dana u dan. Kako na veliki deo sadržaja koji do nas stižu isključivo preko računara ne možemo da utičemo, potrebno je što ranije postati svestan svih mogućnosti i ograničenja korišćenja računara, i skrenuti pažnju kako na negativne, tako i na pozitivne aspekte.

U ovom poglavlju će ukratko biti pređeni najvažniji pojmovi koji se javljaju pri korišćenju računara. Kako su mnogi od njih nastali tek sa pojavom računara, odnosno može se reći kao posledica pojave računara, društvo kao celina na neki način je konstantno u trci da na adekvatan način pristupi razvoju računara i izazovima koje taj razvoj nosi.

13.1 Socijalni i etički aspekti računarstva

Iz perspektive studenata matematike i informatike, razvoj računarstva prvenstveno ima ogroman uticaj na razvoj računarskih nauka. Oblasti se menjaju i nove oblasti razvijaju, što dovodi do toga da se kursevi koji se predaju na fakultetu češće menjaju nego što je to bio slučaj u 20. veku. Međutim, kada govorimo o socijalnim i etičkim aspektima računarstva, njihov uticaj je mnogo širi i zahvata čitavo društvo.

13.1.1 Uticaj računarstva na društvo

Tokom prethodnih decenija računari su izvršili dramatičan uticaj na ljudsko društvo. U vreme prvih računara, njihove primene bile su ograničene na svega nekoliko tradicionalnih oblasti: vojne, naučne, bankarske primene i slično. Sa pojavom personalnih računara i, pogotovo kasnije, sa pojavom interneta, računari ulaze u mnoge domove i danas su skoro sveprisutni (godine 2019, u Srbiji je desktop ili laptop računar imalo više od 73% domaćinstava, a raspoloživ internet više od 80% domaćinstava; u Severnoj Americi i u Evropskoj uniji desktop ili laptop računar imalo je oko 85% domaćinstava, a više od 90% je imalo raspoloživ internet; u svetu je desktop ili laptop računar imalo oko 50% domaćinstava, a više od 60% je imalo raspoloživ internet; ukupno, više od 60% osoba na svetu koristi internet svakodnevno ili često).

Danas se računari koriste u skoro svim oblastima ljudskog života: u obrazovanju, u telekomunikacijama, u industriji zabave, za prognozu vremena, itd. Najnovije primene računara omogućavaju, na primer, automatsko dijagnostikovanje bolesti, analizu ljudskog genoma, automatsko navođenje automobila, *internet stvari* (sistem automatskog, efikasnijeg i jeftinijeg upravljanja elektronskim uređajima putem senzora, adekvatnog softvera i računarske mreže, na primer, za kontrolu temperature vazduha u nekom magacinu ili kući), a računari pobeđuju ljude u kvizovima opšteg znanja.

U nastavku su nabrojani neki od faktora zbog kojih računari sve više utiču na društvo u celini:

Sveprisutnost: Računari su sveprisutni, čak i onda kada se njihov rad ne primećuje.

Laka dostupnost informacija: Količina podataka koji su dostupni na internetu raste ogromnom brzinom, baš kao i broj ljudi kojima su ti podaci dostupni. Sa bilo kog mesta, u skoro bilo kojoj situaciji, dostupne su ogromne količine teksta, zvučnih i video zapisa. Može se reći da su internet pretraživači u velikoj meri potisnuli enciklopedije, ali i mnoge druge knjige čiji se sadržaj može naći na internetu.

Efekat umnožavanja: Porast raspoloživih informacija, omogućava hiperprodukciju. Često se prisutne informacije dodatno obrađuju, kombinuju i obogaćuju u cilju još veće proizvodnje i umreženosti podataka. Zbog rasprostranjenosti interneta ovim je omogućeno da i korisne i nekorisne informacije, pa čak i određeni softverski problemi stižu ogromnom brzinom do miliona ljudi.

Elektronska verzija (2024)

Momentalnost: Zahvaljujući računarima i internetu, mnoge ljudske aktivnosti odvijaju se lakše i brže. To postavlja i sve viša očekivanja (na primer, za brzinu objavljivanja rezultata popisa ili rezultata izbora). Sve više elektronskih usluga raspoloživo je neprekidno, dvadeset četiri sata dnevno.

Prostor: Zahvaljujući računarima i internetu, fizička udaljenost više nije ograničenje za mnoge vrste poslova. Kako je moguć pristup velikim količinama podataka putem interneta, mnogi poslovi se mogu raditi na daljinu. Ova pojava je bila posebno vidljiva u doba Korone, kada su mnogi radnici počeli da rade od kuće, a u određenom procentu je zadržana i danas.

Neuništivost: Zahvaljujući sveprisutnoj umreženosti, podaci se sve češće čuvaju na udaljenim računarima ili na mnogo njih. Sa jedne strane to obezbeđuje visok nivo pouzdanosti sistema, ali dovodi i do toga da se gomile beskorisnih podataka skladište i čuvaju zauvek.

13.1.2 Pouzdanost računarskih sistema i rizici po društvo

Dok rade ispravno, računarski sistemi često su neprimetni. Ali, ako dođe do greške u radu sistema, posledice mogu da budu katastrofalne. Računarski sistemi mogu da rade neispravno zbog neispravne specifikacije sistema, greške u dizajnu hardvera, hardverskog otkazivanja, greške u dizajnu softvera, baga u softveru, zbog neispravnog održavanja itd. U kompleksnim sistemima, greška može biti i neka kombinacija navedenih mogućih uzroka. Često su se dešavale greške u radu računarskih sistema koje su ugrozile ili mogle da ugroze živote velikog broja ljudi. Takva je, na primer, bila greška 1980. godine zbog koje je računarski sistem u SAD ukazivao na započeti nuklearni napad Sovjetskog Saveza što je moglo je da dovede do stvarnog nuklearnog rata i uništenja ljudske civilizacije.

Mnoge od primena računara neposredno su unapredile kvalitet ljudskog života i ne postavlja se pitanje da li su računari korisni. Međutim, sve češće se primećuju oblasti u kojima uticaj računara može biti štetan ili, makar, upitan.

- Igranje računarskih igrica, iako može razviti logičko mišljenje i brzinu reakcije u nekim situacijama, takođe može dovedesti do bolesti zavisnosti koje se teško prepoznaju i još teže leče.
- Razvoj elektronskih društvenih mreža doprinosi većoj povezanosti ljudi sa različitih lokacija, ali loše utiče na tradicionalne forme komunikacije. Može dovesti do smanjenog sampouzdanja osobe, kao i do povlačenja u sebe.
- Upotreba elektronske pošte elimiše mnoge atribute komunikacije uživo, ali to može biti ocenjeno i negativno i pozitivno. Osim uštede vremena, elektronska komunikacija može da maskira atribute kao što je rasa, nacionalnost, pol, starost i slično (a koji u nekim situacijama mogu da dovedu do diskriminisanja neke osobe).
- Razvoj veštačke inteligencije budi kod ljudi nove strahove, ne samo od smanjenja radnih mesta za ljude, već i od stvaranja mašina koje čovek neće moći da kontroliše ili od inteligentnih mašina za ubijanje koje će se koristiti u ratovima kao najmodernije oružje. Ti strahovi nisu karakteristični samo za one koji ne poznaju računarstvo, već i za mnoge stručnjake u oblasti veštačke inteligencije (grupa naučnika objavila je 2015. pismo u kojem se poziva na pojačan oprez u razvoju sistema veštačke inteligencije).

Kada se govori o pouzdanosti i greškama u okviru računarskih sistema, bitno je pomenuti i situaciju kada se greške sistema otkrivaju namerno od strane osobe koja traži propuste u sistemu. Iako je inicijalno hakerska kultura (engl. hacking) predstavljala unapređenje sistema ili uređaja i podrazumevala je osobu koja je zasluživala veliko poštovanje, sa vremenom su se razvile ideje zlonamernih napada sa ciljem iskorišćavanja propusta u različitim sistemima i računarskim mrežama (engl. black hat). Kompanije su prepoznale da je rizik od ovih napada veliki i počele su da zapošljavaju etičke hakere (engl. white hat) koji bi uz njihovu dozvolu i za njih testirali sigurnost sistema. Pored ove dve drastično različite grupe hakera postoji i siva zona hakera (engl. gray hat) koji napadaju sisteme i otkrivaju greške ali isključivo sa ciljem dokazivanja svoje umešnosti ili prikazivanja nesigurnosti određenih sistema za celo društvo.

Nažalost, sigurnost sistema nekada zavisi od korisnika sistema mnogo više nego što smo toga svesni. Na primer, šifre koje administratori sistema koriste za pristup sistemu često mogu biti veoma laka meta hakera. Pokazuje se da čak 60% ljudi koristi istu šifru za različite naloge (što znači da bi proboj nekog drugog slabijeg sistema mogao da dovede do otkrivanja šifre koja se koristi na jačem sistemu). U poslednje vreme se sve češće koriste pouzdani softveri koji mogu služiti kako za generisanje šifri, tako i za njihovo čuvanje[1]. Neka opšta preporuka koja danas važi je sa jedne strane korišćenje dugačke nasumične lozinke, a sa druge strane korišćenje

[1]Sistemi poput KeePass, LastPass, Proton Pass, 1Password su poznati kao sigurni sistemi za rad sa šiframa.

Elektronska verzija (2024)

barem dvofaktorske autentifikacije (najčešće kroz dobijanje jednokratnog koda, davanja odgovora na sigurnosno pitanje i slično). Dodatno bitno je napomenuti da se korišćenje besplatnih Wi-Fi mreža (koje su sveprisutne u kafićima, restoranima i hotelima) ne preporučuje kada se preko njih korisnik loguje sa nekim za njega bitnim nalogom jer se na taj način kontrola u potpunosti potencijalno prepušta trećim licima (ovaj način predstavlja lak ulaz i za hakere).

13.1.3 Građanska prava, slobode i privatnost

Privatnost pojedinca je u tesnoj vezi sa građanskim pravima i obavezama i postoji veliki broj zakona koji ih reguliše. Neformalno rečeno, privatnost predstavlja društveni ugovor koji dozvoljava pojedincu da ima značajan nivo kontrole nad vidljivošću njegovih podataka i nad pristupu njegovom umu i telu. Međutim, konstantno korišćenje računara i pametnih mobilnih telefona omogućava, više nego ikad, sa jedne strane uvid i države i raznih drugih organizacija u život pojedinaca, a sa druge strane omogućava preveliki uticaj okoline na individualnu osobu.

Potreban je poseban oprez prilikom ostavljanja elektronskog traga aktivnosti pojedinca. Iako čuvanje velikih informacija online često pojednostavljuje svakodnevicu, na primer, kada treba odrediti put od kuće do neke lokacije, istovremeno ostavlja prostor za mnoge zloupotrebe. Kako mnogi ljudi objavljuju svoju tačnu lokaciju istog trenutka kada odu od kuće, na primer na odmor, to ostavlja mogućnost zlonamernim osobama da opljačkaju njihove domove ili na druge načine zloupotrebe tu informaciju.

Situacija je još komplikovanija zbog bezbednosnih izazova sa kojima su suočene mnoge zemlje zbog potencijalnih terorističkih napada. Naime, tim izazovima opravdava se elektronski nadzor potencijalnih terorista, ali u okviru toga se pod nadzorom nađu i mnogi drugi. Nedavno je otkriveno da su, u okviru svoje borbe protiv terorizma, Sjedinjene Američke Države elektronski nadzirale ogroman broj svojih građana ali i građana drugih država, uključujući i političke lidere najbližih saveznika. Dok jedni smatraju da je to činjeno sa pravom i iz najboljih motiva, drugi smatraju da je to činjeno nelegalno i neetički. Osobe koje su objavile dokumenta koja govore o tom nadzoru, za jedne su heroji, a za druge – izdajnici. U mnogim državama, tokom prethodnih godina usvojeni su zakoni koji štite privatnost pojedinaca i dozvoljavaju elektronski nadzor samo u specijalnim situacijama.

Nisu samo države zainteresovane za prikupljanje privatnih podataka pojedinaca. Određene kompanije ili pojedinci takođe mogu da upotrebe ili bolje rečeno zloupotrebe informacije o privatnim licima. Prilikom otvaranja naloga na raznim sajtovima prisutan je veliki procenat netransparentnosti prilikom dobijanja saglasnoti. Korisnici ostavljaju veliku količinu ličnih podataka ne razmišljajući o potencijalnim rizicima koje takva akcija donosi. Dodatno, mnoge podatke, uz prećutnu saglasnost korisnika, prikupljaju pretraživači veba, društvene mreže i razne aplikacije. Deo tih informacija se dobija kroz postavljanje konkretnih pitanja, međutim deo informacija se dobija i na osnovu posmatranja aktivnosti korisnika. Danas su algoritmi za analizu ponašanja korisnika toliko razvijeni da se lako može primetiti da se na osnovu postavljenih upita i poslatih poruka korisnika, automatski kreira precizan profil korisnika ka kojem se onda usmeravaju personalizovane reklame i popusti. Cela organizacija online informacija je sve više takva da pojedinac stiče utisak da se bez njih ne može, sve više i više aspekata ličnog pa i poslovnog života se oslanja na informacije koje se automatski čuvaju i ažuriraju za nas na internetu. Otuda velika većina ljudi nema u stvari osećaj da je njihova privatnost i na koji način ugrožena i bez mnogo zadrške lako dele lične informacije.

Sve ovo se može prepoznati i u citatu Erika Šmita (engl. Eric Schmidt) koji je bio CEO u kompaniji Google: *Sa vašim odobrenjem dajete nam još više informacija. Ako nam dajete informacije o tome ko su neki od vaših prijatelja, mi verovatno možemo koristiti neke od tih podataka, opet, sa vašom dozvolom, da poboljšamo kvalitet naših pretraga. Vi uopšte ne treba da kucate, jer znamo gde ste, uz vaše odobrenje. Znamo gde ste bili, uz vašu dozvolu. Više ili manje možemo pretpostaviti o čemu razmišljate.*

Bitno je naglasiti da privatnost pojedinaca može da bude ugrožena i nehatom, greškom ili usled napada hakera. Kako se velika količina ličnih podataka korisnika nalazi na serverima različitih aplikacija, opasnost koje moramo biti svesni je mogućnost objavljivanja tih podataka. U prošlosti su se više puta dešavale situacije kada je dolazilo do curenja spiskova koji sadrže informacije o milionima korisnika određene usluge, koji su nakon toga bili objavljeni ili prodati. Zbog takvih situacija, potrebno je preduzimati rigorozne mere koje minimizuju mogućnost gubljenja ili curenja podataka iz sistema.

Dodatno, u prošlosti su se dešavale situacije kada su firme prodavale privatne podatke svojih korisnika bez traženja njihove saglasnosti[2]. Otuda, prilikom korišćenja računara posebna pažnja treba biti usmerena ka društvenim mrežama. Koncept društvenih mreža u suštini je takav da skoro u potpunosti narušava privatnost

[2]Kompanija Facebook je prodala podatke o minimum 50 miliona korisnika, neki od korisnika su dali saglasnost za prikupljanje podataka, međutim podaci su prikupljani i od njihovih facebook prijatelja koji takvu saglasnost nisu dali

Elektronska verzija (2024)

svojih korisnika. Iako je koncept privatnih naloga[3] postao sve aktuelniji poslednjih godina, većina korisnika i dalje nije u potpunosti svesna važnosti ograničavanja pristupa svom nalogu. Ovo pogotovo važi za mlade ljude, koji su najčešće neiskusni i željni kontakata i prihvatanja od svoje okoline. Tako da se čak i primećuje povećanje u broju informacija koje tinejdžeri dele na društvenim mrežama.

U kontekstu privatnosti, specifično je pitanje privatnosti zaposlenog i njegovih komunikacija u okviru kompanije u kojoj radi. Dešava se da u nekim firmama nadređeni u potpunosti ima pristup mail-u, a nekad i istoriji pretraživanja, zaposlenog. Dodatno, mail-ovi svih zaposlenih su najčešće na jednoj zajedničkoj listi što omogućava jednostavno (iako nekad ne-zlonamerno) slanje mail-a svim zaposlenima i u situacijama lične prirode ili u situacijama kada jedan zaposleni odluči da obavesti sve ostale o svojim ličnim stavovima po određenom pitanju.

Cenzura. Namera da se ograniči pristup određenim informacijama koje određene grupe smatraju štetnim zove se cenzura. Pretežno je kontrolisana od strane država i vladajućih stranki, međutim postoje i manje grupe koje sprovode cenzuru zarad svojih ličnih ciljeva. Najčešće se radi na tome da informacije ne budu objavljene uopšte, međutim kako je u današnje doba to skoro pa nemoguće, danas cenzura predstavlja smanjivanje vidljivosti tih informacija. Cenzura se kosi sa jednim od osnovnih građanskih prava, pravo na slobodu govora – da se, ukoliko one ne ugrožavaju druge, iznesu sopstvene ideje bez straha od kažnjavanja ili cenzure. Nastanak interneta nudi i u principu omogućava potpunu slobodu izražavanja širokom krugu ljudi, ali istovremeno omogućava i promovisanje radikalnih ideja, mržnje, pa i terorističke borbe. Pa se može steći utisak da je danas cenzura možda još potrebnija nego inače, makar kada su u pitanju izvori koje prate velike količine ljudi. S obzirom da se objavljivanje informacija na društvenim mrežama vrši skoro bez ikakve prethodne provere, internet je postao poligon za širenje svih vrsta poluinformacija, dezinformacija, „lažnih vesti“ (eng. „fake news“) i manipulacija. U ranijim godinama ovakva pojava je bila mnogo manje raširena, ali danas je ovakvo ponašanje u toj meri prisutno da može da utiče i na donošenje odluka, uključujući i značajne političke odluke ili ishod izbora. Pojedine države (u manjoj ili većoj meri) kontrolišu internet saobraćaj svojih građana ili pokušavaju da kontrolišu sadržaj na društvenim mrežama, a sa obrazloženjem da to čine u cilju očuvanja sopstvenih sloboda i nezavisnosti.

Međutim, potrebno je istaći da postoje situacije kada je cenzura preko potrebna zarad zaštite mladih i maloletne dece od sadržaja koji nisu primereni. Svakako ispravna granica za cenzuru se teško određuje i potrebna je pažljiva analiza svih informacija i korisnika koji su tim informacijama izloženi. Mnogi filmovi i članci imaju oznaku minimalnog broja godina koje gledalac/čitalac bi trebao da ima. Osim toga, postoje razni načini na koji roditelji pokušavaju da zaštite svoju decu. Možda najpopularniji način je instaliranje aplikacija koje kontrolišu sajtove koji su dozvoljeni deci ili video sadržaj na različitim platformama ili na televiziji. Međutim i ovde roditelji treba da budu obazrivi jer u današnje doba nije neobično da deca znaju više o mobilnim telefonima i računarima od odraslih i dešava se da ih prevare tako što na primer kreiraju na uređaju dva naloga, jedan sa ograničenim pristupom koji roditelji kontrolišu i drugi bez kontrole koje dete samo koristi bez ikakvog nadzora.

Pomenimo da postoji i samocenzura, koja se prvenstveno odnosi na naše lične odluke da pripazimo šta radimo i govorimo da ne bismo (u suprotnom) povredili ili uvredili druge ljude. Osim što se prvenstveno odnosila na pojedince, samocenzura je danas veoma prisutna i u okviru novina, kada izdavači odlučuju da ne objavljuju potencijalno problematične informacije da bi ostali u dobrim odnosima sa vladajućom strankom.

Iako možda na prvi pogled tako ne deluje, samocenzura je mnogo rasprostranjenija od direktne cenzure. Samocenzura je prisutna svakodnevno, pogotovo kada odlučujemo šta ćemo postaviti na društvene mreže. Međutim, kako je internet glavni "protivnik" direktnoj cenzuri, prepoznato je da će ograničenje pristupa internetu ili ograničenje kojim stranicama na internetu možemo da pristupimo dati pozitivne efekete u smislu cenzure. Otuda različite zemlje imaju različita ograničenja pristupa internetu (od Severne Koreje koja ne dozvoljava pristup internetu običnim građanima, do SAD koja ima pristup skoro svemu), i različite zajednice mogu uvesti još stroža pravila (univerzitetske mreže često onemogućavaju pristup sajtovima za klađenje).

13.1.4 Etički aspekti računarstva

Etika se uopšteno bavi određivanjem koje ponašanje je ispravno, a koje ponašanje nije prihvatljivo. Postoji više vrsta etike, i primećuje se postojanje razlika u etički prihvatljivom ponašanju među različitih grupa ljudi. Kako je prepoznato da određena etička pravila važe na globalnom nivou, neka od tih pravila su prevedena u zakone, tačnije postala su kažnjiva zakonom. Danas, ključna razlika između zakona i etičkih pravila je u tome što vladajuća tela sprovode ova prva, ali ne i ova druga. Opet, mnoga etička pravila su univerzalno prihvaćena, ali postoje i pravila zasnovana na stavovima i običajima neke uže zajednice. Na primer, zajednice kao što su lekarska ili advokatska, imaju detaljna etička pravila i članu zajednice koji ih krši može biti zabranjen dalji rad.

[3]Privatni nalozi su nalozi u okviru kojih korisnik maksimalno ograničava pristup svojim podacima i svom nalogu, ostavljajući najčeše pristup samo malom broju unapred poznatih ljudi.

Etička pitanja u informatici imaju dosta toga zajedničkog sa etičkim pitanjima u drugim oblastima, ali imaju i svojih specifičnosti. Etička pitanja mogu se ticati naručioca posla ili samog naručenog proizvoda (na primer, softver za ratne potrebe, softver koji se može koristiti za neovlašćeno prikupljanje podataka, itd). Etička pitanja postoje i u situacijama kada, na primer, zaposleni neovlašćeno koristi računarske resurse za lične potrebe, kada zaposleni primeti da se u njegovoj kompaniji koristi i neki nelegalno nabavljen softver, kada zaposleni zaključi da softver za medicinski uređaj svesno dozvoljava rizik po zdravlje korisnika, da je softver za merenje količine izduvnih gasova automobila napravljen da namerno radi neispravano (kao što je utvrđeno da je 2015. godine rađeno u kompaniji Volkswagen) itd. Etička pitanja mogu da postoje i u slučajevima prelaska od jednog poslodavca kod drugog koji mu predstavlja direktnu konkurenciju (što se često onemogućava ugovorom o zaposlenju).

Grupa programa za koje je jasno da krše etičke principe su svakako zlonamerni programi. Uticaj takvih programa na sistem koji napadaju može biti raznolik, počevši od jednostavnih virusa koji šamožauzimaju memorijski prostor, do opasnih virusa koji mogu u potpunosti preuzeti kontrolu nad sistemom koji napadaju (čak i isključivši pristup administratorima sistema). Osim kontrole sistema koja se na taj način dobija, ovakvo preuzimanje kontrole nad računarom može imati i drugi sloj napada kroz atak na druge računare u mreži ili kroz zloupotrebu podataka koji se na računaru nalaze (preuzimanje kontrole nad bankovnim računom ili slično).

Generalno, razmatranje i razrešavanje etičkih dilema može da bude veoma kompleksno i obično se zasniva na detektovanju svih aktera u konkretnoj situaciji, na detektovanju svih pojedinačnih etičkih pitanja, razmatranju na osnovu primera, analogije i kontraprimera. Upravo nedostatak situacija sa dovoljnim stepenom analogije često može da uzrokuje dileme u nekim situacijama koje se tiču računara. Na primer, sve aktuelnija su pitanja u vezi sa automobilima sa automatskom navigacijom. Kako taj softverski sistem treba da bude obučen – ukoliko se vozilo nađe u situaciji kada mora da riziku izloži ili vozača ili pešaka, šta će odlučiti?

Američki etički institut (engl. Computer Ethics Institute) objavio je 1992. *deset zapovesti računarske etike*:

1. Ne koristi računar da naudiš drugim ljudima.
2. Ne mešaj se nepozvan u tuđi rad na računaru.
3. Ne njuškaj kroz tuđe datoteke.
4. Ne koristi računar da ukradeš.
5. Ne koristi računar da svedočiš lažno.
6. Ne kopiraj i ne koristi softver koji nisi platio.
7. Ne koristi tuđe računarske resurse bez odobrenja ili odgovarajuće nadoknade.
8. Ne prisvajaj tuđe intelektualne rezultate.
9. Misli o društvenim posledicama programa koji pišeš ili sistema koji dizajniraš.
10. Uvek koristi računar na načine koji osiguravaju brigu i poštovanje za druge ljude.

Navedene „zapovesti“ daju neke opšte smernice i razrešavaju neka, ali nikako sva moguća etička pitanja u vezi sa računarima i programiranjem.

Elektronska verzija

Pitanja i zadaci za vežbu

Pitanje 13.1. *Koje sve aplikacije koriste „sisteme za preporučivanje“? Šta su moguće koristi a šta moguće štete od ovakvih sistema?*

Pitanje 13.2. *U kojim situacijama privatnost na internetu predstavlja moguću opasnost.*

Pitanje 13.3. *Šta opravdava a šta ne opravdava ograničavanje privatnosti na internetu?*

Pitanje 13.4. *Pronađi na internetu neke građanske grupe koje se zalažu za privatnost na internetu i kritički razmisli o njihovim stavovima.*

Pitanje 13.5. *Pronađi na internetu najznačajnije „uzbunjivače“ koji su ukazali na nelegalno prisluškivanje internet komunikacija.*

13.2 Zavisnost od interneta i društvene mreže

Iako je pojava interneta jedna od najvećih revolucija u razvoju računara, bitno je istaći i njegove dobre i njegove loše strane. Kao što važi za većinu bitnih stvari u ljudskoj istoriji, dobrobit koja se može dobiti od interneta u velikoj meri zavisi od pojedinca, njegove organizacije i stava.

13.2.1 Zavisnost od interneta

Kako je upotreba računara rasla, tako je počela da se javlja i zavisnost od interneta. Ovaj pojam pogotovo dobija na značaju od pojave pametnih mobilnih telefona, jer internet sada postaje još prisutniji i maltene nezamenljiv u mnogim sferama života. Počevši od navigacije na telefonu (koja je u potpunosti zamenila nekadašnje papirne mape), preko Viber i Watsupp aplikacija za dopisivanje i razmenu videa i slika, do različitih društvenih mreža koje dobijaju neprikosnoveni značaj u poslednjih par godina. Naravno, kako je većina stvari na internetu veoma korisna jako je teško ustanoviti trenutak kada neko razvija zavisnost od interneta. Uglavnom se misli na prekomerno korišćenje kompjutera u cilju igranja igrica, provođenja previše vremena na društvenim mrežama, komuniciranje sa ljudima isključivo preko računara. Kako je danas moguće gotovo ceo život organizovati online, postoji mogućnost da neko nedeljama, mesecima ili čak i godinama ne izlazi iz kuće zahvaljujući pogodnostima koje internet pruža što svakako nije bila inicijalna zamisao. Ukratko, svako nekontrolisano i kontinuirano ponašanje koje izuzetno šteti (fizičkim ili psihičkim) aspektima života

Testovi za prepoznavanje zavisnosti od interneta su uglavnom nastali modifikacijom testova za prepoznavanje zavisnosti od kockanja. Na osnovu Kit Bird (engl. Keith Beard) narednih pet kriterijuma su zabrinjavajući i predstavljaju signal za postojanje zavisnosti:

- Konstantno razmišljanje o korišćenju interneta
- Duže ostajanje na mreži od planiranog
- Bezuspešni pokušaji kontrolisane upotrebe interneta
- Zadovoljstvo se stiče tek nakon dužeg vremena provedenog za internetom
- Pokušaj kontrolisane upotrebe interneta izaziva depresivno ili razdražljivo ponašanje

Dodatno, ako još jedno od narednih pitanja ima pozitivan odgovor, to se smatra dovoljnim da bi se postavila dijagnoza:

- Laganje drugih ljudi o korišćenju interneta
- Korišćenje interneta da bi se pobeglo od problema
- Rizikovanje gubitka veze ili posla zbog interneta

Posledice internet zavisnosti mogu biti depresija, anksioznost, neorganizovanost, problemi sa snom i razni fizički propratni problemi.

Ističu se dva načina korišćenja interneta koja proizvode najviše zavisnika, a to su igranje online igrica i online kockanje.

Igranje online igrica omogućava korisnicima da se povežu sa velikom bazom igrača, što s vremenom dovodi do formiranja velike online zajednice u kojoj je status postao nenadoknadiv ako se ne provodi dovoljno vremena u igrici. Vreme koje je potrebno uložiti da bi igrač postao dovoljno iskusan u igrici i "jak" je dovoljno da se razvije zavisnost, i pogotovo je primetan kod ljudi koji ispoljavaju određeno nezadovoljstvo ličnim i profesionalnim životom. Virtuelni život koji igrač razvija kroz svoj virtuelni lik je često mnogo ispunjeniji, zanimljiviji i na neki način njemu samom vredniji od njegovog stvarnog života koji onda dodatno biva zanemaren i stagnira.

Online klađenje bi trebalo biti zabranjeno (kao i uživo klađenje) osobama ispod 18 godina, međutim online pristup olakšava smanjenje ove granice pristupa. Osim vremena koje se troši u toku klađenja (nikada se ne postavlja samo jedna opklada, i nikada se ne dešava da to bude samo jedan put), ovakva aktivnost može doneti ozbiljne materijalne štete porodici.

Elektronska verzija

13.2.2 Društvene mreže

Kultura društvenih mreža danas uzima maha više nego ikad, u potpunosti narušavajući privatnost pojedinca uz njegovu potpunu saglasnost. Osim što postavljaju svoje slike, mnogi ljudi danas kače slike i lokacije treninga svoje maloletne dece što može biti jako opasno. Drugi roditelji otvaraju deci naloge na društvenim mrežama odmah po rođenju i ne shvatajući da na taj način u potpunosti izlažu identitet svog deteta potencijalno zlonamernim ljudima.

Posledica preterane upotrebe telefona i poražavajućih negativnih uticaja je pogotovo izražena kod mladih i male dece. Što više vremena provedenog u virtuelnoj sferi, znači sve manje i manje vremena provedenog na igralištima, sve manje kretanja i sve manje komunikacije licem u lice. Tako da su osim mentalnih problema (slaba koncentracija, nemogućnost ostvarivanja kontakata) sve prisutniji i fizički problemi (loše držanje, slabi mišići, problemi sa vidom i slično).

Osim negativnih aspekata koji utiču direktno na osobu, jako je važno istaći i opasnosti koje vrebaju sa interneta u vidu lakog otvaranja i održavanja lažnih profila (kroz zloupotrebu tuđeg indentiteta, kada se na primer odrasle osobe predstavljaju kao vršnjaci tinejdžerima u cilju manipulacije i iskorišćavanja).

13.3 Pravni i ekonomski aspekti računarstva

Razni aspekti programiranja i primene računara pokrivene su zakonima i drugim sličnim normama. Ipak, i nakon istorije računarstva duge više od sedam decenija, mnoge situacije, na primer, u vezi sa autorskim pravima vezanim za softver, izazivaju dileme ili vode do kompleksnih sudskih procesa. To je samo jedan od mnogih pravnih i ekonomskih aspekata računarstva.

13.3.1 Intelektualna svojina i njena zaštita

Intelektualna svojina odnosi se na muzička, likovna, književna dela, simbole, dizajn, otkrića i pronalaske, itd. Intelektualna svojina štiti se raznovrsnim pravnim sredstvima, kao što su autorsko pravo ili kopirajt (engl. copyright), patenti, zaštićeni registrovani simboli (engl. trademarks), itd. Dok se autorsko pravo odnosi na lično, neotuđivo pravo autora i njegovog dela, kopirajt se odnosi na konkretno delo i može menjati vlasnika.

Autorska prava. Autorska prava na softver treba da spreče neovlašćeno korišćenje i kopiranje softvera. Nosilac autorskih prava ima pravo kopiranja, modifikovanja i distribuiranja softvera, što može odobriti i drugima. U Evropskoj uniji, na računarski softver polažu se autorska prava, obezbeđena zakonom, isto kao na, na primer, književna dela i to bez ikakve registracije ili formalne procedure. Takva autorska prava odnose se na sve aspekte kreativnosti autora, ali ne i na ideje na kojima je program zasnovan, algoritme, metode ili matematičke pojmove koje koristi. Dakle, autorska prava štite samo program u formi (u izvornom kodu) u kojoj je napisan od strane programera. Funkcionalnost programa, programski jezik, format datoteka koje program koristi nisu zaštićeni autorskim pravom. S druge strane, grafika, zvuci i izgled programa mogu biti predmet autorskih prava, a skup funkcionalnosti programa može biti zaštićen patentom.

U većini zemalja, podrazumeva se da autorska prava pripadaju autorima dela, pri čemu se pod autorima podrazumevaju poslodavci koji su svojim zaposlenim dali zadatak da naprave softver. Autorska prava na softver koji napravi zaposleni, ali ne po zadatku i uputstvima poslodavca (na primer, u slobodno vreme) pripadaju zaposlenom. Stvari postaju komplikovanije u slučaju kada poslodavac za razvoj softvera angažuje spoljašnjeg saradnika, a još komplikovanije ako je takav saradnik isporučio naručeni program a nije dobio dogovoreni honorar. Zbog takvih situacija, umesto oslanjanja na opšte zakone, bolje je unapred sklopiti namenski i precizan ugovor.

Sa postojanjem interneta, postalo je popularno i krajnje prihvatljivo deljenje sadržaja koje je kreirao neki drugi korisnik. Na taj način se često povećava vidljivost oba korisnika (i onog koji je kreirao originalni sadržaj, i onog koji taj sadržaj deli), međutim jedno vreme su postojale situacije kada su velike platforme zarađivale dosta novaca od korisnika pri čemu su sav profit zadržavali za sebe. Danas to više nije moguće, zahvaljujući zakonskoj regulativi koja zahteva da se novac podeli ravnomerno. Naravno, postoje načini da se i to zaobiđe, ali o tome više u kursu koji posebno obrađuje ove teme.

Patenti. U većini zemalja, računarski program ne može se registrovati patentom. Razlog je to što se svaki program, u manjoj ili većoj meri, oslanja na kumulativni razvoj i korišćenje tuđeg rada, ali i zbog toga što je teško ili nemoguće kontrolisati buduće korišćenje programa koji je patentiran. U Evropskoj uniji, patentom se ne može zaštititi računarski program kao takav, ali mogu pronalasci koji uključuju upotrebu računara i namenskog softvera.

Elektronska verzija (2024)

Poverljivost. Ako se algoritam ne može zaštititi (od budućeg neovlašćenog korišćenja na navedene načine), onda se preporučuje da se on čuva u tajnosti i štiti na taj način. On se onda čini dostupnim za korišćenje ili modifikacije na vrlo ograničen način, ograničenom skupu osoba i u skladu sa ugovorom o neotkrivanju i o poverljivosti (engl. non-disclosure or confidentiality agreements; NDAs).

Razumevanje i primena pomenutih pravila i zakona često nije jednostavno i pravolinijsko, te su česti sudski sporovi o pojedinim programima, idejama i slično, a koji uključuju i softverske gigante kao što su Microsoft i Apple.

13.3.2 Kršenja autorskih prava

Kršenje autorskih prava (engl. copyright infringement) je korišćenje autorskog dela kao što je softver bez ovlašćenja, uključujući reprodukovanje, distribuiranje, modifikovanje, prodavanje i slično. Za kršenja autorskih prava često se koristi termin piraterija (engl. piracy). Neki od čestih razloga za kršenje autorskih prava su: cena, nedostupnost (na primer, u nekoj zemlji), pogodnost (na primer, ukoliko legalnu verziju nije moguće dobiti internetom), anonimnost (ukoliko je za legalnu verziju neophodno identifikovanje), itd. Slučajevi kršenja autorskih prava se često razrešavaju neposrednom pogodbom ili sudskim procesom.

Zbog digitalnog zapisa i interneta, kopiranje softvera i umetničkih dela često je veoma jednostavno i omogućava masovnu pirateriju. Procenjuje se da je u 2016. godini, čak oko 39% programa na personalnim računarima bilo nelicencirano. Na osnovu jedne velike ankete iz 2017. godine, čak 57% pojedinačnih korisnika barem ponekad koristi piratski softver. Kompanija Google dobija dnevno oko dva i po miliona zahteva nosioca autorskih prava za uklanjanje linkova na piratske verzije njihovih proizvoda.

Autorska prava obično se mogu ignorisati i autorsko delo se može kopirati bez eksplicitnog ovlašćenja u nekim specijalnim nekomercijalnim situacijama koje se smatraju „fer korišćenjem" (engl. fair use), na primer, u okviru predavanja (kada se kreira nekoliko primeraka softvera za korišćenje na času), izveštavanja u medijima, naučnim istraživanjima i slično. Granica za fer korišćenje često nije jasna, pa se često pod maskom fer korišćenja distribuiraju čitava umetnička dela ili računarski programi.

13.3.3 Vlasnički i nevlasnički softver

Vlasnički softver (engl. proprietary software) je softver čiji je vlasnik pojedinac ili kompanija (obično neko ko je softver razvio), postoje oštra ograničenja za njegovo korišćenje i, gotovo isključivo, njegov izvorni kôd čuva se u tajnosti.

Softver koji nije vlasnički pripada obično nekoj od sledećih kategorija:

Šerver softver (engl. shareware software) distribuira se po niskoj ceni ili besplatno za svrhe probe i testiranje, ali zahteva plaćanje i registraciju za legalno, neograničeno korišćenje i neku vrstu tehničke podrške. Autorska prava na šerver softver zadržavaju originalni autori i nije dozvoljeno modifikovati ili dalje distribuirati softver.

Friver softver (engl. freeware software) se distribuira besplatno i to su obično mali pomoćni programi, bez obezbeđene tehničke podrške. Autorska prava na friver softver zadržavaju originalni autori.

Javni softver (engl. public software) ili softver javnog domena (engl. public domain software) ne podleže autorskim pravima, objavljuje se bez ikakvih ograničenja za njegovo korišćenje i nema nikakvu obezbeđenu tehničku podršku.

Softver otvorenog koda (engl. open source software) ili besplatni, slobodni softver (engl. free software)[4] je softver kojem se besplatno može pristupiti, koji se besplatno može koristiti, modifikovati i deliti (u originalnoj ili izmenjenoj formi) od strane bilo koga. Softver otvorenog koda obično je razvijan od strane velikog broja autora i distribuira se pod odgovarajućom licencom. Licence za otvoreni kôd oslanjaju se na autorska prava.

Softver otvorenog koda nije vlasnički softver (engl. proprietary softver), ali može biti komercijalan, tj. prodavati se. Ako je neko preuzeo neki softver pod licencom za otvoreni kôd, može da ga prodaje, ali je u principu u obavezi da ga distribuira pod istom tom licencom.

Licence za softver otvorenog koda odnose se na sledeća pitanja i kriterijume:

1. Slobodno redistribuiranje;

[4] Neki autori prave razliku između pojmova „open source software" i „free software", naglašavajući da se reč „free" (koja ima više značenja) u ovom drugom odnosi na slobodu, slobodu upotrebe, a ne na cenu upotrebe.

2. Raspoloživost programa u izvornom kodu;
3. Dozvola za dela izvedena iz originalne verzije;
4. Nepovredivost autorskog izvornog koda;
5. Nema diskriminisanja prema osobama ili grupama;
6. Nema diskriminisanja prema polju primene;
7. Redistribuiranje licence uz redistribuirani program;
8. Licenca ne može biti specifična za konkretan proizvod;
9. Licenca ne može da ograničava drugi softver;
10. Licenca mora da bude neutralna u odnosu na tehnologiju.

Neke od najčešće korišćenih okvira za licenciranje softvera otvorenog koda su GNU General Public License (GPL), Apache License, Creative Commons Licenses, itd.

13.3.4 Nacionalna zakonodavstva i softver

Iako postoje mnogi univerzalni principi u zakonskom regulisanju oblasti računarstva, za zakonodavstva mnogih zemalja postoje i brojne specifičnosti. U nekim zemljama zabranjene su određene internet usluge (na primer, u Kini je nedostupno internet pretraživanje putem pretraživača Google), u nekim zemljama zabranjen je uvoz ili izvoz nekih vrsta softvera (na primer, ograničen je izvoz kriptografskog softvera iz SAD), a u nekim zemljama zabranjen je izvoz nekih vrsta podataka ili njihova obrada u inostranstvu (u Evropskoj uniji, pravni akt GDPR, *The General Data Protection Regulation*, iz 2018. godine propisuje niz pravila o upravljanju ličnim podacima građana EU – o njihovoj zaštiti, kao i o strogoj kontroli izvoza ili obrade izvan zemalja EU). Zbog toga, u svim računarskim poslovima koji se sprovode u više zemalja neophodno je voditi računa o specifičnostima njihovih zakonodavstava.

Iako možda na prvi pogled deluju neosnovane, ove akcije su u stvari reakcije na situacije u kojima su se računari, i društvene mreže, koristili za uticaj na javno mišljenje ili stav o određenim događajima (uključujući na primer i uticaj na izbore). Ovakve situacije se kreiraju kroz sinhronizovanje velikog broja lažnih naloga[5] sa ciljem širenja lažnih (najčešće uznemirujućih) vesti. Ovakve situacije mogu biti kreirane za uticaj jedne vlade na drugu, ali takođe i za uticaj vlade na sopstveno stanovništvo.

Odgovornost u ovakvim situacijama, nije inicijalno na običnom građaninu, ali njegova je dužnost i u neku ruku i obaveza da svim informacijama koje se danas mogu naći na internetu i društvenim mrežama pristupa sa izrazitim kritičkim stavom i da se trudi da pre svega u bitnim situacijama donese informisanu odluku.

13.3.5 Sajber kriminal

Sajber ili računarski kriminal (engl. cyber crime) je svaka protivzakonita aktivnost koja se sprovodi putem računara. U mnogim zemalja postoje zakoni koji se odnose na sajber kriminal i kazne mogu da budu novčane ili zatvorske u trajanju i do 20 godina. Zakoni obično tretiraju dela sajber kriminala učinjena za sticanje komercijalne prednosti, za lični novčani dobitak ili za pripremu kriminalne radnje. Neka od sredstava kojima se radnje sajber kriminala mogu sprovesti su: krađa identiteta, zloupotreba mejla i spam, neovlašćeni upadi u sisteme, distribuiranje nelegalnih sadržaja, itd.

Sajber kriminal javlja se na ogromnoj skali i u raznim vidovima, počev od pojedinačnih prevara, pa do napada na najvišem nivou, kada jedna država vrši upade u računarske sisteme druge države.

Individualni korisnici mogu donekle da se potrude da zaštite svoje naloge i lične informacije, međutim uprkos tome se često dešavaju različite vrste prevara od kojih ćemo neke navesti ovde.

- *Pecanje (engl. phishing).* Napadi koji za cilj najčešće imaju preuzimanje identiteta korisnika koje se odvija kroz kreiranje sajta vizuelno veoma nalik originalnom sajtu koji korisnik želi da upotrebi, u cilju preuzimanja šifre korisnika ili njegovih ličnih podataka kao što su broj telefona, broj bankovnog računa i slično[6].
- *Preuzimanje kontrole nad mobilnim telefonom.* Kako je u poslednje vreme verifikacija svih online kupovina uglavnom povezana sa mobilnim telefonom, ukoliko neko uspe da dođe do broja bankovne kartice i uspe da preuzme mobilnu karticu, ima potpunu kontrolu nad potrošnjom. U nekim situacijama ispostavlja se da je dovoljno da se osoba javi na poziv, da dođe do potpunog preuzimanja kartice.

[5] Lažni nalozi su danas poznatiji kao botovi.

[6] Male izmene u nazivu sajta koje olakšavaju kreiranje ovakvih sajtova su zamena slova ili broja vizuelno sličnom oznakom kao što je na primer zamena slova o brojem 0.

Elektronska verzija (2024)

- *Lažni profili na društvenim mrežama.* Društvene mreže su inicijalno imale za cilj povezivanje grupa ljudi sličnih interesovanja, da biste otvorili poziv bilo je potrebno da vas pozove neko ko već ima nalog, i vidljivost profila je isključivo bila dopuštena među prijateljima. S vremenom društvene mreže su postale alat za preuzimanje identiteta ljudi, za kreiranje lažnih profila koji se naknadno koriste za izazivanje svađa, izazivanje, napade lične prirode i slično.

- *Spam.* Iako sam po sebi spam ne predstavlja prevaru, ako se koristi u organizovanom obliku spam može biti upotrebljen da napravi ozbiljne probleme. Možda na prvi pogled najnaivniji način na koji nam spam šteti jeste kroz vreme koje potrošimo na čitanje mail-a koji nije namenjen nama lično ili koji ne želimo da primimo. Međutim, ako su te situacije česte, to može dovesti do puno izgubljenog vremena, pogotovo ako se posmatra ukupno vreme svih zaposlenih u velikoj firmi (spam će stići svim tim osobama, što značajno može da uspori celokupni radni dan). Sa druge strane putem spama možemo dobiti neki problematični link koji nas potencijalno može usmeriti ka nekoj ozbiljnijoj prevari. Otuda mnoge kompanije koriste spam filtere, koji opet nisu idealni, pa ima smisla s vremena na vreme proveravati spam folder za slučaj da neki nama koristan mail završi tamo.

Pored sajber kriminala, postoje i mnoge vrste sajber nasilja (engl. cyberbullying), kao što je vršnjačko sajber nasilje, sajber uhođenje i slično, koje se takođe tretiraju zakonima u mnogim zemljama, kao i pratećim policijskim jedinicima za sajber kriminal.

13.3.6 Računarstvo, produktivnost i nezaposlenost

Kao što je slučaj i sa mnogim drugim tehnologijama, pojava i razvoj računara i njihovih primena doveli su do otpuštanja mnogih ljudi i gubljenja mnogih poslova. Međutim, to je samo jedan aspekt i pravo pitanje je da li razvoj računara dovodi do ukupno većeg ili manjeg broja radnih mesta za ljude. Slična dilema postojala je i u vreme industrijske revolucije, kada su mašine počele da u proizvodnji zamenjuju ljude i kada su, početkom devetnaestog veka, u strahu od nezaposlenosti, pripadnici ludističkog pokreta uništavali proizvodne mašine. Vreme je, međutim, pokazalo da su, na duže staze, pojava i razvoj mašina omogućile veliki porast produktivnosti, porast proizvodnje i veliki porast radnih mesta. Zato mnogi smatraju da će računari, iako će na mnogim radnim mestima odmeniti ljude, zapravo stvoriti mnogo više novih radnih mesta – za razvoj tih računara i upravljanje njima, kao i za čitav niz usluga koje danas i ne postoje na tržištu.

13.3.7 Kriptovalute

Od pre nekoliko godina postoji još jedan način na koji računari utiču na globalnu ekonomiju – kripto valute. Kripto valute su digitalno sredstvo plaćanja. Za kontrolu stvaranja novih jedinica valute, kao i za sigurnost i registrovanje transakcija koriste se kriptografski algoritmi. Za razliku od klasičnih nacionalnih valuta koje izdaju centralne banke, kripto valute nemaju centralnog izdavača niti centralizovanu kontrolu toka. Kontrola svake kriptovalute odvija se kroz blokčejn (engl. blockchain) tehnologiju, kao javnu, distribuiranu bazu podataka. Ta baza podataka ne čuva se na jednoj lokaciji i ne postoji centralizovana verzija koja može biti oštećena ili uništena. Ta baza podataka zapravo je deljena između miliona računara i dostupna svakome putem interneta. U okviru blokčejna, registruje se svaka transakcija u vidu novog bloka i, zahvaljujući kriptografskoj zaštiti, ne može kasnije biti izbrisana.

Bitcoin je prva kripto valuta, kreirana 2009. godine. Od tada se pojavilo na stotine drugih kripto valuta. Uprkos mnogim turbulencijama na tržištu kripto valuta, njihova vrednost tokom prethodnih godina je uglavnom rasla. Teško je predvideti njihovu dugoročnu sudbinu.

Pitanja i zadaci za vežbu

Pitanje 13.6. *Za svoje omiljene aplikacije proveri pod kojom licencom se distribuiraju.*

Pitanje 13.7. *Istraži na internetu koje su kriptovalute trenutno najpopularnije.*

Pitanje 13.8. *Istraži na internetu najznačajnije sajber napade među različitim zemljama. Kakav je bio stav vlada tim zemljama u vezi sa tim napadima?*

Pitanje 13.9. *Istraži na internetu osnovne principe blokčejn tehnologije.*

Elektronska verzija (2024)

Spisak preuzetih slika

Slika **??** napravljena je po uzoru na sliku iz kolekcije [24].

Slika **??** napravljena je po uzoru na sliku iz kolekcije [24].

Slika **??** preuzeta je sa Vikipedije.

Slika **??** preuzeta je uz modifikacije iz izvornog rada [9].

Elektronska verzija (2024)

Elektronska verzija (2024)

Literatura

[1] D. Bahdanau, K. Cho, and Y. Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *ICLR 2015*, 2015.

[2] A. Biere, M. Heule, H. Van Maaren, and T. Walsh. *Handbook of Satisfiability*. IOS Press, 2009.

[3] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[4] Z. Cao, G. Hidalgo, T. Simon, S.-E. Wei, and Y. Sheikh. Openpose: Realtime Multi-Person 2D Pose Estimation Using Part Affinity Fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[5] D. Gabbay, C. J. Hogger, and A. Robinson. *Handbook of Logic in Artificial Intelligence and Logic Programming*. Clarendon Press, 1998.

[6] B. Goertzel, N. Geisweiller, L. Coelho, P. Janičić, and C. Pennachin. *Real-World Reasoning: Toward Scalable, Uncertain Spatiotemporal, Contextual and Causal Inference*. Atlantis Thinking Machines, 2011.

[7] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[8] P. Janičić. *Matematička logika u računarstvu*. Matematički fakultet, 2009.

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 1998.

[10] G. Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Pearson, 2009.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglu, H. King, D. Kumaran, D. Wiestra, S. Legg, and D. Hassabis. Human-Level Control Through Deep Reinforcement Learning. *Nature*, 2015.

[12] K. Murphy, N. Ye, S. Guadarrama, S. Liu, and Z. Zhu. Improved Image Captioning via Policy Gradient Optimization of SPIDEr. *ICCV 2017*, 2017.

[13] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang. Autonomous Inverted Helicopter Flight via Reinforcement Learning. *Experimental Robotics IX*, 2006.

[14] M. Nikolić and A. Zečević. *Mašinsko učenje*. U pripremi, 2021.

[15] F. Petroni. Language Distance and Tree Reconstruction. *Journal of Statistical Mechanics Theory and Experiment*, 2008.

[16] M. Popova, O. Isayev, and A. Tropsha. Deep Reinforcement Learning for de Novo Drug Design. *Science Advances*, 2018.

[17] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *CVPR 2016*, 2016.

[18] A. Robinson and A. Voronkov. *Handbook of Automated Reasoning*. Elsevier, 2001.

[19] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 2020.

Elektronska verzija (2024)

[20] A. Shi. Visualize How a Neural Network Works from Scratch. Towrads Data Science, https://towardsdatascience.com/visualize-how-a-neural-network-works-from-scratch-3c04918a278, 2020.

[21] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake. Real-Time Human Pose Recognition in Parts from Single Depth Images. *CVPR 2011*, 2011.

[22] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural Networks. *NIPS 2014*, 2014.

[23] Maloof, A. Marcus and others Machine learning and data mining for computer security: methods and applications. *Springer*, 2006.

[24] P. Veličković. Collection of PGF/TikZ figures. https://github.com/PetarV-/TikZ, 2018.

Elektronska verzija (2024)