Programiranje I, I tok, šk. 2014/15. g.

dr Gordana Pavlović-Lažetić

Sadržaj

1 Uvod - računarstvo i programiranje **7**
1.1 Računarstvo 7
1.1.1 Podoblasti računarstva 8
1.2 Programiranje 9
1.3 Informacione tehnologije 10
1.4 Istorija informacionih tehnologija i sistema 12
1.4.1 Generacije savremenih elektronskih računara 14

2 Računarski sistemi **17**
2.1 Fon Nojmanova (von Neumann) mašina 17
2.2 Struktura savremenog računarskog sistema 19
2.2.1 Tehnički sistem računara 19
2.2.2 Programski sistem računara 23
2.3 Funkcionisanje računarskog sistema 25

3 Karakterski skup i kodne sheme **27**
3.1 Karakterski skup 27
3.2 Kodne sheme 27
3.2.1 7-bitne kodne sheme 28
3.2.2 Proširenja 7-bitnog kôda 29
3.2.3 Industrijski standardi 30
3.2.4 Unicode 31

4 Azbuka, reč, jezik, jezički procesori **33**
4.1 Azbuka, niska 33
4.2 Jezik 33
4.3 Jezički procesori 35
4.3.1 Prednosti i nedostaci interpretera i kompilatora 37

5 Rešavanje problema uz pomoć računara **39**
5.1 Specifikacija 42
5.1.1 Dvovalentna logika 42
5.1.2 Formalna specifikacija 43

5.2 Algoritam – intuitivni pojam 43
5.2.1 Algoritam razmene – swap 45

6 Pregled programskog jezika C **47**
6.1 Filozofija jezika 47
6.2 Struktura programa 48
6.3 Primer programa 49
6.4 Preprocesor C-a 50
6.5 Tipovi podataka 51
6.6 Ključne reči 54
6.7 Izrazi 54
6.8 Iskazi 56
6.9 Ulaz-izlaz 57

7 Osnovni tipovi podataka **61**
7.1 O tipovima 61
7.2 Osnovni tipovi u C-u 62
7.2.1 Znakovni tip 63
7.2.2 Celobrojni tip 65
7.2.3 Logički tip 68
7.2.4 Realni tip 69
7.3 Izrazi 72
7.3.1 Operatori nad bitovima 73
7.3.2 Konverzija tipova 74
7.4 Prioritet operatora. 75

8 Formalni opis konstrukcija jezika **77**
8.1 Formalna gramatika 77
8.2 Meta jezici 79
8.2.1 Bekus-Naurova forma 80
8.2.2 Još primera 82

9 Iskazi – upravljanje **83**
9.1 Jednostavni iskaz: iskaz dodele 83
9.2 Serijski komponovani iskaz 84
9.3 Uslovni iskazi (selektivna, alternirajuća upravljačka struktura) . . . 85
9.3.1 Uslovni iskaz sa višestrukim grananjem 87
9.4 Repetitivni iskazi 88
9.4.1 Iskaz ciklusa sa pred-proverom uslova 88
9.4.2 For iskaz 89
9.4.3 Iskaz ciklusa sa post-proverom uslova 91
9.4.4 Ekvivalentnost repetitivnih iskaza while i do-while 93
9.5 Iskazi bezuslovnog skoka 94

10 Funkcije **97**
10.1 Definicija, deklaracija (prototip) i poziv 97
10.2 Spoljašnje (globalne) i unutrašnje (lokalne) promenljive 100
10.2.1 Statičke promenljive 103
10.2.2 Inicijalizacija 103
10.2.3 Konflikt identifikatora 104
10.3 Parametri 105
10.4 Direktive preprocesora C-a 106
10.5 Pokazivači i argumenti funkcija 108
10.6 Biblioteke funkcija 110

11 Nizovi **111**
11.1 Nizovi karaktera - niske, stringovi 111
11.2 Pokazivači i nizovi 113
11.3 Funkcije i pokazivači na karaktere 115
11.4 Višedimenzioni nizovi 118
11.5 Razvoj algoritama sa nizovima 119
11.5.1 Konverzija broja iz dekadnog u binarni oblik 119
11.5.2 Efikasnost algoritama 120
11.6 Pretraživanje nizova 120
11.6.1 Linearno pretraživanje 120
11.6.2 Binarno pretraživanje 122
11.6.3 Interpolirano pretraživanje 123
11.7 Sortiranje 124
11.7.1 Elementarne metode sortiranja 124
11.7.2 Sortiranje izborom najvećeg elementa 125
11.7.3 Sortiranje umetanjem 126
11.7.4 Šelsort 128
11.7.5 Bubble sort 129

12 Strukture **131**
12.1 Strukture i funkcije 132
12.2 Nizovi struktura 133
12.3 Pokazivači na strukture 135
12.4 Unija 136
12.5 Bit-polje 138

13 Datoteke – ulaz i izlaz **139**
13.1 Standardni ulaz i izlaz 139
13.1.1 Formatirani izlaz 140
13.1.2 Formatirani ulaz 141
13.2 Pristup datotekama 141
13.3 Argumenti u komandnoj liniji 143

1

Uvod - računarstvo i programiranje

1.1 Računarstvo

Ma koliko različite bile, sve definicije računarstva se manje-više slažu u konstataciji da računarstvo deli zajedničke naučne metode sa matematikom, prirodnim naukama i tehnikom. Paradigma koja ga vezuje za matematiku je teorija, sa prirodnim naukama vezuje ga eksperimentalni naučni metod, dok je projektovanje zajedničko za računarstvo i tehniku. To se ogleda i kroz istoriju razvoja računarstva, ali i kroz savremeni razvoj računarstva kao spoja teorije algoritama, matematičke logike i programabilne elektronske računske mašine – računara.

Kratko rečeno, računarstvo je *sistematično izučavanje algoritamskih procesa koji opisuju i transformišu informaciju: njihove teorije, analize, projektovanja, efikasnosti, implementacije i primene.* U središtu računarstva je računar. On se može opisati (prema Enciklopdeiji Larousse) kao *mašina za transformaciju informacije iz jednog oblika u drugi.* To znači da je sam računar jedna matematička mašina koja se ponaša prema strogo definisanim principima i zakonima, i da je potpuno nezavisan od svog spoljašnjeg oblika koji se kroz istoriju veoma menjao, i koji se i dalje dramatično menja. Danas se pod računarom podrazumeva elektronska mašina koja može da se programira za izvršavanje različitih zadataka.

Naučni progres računarske nauke se značajno komplikuje zbog rasta računarske tehnologije i uzastopnih revolucija komponenti računara. Tehnološka baza računarstva menja se veoma brzo, i sve što se uči *ad hoc* zastareva vrlo brzo.

Ipak, računarska tehnologija čini računarstvo jednim od stubova savremenog društva. Informaciono društvo, kao oblik društvenog uređenja u ovom veku, nameće nove kriterijume razvoja privrede i društva, a oni se baziraju na brzini stvaranja informacije i dostupnosti informaciji. Osnova globalne informacione infrastrukture je internet, pa se veliki broj privrednih grana orijentiše ka njegovom korišćenju

i uključuje u digitalnu ekonomiju i elektronsko poslovanje. Život na društvenim mrežama nad internet infrastrukturom simulira pravo životno okruženje.

1.1.1 Podoblasti računarstva

Klasifikaciona šema Računarskog pregleda (Computing Reviews) iz 2012. godina, najvećeg računarskog udruženja, Association for Computing Machinery (ACM), koji prikazuje sve relevantne publikacije i rezultate u celokupnoj oblasti računarstva u svetu, prepoznaje 13 podoblasti od kojih svaka ima hijerarhijsku strukturu. Deo te hijerarhijske strukture oblasti računarstva predstavljaju sledeće podoblasti:

OPŠTE TEME
HARDVER
ORGANIZACIJA RAČUNARSKIH SISTEMA
  Arhitekture
  Sistemi u realnom vremenu
  ...
MREŽE
  Mrežne arhitekture
  Mrežni protokoli
  Mrežni algoritmi
  ...
SOFTVER I SOFTVERSKI INŽENJERING
  Programske tehnike
  Formalne metode
  Programski jezici
  Operativni sistemi
  ...
TEORIJA IZRAČUNAVANJA
  Analiza algoritama i složenost problema
  Matematička logika i formalni jezici
  Modeli izračunavanja
  Strukture podataka
  ...
MATEMATIKA IZRAČUNAVANJA
  Matematička analiza
  Diskretna matematika
  Verovatnoća i statistika
  ...
INFORMACIONI SISTEMI
  Modeli i principi
  Upravljanje bazama podataka
  Pretraživanje informacija
  Upitni jezici
  Aplikacije informacionih sistema
  Veb (www)

...
BEZBEDNOST I PRIVATNOST
Kriptografija
...
IZRAČUNAVANJE OKRENUTO ČOVEKU
Interakcija čovek – računar
...
METODOLOGIJE IZRAČUNAVANJA
Veštačka inteligencija (obrada prirodnog jezika)
Računarska grafika
Mašinsko učenje
Obrada slika
Prepoznavanje oblika
Simulacija i modeliranje
...
PRIMENENJENO RAČUNARSTVO
Elektronska trgovina
Forenzika
Pravo i društvene nauke
CAD (projektovanje uz pomoć računara)
Bioinformatika
...
DRUŠTVENE I PROFESIONALNE TEME
Računarska industrija (i računarski standardi)
Računari i obrazovanje ...
Intelektualna svojina
...

1.2 Programiranje

Značajna komponenta računarstva, prisutna u gotovo svim granama računarstva, jeste programiranje. Ipak, izjednačavanje računarstva sa programiranjem zanemaruje druge bitne aspekte računarstva, koji nisu programiranje, kao što su projektovanje hardvera, arhitektura sistema, projektovanje nivoa operativnih sistema, struktuiranje baze podataka za specifične aplikacije, validacija modela, itd.

Programiranje je u najopštijem smislu aktivnost izrade programa za elektronsku računsku mašinu (računar).

Programiranje, pored izučavanja važnih klasa algoritama i programa koji se, kao česte komponente drugih programa mogu primeniti u rešavanju novih zadataka, mora da uključi i metode razvoja algoritama i programa sa unapred poznatim ponašanjem, tj. metode koje omogućuju da se, zajedno sa izvođenjem programa izvodi i dokaz njegove korektnosti (njegova semantika), što približava razvoj programa dokazu teoreme u matematici. Ponašanje programa sada se izvodi iz samog

(statičkog) teksta programa, bez interpretacije izvršnog programa, tj. bez testiranja.

Poslednjih decenija, programski sistemi kod kojih je kritična bezbednost (npr. kontrola lêta), razvijaju se upravo korišćenjem ovakvih metoda. One (za razliku od testiranja) mogu egzaktno da dokažu korektnost programa, ali i da pomognu u otkrivanju (eventualnih) grešaka.

Koliko je značajan ovaj pristup programiranju ilustruje i sledeći primer (R.Sethi: Programming Languages – Concepts and Constructs). Jula 1962. godine, raketa koja je nosila Mariner I (Venerinu sondu bez posade), morala je da bude uništena 290 sekundi posle lansiranja. Gubitak je procenjen na 18-20 miliona dolara. Uzrok je bio greška u programskom fragmentu koji je trebalo da ima oblik:

if not (raketa u kontaktu sa radarom) **then**

ne korigovati putanju leta

Reč "**ne**" je greškom ispuštena pa je glavni računar nastavio sa korekcijom putanje i kada je raketa izgubila kontakt sa radarom. Program je prethodno bio korišćen bez problema u četiri lansiranja na Mesec.

Glavni argument koji je korišćen u odbrani ovog slučaja bila je činjenica da je program uspešno prošao tri stotine nezavisnih testiranja. To, očigledno, nije bilo dovoljno pouzdano, pa je zaključak o neophodnosti "da se nešto preduzme da se slične greške nikada više ne bi ponovile" pretočen u primene metoda dokazivanja korektnosti programa iz njegove interne (tesktualne) strukture, bez potrebe za testiranjem.

1.3 Informacione tehnologije

Računarstvo se u savremenom društvu razvija pre svega u okviru računarske tehnologije kao jedne od informacionih tehnologija. Pod informacionim tehnologijama podrazumevaju se postupci, metode i tehnike prikupljanja, prenosa, obrade, čuvanja i prezentacije informacija. Te tehnologije danas uključuju sredstva kao što su senzorni uređaji kakvi su uređaj za merenje aerozagađenja, čitač bar kôda na proizvodima, ekran osetljiv na dodir, preko telefonskih i kablovskih mreža i faks mašina do računara svih vrsta. Informacione tehnologije se primenjuju za, na primer, kreiranje i praćenje dokumenata u kancelarijama, kontrolu proizvodnje u fabrikama, projektovanje novih proizvoda, popravku automobila i druge složene opreme, prodaju proizvoda širom sveta, itd.

Informacione revolucije. Od pronalaska pisma pre oko 5000 godina, do današnjih dana, najmoćnije i najdalekosežnije tehnologije bile su pisana reč, novine i knjige, zasnovane na štamparskoj presi. Moderni svet kakav poznajemo zavisi od štampane reči. Štamparska presa ("štamparija"), konstruisana u 15. veku, predstavljala je prvu informacionu revoluciju i potencijal za rasprostranjenje pismenosti, a time i široke narodne kulture.

Do slične transformacije dolazi i u 20. veku: tehnologija digitalne informacije (tehnologija koja koristi elektroniku za transformisanje informacije u digitalni, binarni oblik, sastavljen od 0 i 1), predstavlja drugu informacionu revoluciju. Ova tehnologija proširuje a u nekim slučajevima i zamenjuje tehnologiju pisane reči. Tako se znanje i mudrost 5000 godina civilizacije, prikupljena u obliku više od 100 miliona jedinica (knjiga, časopisa, tehničkih izveštaja, novina, filmova, mapa, notnih i zvučnih zapisa) koje se čuvaju u Kongresnoj biblioteci SAD, mogu smestiti na nekoliko stotina optičkih (kompakt) diskova.

Informacione tehnologije. Današnje informacione tehnologije mogu se podeliti u četiri široke kategorije: **tehnologije senzora** (ili prikupljanja podataka i informacija, "ulazne" tehnologije), **komunikacione tehnologije**, **računarske tehnologije** i **tehnologije prikazivanja** (displej tehnologije, tehnologije "izlaza" ili izdavanja i prezentacije informacija).

Neki primeri **senzorne tehnologije** su digitalne video kamere, ledni senzori na krilima aviona, tastatura tarminala, miš, skeneri slike, čitači bar kôda, uređaji za glasovni ulaz, itd.

Komunikacione tehnologije uključuju komunikacione kablove, telefonske linije, internet – "mreža svih mreža", mreža koja povezuje stotine hiljada drugih računarskih mreža i skoro 3 milijarde računara tj. korisnika (podatak iz 2014. godine) u jedinstveni "cyberspace", koja se koristi u preko 100 zemalja. Internet prenosi digitalni tekst, grafiku, audio i video materijal preko geografskih i političkih granica i predstavlja bogato elektronsko tržište dobara i usluga. Tu su, zatim, lokalne mreže (LAN) koje omogućuju grupama ljudi da rade na zajedničkim poslovima, faks mašine, mobilna telefonija, modemi, telefonske mreže, kablovske mreže, i sl.

Računarska tehnologija – pre svega računarski sistemi za čuvanje i obradu informacija, koji se sastoje od tehničkog dela računara – hardvera (fizičke opreme) i programa ili instrukcija računaru šta da uradi – softvera. Računarski sistemi primaju informaciju od senzornih uređaja i komunikacionih uređaja, zatim je skladište i obrađuju, a rezultate obrade prosleđuju tehnologijama prikazivanja.

Tehnologije prikazivanja – uređaji i programi koji omogućuju da se obrađeni podaci prikažu korisniku u što prikladnijem obliku. Oni čine sumeđe (engl. interface) senzornih, računarskih i komunikacionih tehnologija prema čoveku – npr. ekrani terminala, štampači, ravni monitori – LCD (Color Liquid Crystal Display), HDTV (High Definition TV), UHDTV (Ultra High Definition TV), glasovni izlaz, i sl.

U središtu računarskih tehnologija je savremeni elektronski računar. Ono što ga čini univerzalnom mašinom jeste njegova sposobnost programiranja – druge mašine su namenjene specifičnom zadatku i samo taj zadatak mogu da izvršavaju – npr. CD plejer, automobil, mikser, štednjak, – njihov "program" je tvrdo realizovan u samoj mašini. Od 1940. godine računari imaju mogućnost izvršavanja različitih

zadataka prema različitim programima. – npr. za obradu teksta, za igranje igara, grafičko crtanje, i sl.

1.4 Istorija informacionih tehnologija i sistema

Istorija informacionih tehnologija i sistema se deli u četiri osnovna perioda, od kojih se svaki karakteriše osnovnom tehnologijom koja se koristi u rešavanju problema ulaza, obrade, izlaza i komunikacije. Ti periodi su: premehanički, mehanički, elektromehanički i elektronski.

Premehanička era je period od 3000.g.p.n.e do 1450.g.n.e. Osnovni problemi ovog perioda bili su kako predstaviti koncepte kao što su jezik i brojevi, i kako sačuvati informaciju i predstaviti je tako da ostane precizna, trajna i jednoznačna. Rešenja su bili sistemi pisanja i brojanja – azbuke i brojevni sistemi, uz pomoć trenutne tehnologije – glinenih ploča, papira, olovke, abakusa.

Prvi brojevni sistemi nalik onima koje koristimo danas nastali su tek između 100 i 200.g.n.e. – Hindu u Indiji, devetocifreni brojevni sistem. Tek oko 875.g.n.e. došlo se do koncepta nule. Indoarapski dekadni brojevni sistem – devet cifara i nula – Azijom su prenosili arapski trgovci, a u 13. veku ga je od arapskih matematičara u Evropu preneo i širio Leonardo Fibonači (Leonardo Fibonacci), italijanski matematičar. Konstruisana je "računaljka" – **abakus**. To je bio prvi "obrađivač" informacija sa mogućnošću privremenog predstavljanja brojeva i izvođenja računskih operacija pomoću žica i kuglica. Abakus je bio značajno računsko sredstvo kroz ceo srednji vek.

Mehanička are je period od 1450 do 1840. godine. Johan Guntenberg (Johannes Gutenberg, Majnc, Nemačka) oko 1450.g projektovao je i izgradio štamparsku presu. Ovo revolucionarno otkriće omogućilo je masovno korišćenje knjiga i prenošenje znanja (o brojevnom sistemu, abakusu, itd.)

Ranih 1600-tih, Vilijam Otred (William Oughtred), engleski sveštenik, konstruisao je "klizajući lenjir" (šiber) – spravu od drveta za množenje i deljenje; to je rani primer analognog računara, instrumenta koji meri umesto da broji ili računa.

Sredinom 17. veka Blez Paskal (Blaise Pascal), kasnije poznati francuski matematičar, izumeo je jednu od prvih mehaničkih računskih mašina, koja sabira i oduzima brojeve. Mašina – Paskalina, sastojala se od niza nazubljenih točkića. Gotfrid Vilhelm Lajbnic (Gottfried Wilhelm Leibniz), značajni nemački matematičar i filozof, unapredio je Pascaline 1672. godine komponentama za množenje i deljenje.

Do ranih 1800-tih godina nauke kao što su geografija, astronomija, matematika, bile su toliko razvijene da se javila potreba za novim i preciznijim računskim sredstvima. Engleski matematičar Čarls Bebidž (Charles Babbage), projektovao je mašinu koja računa i štampa rezultate tih računanja. Godine 1820. napravio je mali

model te mašine i nazvao je Diferencijska mašina (Difference Engine), po metodi tzv. konačnih razlika za rešavanje diferencijalnih jednačina. Mašina se sastojala od zupčanika i poluga i bila je projektovana da računa i izdaje dijagrame polinomijalnih funkcija. Mašina je u potpunosti realizovana tek 1991. godine (i izložena u Londonskom muzeju) sredstvima koja su bila na raspolaganju u Bebidžovo vreme.

1830-tih Bebidž je razmišljao i o "programabilnom računaru", tzv. analitičkoj mašini čiji su neki delovi projekta veoma slični današnjim računarima (deo za ulazne podatke i smeštanje rezultata, i deo za obradu podataka). Nameravao je da koristi bušene kartice za usmeravanje aktivnosti mašine. Projekat analitičke mašine čini Bebidža tvorcem ideje programabilnog računara.

Bebidžu je pomagala ćerka lorda Bajrona – Augusta Ada Bajron, neobično obrazovana žena tog vremena. Ada je pomagala Bebidžu oko instrukcija na bušenim karticama, zato se naziva "prvom ženom programerom". Ona je i analizirala i pisala o njegovim dostignućima, pa je zahvaljujući njenim rukopisima i prepisci sa Bebidžom i otkriveno njegovo delo 1950.g. (do kada je ostalo skriveno od javnosti).

Mada se Bebidžova ideja analitičke mašine smatra osnovom prvogramabilnog računara, njoj je prethodila "programabilna", mada ne i računska mašina, poznata kao Žakarov razboj. To je bio razboj koji je 1801. godine konstruisao Žozef Mari Žakar (Joseph Marie Jacquard), tkač i pronalazač iz Liona, i koji je za tkanje šara koristio bušene kartice.

Elektromehanička era je period od 1840 do 1940. godine i karakteriše se kombinovanom upotrebom mehaničkih i elektronskih komponenti.

Elektromehaničko računanje. Tehnologije zasnovane na elektricitetu i mehaničkom računanju kombinovane su prvi put 1880.g. za potrebe popisa stanovništva. Herman Holerit (Herman Hollerith, Vašington, SAD) je osnovao kompaniju za proizvodnju i prodaju mašina za čitanje i obradu bušenih kartica koje su nosile podatke, koja je kasnije prerasla u International Business Machine Corporation (IBM).

U Nemačkoj 1941.g. Konrad Cuze (Konrad Zuse) je izgradio elektromehanički delimično programabilni računar Z3 koji je uključivao 2000 elektromagnetnih prekidača.

Početkom 40-tih godina 20. veka, Hauard Ejkin (Howard Aiken), doktorant na Harvardu (SAD), kombinovao je Holeritove mašine i Bebidžovu ideju o programabilnom računaru, izgradivši mašinu poznatu pod imenom Automatic Sequence Controlled Calculator, ASCC, ili – Mark I. Mašina je imala papirnu traku sa programom na njoj, ulazne podatke na bušenim karticama, brojače za brojeve i elektromehaničke releje za smeštanje rezultata. Proradila je 1944.g. ali je već tada bila zastarela zbog korišćenja elektromehaničkih komponenti. To je bio kraj elektromehaničke ere.

Elektronska era je period od 1940.g. do danas. Elektronske vakuum cevi (lampe) mogle su da zamene elektromehaničke releje – prekidače. Jedna elektronska cev može da registruje jednu binarnu cifru (0 ili 1) – protiče ili ne protiče struja.

Razvijaju se u SAD, Nemačkoj i Velikoj Britaniji. Prvi elektronski računar koji je koristio binarni brojevni sistem bio je ABC računar (Atanasoff-Berry Computer; John Vincent Atanasoff, Clifford Berry, SAD) specijalne namene izgrađen 1939. godine, sa 300 cevi. U Engleskoj, 1943. A. Tjuring pomogao je u izgradnji elektronskog računara Colosus, za dešifrovanje neprijateljskih poruka; ovaj računar je imao specijalnu namenu i bio je samo delimično programabilan.

Godine 1943. američka vojska je finansirala izgradnju računara ENIAC – The Electronic Numerical Integrator and Calculator. Bio je to računar sa elektronskim cevima koji je završen 1946. U milisekundama je množio i delio, i računao trajektoriju za 20 sekundi, ali nije bio programabilan. Ekert i Močli (Eckert, Mauchly), idejni tvorci ENIAC-a, imali su ideju i za izgradnju programabilnog računara opšte namene – EDVAC – Electronic Discreet Variable Computer. Jedan od saradnika na tom projektu, Džon fon Nojman (John von Neumann), američki matematičar, učesnik Menhetn projekta atomske bombe, u to vreme, 1945.godine, sintetisao je i objavio ideje tvoraca ENIAC-a i on se smatra za tvorca koncepta programabilnog računara. Godine 1949, na osnovama EDVAC-a kompletiran je računar EDSAC (Electronic Delay Storage Automatic Calculator). Konstruktor je bio Britanac Moris Vilkis (Maurice Wilkes), i to je prvi realizovni programablini elektronski računar. Ideju za prvi komercijalni računar Ekert i Močli prodali su Remingtonu (bio je to UNIVAC – Universal Automatic Computer), ali su ih u realizaciji opet pretekli za nekoliko meseci Britanci – računarom LEO (Lyons Electronic Office), modeliranim prema EDSAC-u. Računar UNIVAC je proizveden 1951. godine i prodat u 46 primeraka, što je za ono vreme bio veoma veliki broj.

1.4.1 Generacije savremenih elektronskih računara

Savremena istorija (poslednjih 60 godina) informacione tehnologije tradicionalno se deli u 4 (ili 5) faza – računarskih generacija. Svaka se odlikuje tehnologijom koja se koristi za kreiranje osnovnih logičkih elemenata (elektronskih komponenti za skladištenje i obradu informacija).

Prva generacija (1939 – 1958) koristi elektronske cevi kao osnovne logičke elemente. Cevi su nedovoljno pouzdane, troše mnogo struje, mnogo se zagrevaju, velike su i mnogobrojne, pa su računarni ogromni. Kao ulaznu tehnologiju koriste bušene kartice, rotirajuće magnetne doboše kao unutrašnju memoriju, a mašinski ili simbolički jezik za programiranje. Korišćeni su u nauci, inženjerstvu i obimnim komercijalnim aplikacija kao što su plate ili naplata.

Druga generacija (1959–1963) karakteriše se kristalnom mineralnom materijom – poluprovodnicima kao materijalom za **tranzistore**, uređaje – koje je kasnih 40-tih godina proizvodila firma AT&T BEll Labs (SAD). U početku je korišćen skup element germanijum, a tek od 1954. godine za tranzistore se koristi široko prisutan mineral istih svojstava – silicijum (sastojak peska), i time znatno pojeftinjuje proizvodnja tranzistora. Kao tehnologija, tranzistori su sitniji, zauzimaju

mnogo manje prostora, pouzdaniji su, troše manje struje, manje se zagrevaju, brži su od elektronskih cevi i moćniji. Kao i cevi, morali su da se zavaruju i uklapaju zajedno tako da formiraju elektronsko kolo.

Kao unutrašnja memorija koristi se magnetno jezgro (mali magneti oblika torusa) , magnetni diskovi i trake koriste se kao spoljne memorije, a za programiranje računara koriste se programski jezici visokog nivoa, FORTRAN (FORmula TRANslator), COBOL (COmmon Business Oriented Language). Prvi programski jezik ovog nivoa, FORTRAN, konstruisao je Džon Bekus (John Backus, SAD), 1957. godine. On je opovrgao tezu da se efikasni programi mogu pisati samo na mašinskom jeziku. To je jezik blizak matematičkoj notaciji, koji se efikasno prevodi na mašinski jezik. U ovo vreme nastaje i prvi funkcijski jezik, LISP (LISt Processing, Džon Mekarti – John McCarthy, SAD, 1958), bitno različit od imperativnih (proceduralnih) jezika tipa FORTRAN. Korišćenje računara je u velikom porastu, javljaju se novi korisnici i programski paketi. Programi postaju čitljiviji i prenosiviji.

Treća generacija (1964 – sredina 70-tih) Sredinom 60-tih, pojedinačni tranzistori su zamenjenji integrisanim kolima – hiljade malih tranzistora na malom silicijumskom čipu predstavljalo je revolucionarnu promenu u računarskoj industriji. Ostvarena je ušteda prostora, pouzdanost i brzina u odnosu na tranzistore. Memorija je takođe bila na silicijumskom čipu. Povećani kapacitet memorije i moć obrade omogućili su razvoj operativnih sistema – specijalnih programa koji pomažu raznim komponentama računara da rade zajedno da bi obradili informaciju. Javlja se veliki broj programskih jezika novih karakteristika (npr. BASIC). U evoluciji jezika sa podrškom strukturnim iskazima i strukturnim podacima 60.tih godina javlja se Algol 60 (Niklaus Virt – Niclaus Wirth, Švajcarska), koji kroz jezike Algol W – 1966, Pascal – 1971, Modula 2 – 1983, Oberon – 1988, evoluira do objektnog jezika. Istovremeno, u liniji jezika kojoj pripada programski jezik C, 1966. godine javlja se CPL – Combined Programming Language (Kristofer Streči – Christopher Strachey, Britanija; jezik nikada nije u potpunosti implementiran), 1969. godine BCPL (Basic CPL) - alat za pisanje kompilatora, 1972. godine C programski jezik (Denis Riči – Dennis Ritchie, SAD) kao implementacioni jezik za softver vezan za operativni sistem UNIX (1973. godine je i sam UNIX prepisan u C-u), i najzad, opet objektni jezik, C++ (Bjarn Stroustrup – Bjarne Stroustrup, Danska, 1986), jezik koji omogućuje definisanje novih tipova korišćenjem klasa "pozajmljenih" od jezika Simula 67. Pored proceduralnih jezika, u ovo vreme javljaju se i prvi neproceduralni (logički) jezici – Prolog 1972. godine (Robert Kovalski – Robert Antony Kowalski, Vel. Britanija).

Na tržištu mejnfrejm ("velikih") računara dominirala je američka kompanija IBM. Računarska tehnologija je omogučila i proizvodnju miniračunara. Na ovom tržištu dominirala je američka kompanija DEC (Digital Equipment Corporation).

Četvrta generacija (sredina 70-tih – danas) Četvrtu generaciju računara doneli su mikroprocesori, koji su omogućili ugradnju hiljada integrisanih kola na jednom silicijumskomčipu. Ova integracija vrlo velikih razmera (VLSI – Very Large

Scale Integrated Circuits) smešta milione tranzistora na jednom čipu, uključujući memoriju, logiku i kontrolna kola. Personalni računari, kao Apple Macintosh i IBM PC, postaju popularni za poslovnu i ličnu upotrebu. Javljaju se tzv. 4GL – jezici i programska okruženja četvrte generacije , dBASE, Lotus 1-2-3, WordPerfect, a kasnije i sofisticirani jezici baza podataka, programski sistemi za upravljanje tekstom, slikom, zvukom i sl, za korisnike bez ikakvog tehničkog znanja. Savremene tendencije ogledaju se u korišćenju prenosivih računara, minijaturizaciji i multifunkcionalnosti. Razvijaju se uređaji koji imaju, pored ostalih, i funkcije računara, i integrišu informacione tehnologije – mobilna telefonija, tableti i sl.

Pravo teleizračunavanje postiže se kroz Internet realizovan (u početku) kroz obične telefonske linije, a danas kroz brze ADSL, IDSN ili kablovske veze.

Peta generacija (1982 – kraj 80-tih) U literaturi se pominje i peta generacija računarskih sistema kao projekat japanske vlade započet 1982. godine i orijentisan na korišćenje masovno paralelne obrade (istovremeno korišćenje velikog broja procesora). Cilj projekta je bio da ostvari platformu za budući razvoj veštačke inteligencije. Dok su prethodne generacije bile usmerene na porast broja logičkih elemenata na jednom procesoru, peta generacija bila je okrenuta povećanju broja samih procesora da bi se poboljšale performanse sistema. Ovaj projekat se smatra neuspešnim, a prema nekim mišljenjima – preuranjenim.

2

Računarski sistemi

2.1 Fon Nojmanova (von Neumann) mašina

Struktura savremenog računara veoma je slična strukturi tzv. fon Nojmanove mašine (izgrađene kasnih 1940-tih godina u Institutu za napredne studije, IAS, u Prinstonu, gde je Džon fon Nojman bio profesor matematike), pa se za savremene elektronske računare kaže da u osnovi imaju fon Nojmanovu arhitekturu. Arhitekturu ove mašine fon Nojman je opisao 1945. godine u izveštaju o računaru EDVAC. Opišimo je ukratko.

Elementarni fizički objekat fon Nojmanove mašine, koji može da bude u 2 diskretna stanja – protiče struja/ne protiče struja, tj. da "registruje" binarnu cifru 0 ili 1, naziva se **ćelija** (ćelija je elementarni fizički objekat i savremenih elektronskih računara – elektronska cev, tanzistor). U njoj se može prikazati jedna binarna cifra tj. jedan **bit** informacije (engl. "binary digit" – binarna cifra). Ćelije se u fon Nojmanovoj mašini organizuju u nizove fiksne dužine koji se zovu **registri**.

Fon Nojmanova mašina se sastoji od procesora (engl. Central Processing Unit – CPU) i memorije (slika 2.1). CPU se sastoji od kontrolne jedinice i aritmetičke jedinice. Aritmetička jedinica sadrži i dva specijalna registra, akumulator i registar podataka R. Kontrolna jedinica sadrži komponente za izvršavanje instrukcija i niz registara neophodnih za pripremu instrukcije za izvršenje. Memorija sadrži instrukcije (program) i podatke. Memorija se sastoji od 4096 registara od kojih svaki ima svoju **adresu** (mesto, lokaciju) – broj od 1 do 4096, a svaki registar ima po 40 bita. Sadržaj svakog registra može da se interpretira kao jedan ceo broj u binarnom obliku, ili kao dve (20-bitne) instrukcije. Program se sastoji od niza binarnih instrukcija (instrukcija zapisanih binarnom azbukom), tj. program je **na mašinskom jeziku**.

Mašina ima i neki (neprecizirani) metod smeštanja podataka i instrukcija u memoriju i prikazivanja rezultata iz memorije spoljašnjem svetu. Za to je zadužen ulazno/izlazni sistem (I/O).

Fon Nojmanova mašina ima sledeće karakteristike:

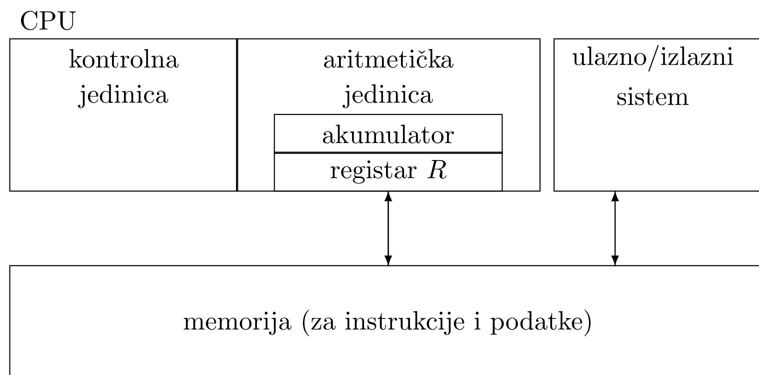

Figure 2.1: Struktura fon Nojmanove mašine

- Podaci. Celi brojevi su jedini oblik podataka. Oni su predstavljani u binarnom obliku i to tako što se dekadni broj $d_n d_{n-1} \dots d_0$ ($d_0, d_1, \dots d_n \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$) predstavlja binarnim brojem $b_m b_{m-1} \dots b_0$ ($b_0, b_1, \dots b_m \in \{0, 1\}$), tako da je $d_n \times 10^n + d_{n-1} \times 10^{n-1} + \cdots + d_1 \times 10^1 + d_0 \times 10^0 = b_m \times 2^m + b_{m-1} \times 2^{m-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$ (precizan algoritam prevođenja dekadnog u binarni broj biće predstavljen u delu 11.5.1 – Razvoj algoritama sa nizovima – Konverzija broja iz dekadnog u binarni oblik).

- Aritmetičke operacije. Mašina može da sabira, oduzima, množi, deli i računa apsolutnu vrednost broja. Rezultat sabiranja ili oduzimanja je smeštan u registar koji se zove akumulator. Rezultat množenja ili deljenja smeštan je u par registara (akumulator, R). Svaka instrukcija ima deo koji određuje vrstu operacije (8 bita) i deo koji određuje adresu operanda u memoriji (12 bita). Na primer, instrukcija (00111100100000001011) ima značenje "dodati (na vrednost u akumulatoru) vrednost sa lokacije (adrese) 968 (niz od nižih – desnih osam bitova, 00001011, jeste kôd operacije "dodati", a viših – levih – dvanaest binarnih cifara predstavljaju celobrojnu adresu operanda). Ova mašinska instrukcija mogla bi se zapisati i tzv. **simboličkim jezikom**, npr. u obliku ADD J, gde je ADD – naziv operacije dodavanja (i odgovara binarnom kôdu 00001011), a J – simbolička adresa lokacije 968 (fon Nojmanova mašina nema mogućnost zapisa instrukcija na simboličkom jeziku).

- Dodela memorijskoj lokaciji/procesorskim registrima. Memorijskoj lokaciji (memorijskom registru) se može dodeliti vrednost iz akumulatora. Akumulatoru ili registru podataka R može se dodeliti vrednost iz specifičnog memorijskog registra.

- Tok upravljanja. Tok upravljanja je jedna instrukcija za drugom, osim u slučaju "goto" instrukcije koja ukazuje na memorijsku lokaciju gde treba naći instrukciju za izvršenje.

2.2 Struktura savremenog računarskog sistema

Savremeni računarski sistem sastoji se od dve osnovne komponente: **tehničkog sistema računara** (u osnovi fon Nojmanove arhitekture) i **programskog sistema računara**.

Objasnimo strukturu tehničkog i programskog sistema.

2.2.1 Tehnički sistem računara

Tehnički sistem računara (hardver, engl. hardware) čine svi uređaji računarskog sistema. Tehnički sistem, kao osnovne komponente koje obezbeđuju njegovu funkcionalnost, uključuje komponentu za obradu podataka (procesor), i skladište (memoriju) za pamćenje programa i podataka. Ove dve komponente zajedno nazivaju se **računar**. Osim ovih, tehnički sistem uključuje i komponente ulaznih i izlaznih informacionih tehnologija, prilagođene ulazu podataka u računar i izdavanju rezultata obrade, kao i komponente komunikacione tehnologije koje omogućuju komunikaciju među pojedinim komponentama.

Pored računarske (operativne) memorije, čiji se sadržaj gubi prestankom električnog napajanja, praktična potreba trajnog čuvanja programa i podataka nameće i postojanje "spoljašnjeg" skladišta – spoljašnje memorije, čiji sadržaj ostaje sačuvan trajno.

Dakle, tehnički sistem računara globalno se može podeliti na **računar** i **periferne uređaje**. Računar se sastoji od **procesora** i **operativne (unutrašnje) memorije** (u današnjim računarima smešteni su na matičnoj ploči – engl. motherboard, zajedno sa grafičkom karticom, video karticom, mrežnim kontrolerom i sl), dok periferni uređaji uključuju **ulazno/izlazne uređaje** i **spoljašnju memoriju**. Sve komponente računarskog sistema međusobno su povezani komunikacionim kanalima, tzv. **magistralama** (engl. bus).

Procesor Procesor (engl. CPU – Central Processing Unit) je osnovna komponenta računarskog sistema jer se izvršavanje programa odvija u procesoru. Procesor ima dve značajne funkcije u računaru: da obrađuje podatke i da upravlja radom ostalih delova računarskog sistema – ulaznih, komunikacionih, izlaznih uređaja, memorije. Procesor je (od 1970. godine) realizovan kao mikroprocesor, tj. izgrađen na silicijumskom poluprovodničkom čipu. Broj tranzistora u jednom elektronskom kolu na mikroprocesorskom čipu prati tzv. Murov zakon, prema kome se broj tranzistora (zbog minijaturizacije) udvostručuje svake dve godine, približno. Taj broj dostiže 4.3 milijarde kod komercijalnih procesora 2014. godine.

Procesor ima specifične elemente (delove) koji izvršavaju pojedine mašinske instrukcije, i konačni (obično neveliki) skup registara opšte namene (koji mogu da prime podatke), i specijalne namene, kao što su akumulator (za privremeno čuvanje rezultata), brojač instrukcija (za memorijsku adresu instrukcije koja je na redu za izvršavanje), registar adresa (za memorijske adrese podataka koji se obrađuju tekućom instrukcijom), registar instrukcija (za tekuću instrukciju), itd.

Skup instrukcija koje procesor može da izvrši, zajedno sa skupom registara, definiše **arhitekturu** procesora. Arhitektura se razlikuje od procesora do procesora.

Procesor se sastoji od obradne jedinice tj. jedinice za aritmetičku i logičku obradu podataka (pa po tome i "aritmetičko-logičke jedinice", ALJ – engl. ALU – Arithmetic-Logic Unit), i kontrolne (ili upravljačke) jedinice, KJ (engl. Control Unit – CU) koja određuje koju komponentu ALJ treba aktivirati u kom trenutku da bi se izvršila tekuća operacije, kao i koja je instrukcija sledeća za izvršenje.

Aritmetičko-logička jedinica je deo procesora u kome se izvršavaju elementarne operacije – aritmetičke, npr. sabiranje, oduzimanje, celobrojno sabiranje, celobrojno oduzimanje i logičke, npr. poređenje dva broja. Ova jedinica prima ulazne električne signale ($\{0, 1\}$) koji predstavljaju ulazne podatke, i elektronski ih transformiše u električne signale koji predstavljaju rezultate. Ulazni signali dolaze u aritmetičko-logičku jedinicu (npr. u njene registre opšte namene) iz nekih polja operativne memorije (čija je adresa sastavni deo instrukcije), a izlazni signal (iz akumulatora ili registra opšte namene šalje se u odredišno polje memorije (koje se takođe navodi u instrukciji).

Kontrolna jedinica upravlja signalima i odgovorna je za to koje operacije i kada izvršiti za vreme izvršavanja programa. Ova jedinica sadrži komponentu koja dekodira operaciju tekuće intsrukcije. Ova jedinica izvršava i posebne instrukcije kao što su uslovni i bezuslovni skok i zaustavljanje programa.

Brzina procesora izražena je u MIPS (miliona instrukcija u sekundi) ili FLOPS (operacija u pokretnom zarezu u sekundi). Najpoznatije mikroprocesorske arhitekture imaju mikroprocesori firmi Intel (16-bitna familija MCS-86 – 80186, 80286, 32-bitna 80386, 80486, Pentium, Celeron, Intel Core, 64-bitni Intel Core 2, Core i3, i5, i7, Xeon, Intel Atom), Motorola (serija 68000), IBM (RISC – "Reduced Instruction Set Computer" arhitektura) itd. Noviji centralni procesori su višejezgarni (multicore), tj. imaju veći broj jezgara – pojedinačnih procesora na jednom mikroprocesorskom čipu.

Operativna memorija Operativna ili unutrašnja memorija je veoma tesno vezana sa procesorom. I ona se gradi na poluprovodničkim silicijumskim čipovima sa milijardama tranzistora. Ova memorija služi za čuvanje informacija neposredno potrebnih u procesu obrade, a to su programi operativnog sistema, programi koji se upravo izvršavaju, i podaci potrebni tim programima. Memorija se sastoji iz niza bitova organizovanih u nizove dužine 8 koji se nazivaju **bajtovima**. Bajtovima su dodeljene **adrese**, (brojevi od 1 do ukupnog broja bajtova). Bajtovi se dalje grupišu u nizove fiksne dužine (obično 2, 4 ili 8 bajtova), koji se nazivaju **registrima**, sa adresama koje predstavljaju najmanje adrese bajta koji učestvuju u tom registru. Kada se novi sadržaj upiše na memorijsku lokaciju (adresu), prethodni sadržaj sa te adrese se izgubi. Veličina (kapacitet) memorije se meri u bajtovima (B), odnosno krupnijim jedinicama – KB (kilobajt = 1024 = 2^{10} bajta), MB (megabajt = 1024KB), GB (gigabajt = 1024MB), TB (terabajt = 1024GB).

Unutar operativne memorije postoje različite lokacije (adrese) za čuvanje različitih vrsta podataka i programa – deo memorije za ulazne podatke (koji dolaze

sa ulaznih uređaja), deo memorije za izlazne podatke (one koji se šalju na izlazne uređaje), deo za smeštanje programa, radni prostor za smeštanje međurezultata, za različite vrste programskih podataka, itd. Ova raspodela unutrašnje memorije nije statična i zavisi od programa koji se izvršavaju.

Računar obično poseduje nekoliko različitih tipova operativne memorije, koji se koriste za različite svrhe.

RAM (Random Access Memory) su memorijski čipovi čijim se proizvoljnim lokacijama može pristupiti direktno, radi upisa ili čitanja podataka ili instrukcija. Sadžaj RAM memorije se gubi kada prestane električno napajanje.

Posebna vrsta RAM-a je tzv. keš memorija kojoj mikroprocesor može da pristupi brže nego običnoj RAM meoriji. Ova memorija je mala i u njoj se čuvaju podaci koji su nedavno (u najbližoj prošlosti) korišćeni iz RAM memorije. Mikroprocesor traži potrebne podatke prvo u keš memoriji.

ROM (Read Only Memory) čipovi koriste se za trajno čuvanje nekih instrukcija. Te instrukcije su obično deo operativnog sistema koji proverava hardver kadgod se računar uključi u struju. Sadržaj ROM čipa se ne gubi kada se računar isključi, i ne može se menjati.

PROM (Programmable ROM) čip je sličan ROM-u, osim što je na početku prazan, a pomoću specijalnog uređaja nabavljač računara ga popunjava željenim instrukcijama. Od tog trenutka PROM se ponaša isto kao ROM.

EPROM (Erasable PROM) čip je varijanta PROM čipa čiji se sadržaj može i menjati pomoću ultravioletnih zraka i specijalne opreme.

EEPROM (Electrically Erasable Programmable Read-Only Memory) je ROM čiji sadržaj korisnik može da briše i ponovo programira (upisuje) elektronskim putem. Specijalna vrsta EEPROM memorije je fleš memorija koja se koristi za USB fleš. Ovaj oblik memorije se danas najčešće koristi kao spoljašnja memorija.

Ulazno/izlazni uređaji Ovi uređaji obezbeđuju unošenje i izdavanje podataka u računar tj. iz računara, i predstavljaju sredstvo komunikacije računara sa spoljašnjom sredinom.

Primer ulaznih uređaja su tastatura terminala, miš, svetlosno pero, mikrofon, džojstik, uređaji za prepoznavanje govora, ekran osetljiv na dodir (touchscreen), jastuče osetljivo na dodir (touchpad) (svi oni uključuju čoveka u proces ulaza podataka), ili skeneri, optički čitači, senzori, veb kamere, kojima se eliminiše čovek iz procesa unošenja podataka. Podaci koji se unose preko ovih ulaznih uređaja, bilo da su to brojevi, tekst, slika, zvuk, smeštaju se, transformisani u binarni oblik, u deo operativne memorije namenjen ulazu podataka. Na primer, skener, uz specifični softver, skenira sliku gradeći u memoriji bitmapu – mapu ćelija, od kojih se svakoj pridružuje binarni kod (broj) koji odgovara njenoj osvetljenosti i boji. Tek iz memorije se svi ti podaci mogu koristiti ili za obradu ili za predstavljanje, sada opet transformisani u za čoveka pogodni oblik (dekadni broj, slova, sliku, zvuk) na izlazne uređaje.

Primeri izlaznih uređaja su ekran terminala, štampač, crtač, uređaj za izlaz na mikrofilm, uređaji za glasovni izlaz, itd. Da bi se ekran mogao koristiti za izdavanje

znakova, brojeva, ali i slika i crteža, neophodna je i odgovarajuća grafička kartica koja obezbeđuje odgovarajuću rezoluciju (broj osvetljenih tačaka na ekranu).

Spoljašnja memorija Potreba za trajnim čuvanjem velikih količina informacija – podataka i programa dovela je do koncepta "spoljašnjeg" skladišta – spoljašnje memorije, čiji sadržaj ostaje sačuvan trajno. Spoljašnja memorija se koristi uz pomoć uređaja koji omogućuje upravljanje njom (tzv. kontroleri spoljašnje memorije, npr. kontroler diska), i programa koji omogućuju čitanje i pisanje iz (u) spoljašnje memorije (npr. pogonaš diska – engl. disk driver).

Podaci i programi koji se čuvaju u spoljašnjoj memoriji po potrebi se prenose u operativnu memoriju i koriste u procesu obrade. Vreme pristupa spoljašnjoj memoriji je mnogo veće od vremena pristupa unutrašnjoj, ali je zato kapacitet neuporedivo veći.

Kao prvi oblici medija spoljašnje memorije korišćeni su papirne kartice i papirne trake, koje nemaju mnogo sličnosti sa današnjim medijima spoljašnje memorije. Kao mediji spoljašnje memorije koriste se danas najčešće magnetni (unutrašnji ili spoljašnji) disk, optički (kompakt) disk (CD-ROM, koji se može samo čitati), ili čitati i pisati (proizvoljan broj puta) – CD-R (CD-RW), DVD (Digital Video (ili Versatile – svestrani) Disc), fleš disk, memorijske kartice. Optički diskovi se koriste za distribuciju velikih programskih paketa (CD do 900MB, DVD do 10GB).

Sadržaj savremenih medija spoljašnje memorije je skup tzv. spoljašnjih datoteka organizovanih na razne načine – npr. u hijerarhiju kataloga. Spoljašnja datoteka je niz znakova (bajtova) kome je pridruženo ime, i koji se može interpretirati na razne načine – npr. kao tekst, kao program na programskom jeziku, kao program preveden na mašinski jezik, kao slika, crtež, zvuk, itd. Na sadržaj datoteke tj. način na koji ga treba interpretirati, ukazuje ekstenzija uz ime datoteke (npr. txt, pas, c, out, obj, exe, bmp, itd). (Pojam spoljašnje datoteke se bitno razlikuje od pojma datoteke kao strukture podataka u nekim programskim jezicima).

Magistrale Magistrale se razlikuju među sobom po tome šta se preko njih prenosi i po svojoj "širini", tj. broju fizičkih linija odnosno broju bitova koji se istovremeno mogu prenositi magistralom. Tako se razlikuju **magistrala adresa** ili **memorijska magistrala** preko koje se prenose memorijske adrese podataka ili instrukcija, **magistrala podataka** preko koje se prenose podaci iz memorije ili u memoriju, i **kontrolna magistrala** preko koje se prenose signali za čitanje ili upis, iz memorije ili sa ulaznih uređaja odnosno u memoriju ili na izlazne uređaje. Što se tiče širine, ona se može razlikovati među magistralama adresa, podataka i kontrole i u jednom računarskom sistemu. Ranije su magistrale podataka obično bile 8-bitne, adresne magistrale 20-bitne. Danas su magistrale 32-bitne ili 64-bitne. Tako, kada se kaže da računar ima 64-bitni procesor, misli se na 64-bitnu magistralu podataka koja prenosi podatke od procesora do unutrašnje memorije.

Osnovna karakteristika računarskog sistema je **konfiguracija**, tj. sastav tehničkog sistema: koji računar (koji procesor, koju i koliku operativnu memoriju), i koje periferne uređaje sadrži tehnički sistem. Jedan isti računar može se koristiti

sa raznim perifernim uređajima i tako učestvovati u različitim konfiguracijama, u zavisnosti od namene računarskog sistema.

Jedan primer personalne (kućne) konfiguracije je računar sa Intel Core i5 procesorom (sa dva ili četiri jezgra) i 4Gb RAM, diskom od 250Gb, sa tastaturom, mišem, TFT monitorom (ravni ekran), laserskim štampačem, CD/DVD jedinicom i USB (Universal Serial Bus) portovima (priključcima).

2.2.2 Programski sistem računara

Postoje dva tipa programskih sistema (softvera): **sistemski softver** i **aplikativni softver**. Sistemski softver ili programski sistem računara je skup programa koji kontroliše i koordinira rad različitih tipova opreme u računarskom sistemu, i omogućuje korisniku efikasno i udobno korišćenje raspoložive opreme. Aplikativni softver omogućuje primenu računara za rešavanje specifičnih problema ili izvršavanje specifičnih zadataka.

Programski sistem računara je nadgradnja tehničkog sistema, i u prvoj generaciji savremenih elektronskih računara nije ni postojao. Korisnik je neposredno preko ulazno/izlaznih uređaja komunicirao sa računarom da bi izvršio svoje programe (napisane na mašinskom jeziku računara). Podaci su takođe bili u sličnom, binarnom zapisu, a programiranje i korišćenje računara složeno.

Savremene generacije računara imaju bogat repertoar sistemskog softvera.

Osnovne komponente programskog sistema računara mogu da se grupišu u **operativne sisteme**, **jezičke procesore** i **uslužne programe** (ili pomoćne programe).

Operativni sistemi Operativni sistem je osnovna komponenta programskog sistema računara. To je skup programa koji ima tri osnovne funkcije: dodeljivanje sistemskih resursa (npr. procesora, memorije), raspoređivanje različitih operacija koje treba da se obave i nadgledanje sistemskih aktivnosti. Deo operativnog sistema je uvek u operativnoj memoriji kada je računar uključen. On prima signale sa uređaja i naredbe od korisnika i omogućuje njihovu obradu i izvršavanje.

Operatvini sistem omogućuje različite politike raspoređivanja tj. dodele računarskih resursa (procesora i memorije) različitim programima (procesima). **Serijsko** izvršavanje programa ili **paketna obrada** označava izvršavanje jednog programa od početka do kraja, pre nego što drugi program može da započne svoje izvršavanje. **Multiprogramiranje** dopušta većem broju korisnika da istovremeno izvršavaju svoje programe, tj. da njihovi programi istovremeno budu u operativnoj memoriji. Jednom od programa se dodeljuje procesor i taj program se izvršava dok ne dođe na red izvršavanje neke njegove ulazno/izlazne operacije. Takva operacija zahteva korišćenje ulazno/izlaznih uređaja a ne procesora, pa se tada procesor dodeljuje drugom programu koji počinje (ili nastavlja) da se izvršava. Ovakva procedura se primenjuje na programe svih korisnika. Ovaj oblik multiprogramiranja bio je karakterističan za rane računare kod kojih je izvršavanje I/O operacija bilo dugo. Alternativni metod za istovremeno izvršavanje većeg broja programa je **rad u razdeljenom vremenu** (engl. time sharing). Svakom programu dodeljuje se

mali vremenski interval (engl. time slot), npr 20msec (milisekundi), u kome može da koristi procesor. Procesor se dodeljuje kružno, tako da svaki program u kratkim vremenskim razmacima dobija procesor na korišćenje, čime se svakom korisniku stvara utisak da sâm koristi procesor. Rad u razdeljenom vremenu se kombinuje sa klasičnim multiprogramiranjem, pri čemu je kriterijum za oduzimanje procesora programu ili potrošeni vremenski interval ili zahtev za ulazno/izlaznom operacijom. Rad u razdeljenom vremenu bio je značajna tehnološka novina u korišćenju računara 70. godina prošlog veka. **Multitasking** omogućuje istovremeno (paralelno) izvršavnje većeg broja procesa - bilo da pripadaju jednom ili većem broju korisnika, bez obzira na politiku dodele procesora i memorije, i u tom smislu obuhvata prethodne politike. Jedan od razloga za multitasking uopšte jesu sistemi za rad u realnom vremenu, kod kojih neki spoljašnji događaji zahtevaju trenutnu dodelu procesora programu koji kontroliše odgovarajući događaj. Savremeni operativni sistemi uglavnom koriste neki oblik rada u razdeljenom vremenu, uz primenu većeg broja nivoa prioriteta. **Multiprocesiranje** je režim rada računara koji koristi veći broj procesora za izvršavanje instrukcija. Ovaj režim omogućuje paralelno (u punom smislu) izvršavanje instrukcija jednog programa ili većeg broja programa.

Najpoznatiji operativni sistemi za personalne i prenosne (laptop) računare su Microsoft Windows (Vista, 2000, XP, 7, 8), IBM OS2, za IBM velike računare (engl. mainframe) z/OS. Najpoznatiji operativni sistem koji se primenjuje na celoj liniji računara je UNIX, i njegova javno dostupna verzija Linux. Primeri popularnih modernih operativnih sistema poteklih od UNIX-a su Android, BSD, iOS, Linux, Mac OS X.

Uslužni programi Druga važna komponenta programskog sistama su **uslužni (pomoćni) programi** (engl. utilities). To su programi koji pomažu, uglavnom na niskom nivou, u analiziranju, optimizaciji ili održavanju računara. Obično se nazivaju alatima (engl. utility, tool). To su, na primer, antivirus programi, programi za arhiviranje, pravljenje rezervnih kopija, čišćenje i komprimovanje diska, "štedišа ekrana" (screensaver) i sl. Ovde pripadaju i programi za pokretanje i korišćenje uređaja kao što su DVD, miš, USB flash i sl, tzv "pogonaši" (engl. drivers).

Jezički procesori Treća važna komponenta programskog sistema računara su **jezički procesori**. Da bi program mogao da se izvrši u računaru, potrebno je da bude zapisan na mašinskom jeziku. Prvi računari su imali samo mogućnost programiranja na mašinskom jeziku. Kako je mašinski jezik krajnje nepodesan za čoveka – programera, izgrađivani su pogodniji programski jezici, prvo simbolički a zatim viši programski jezici. Da bi programi napisani na ne-mašinskom programskom jeziku mogli da se izvršavaju na računaru, neophodno je da se izgrade programi koji će ih prevesti na mašinski jezik. Ti programi nazivaju se jezički procesori. Za prevođenje simboličkog jezika na mašinski jezik grade se **asembleri** a za prevođenje viših programskih jezika grade se **kompilatori** (ili prevodioci).

Da bi se prevedeni program mogao izvršiti, neophodno ga je povezati sa drugim prevedenim modulima (npr. programima za računanje elementarnih funkcija iz

sistemske biblioteke), i tako povezan mašinski program smestiti u deo operativne memorije predviđen za smeštanje izvršnog programa. Ove poslove obavljaju programi za povezivanje i punjenje (engl. linker, loader).

Jednom prevedeni program može se više puta izvršavati, pri čemu prevođenje i izvršavanje programa nije vremenski povezano. Program može da se izvršava korišćenjem programa za otkrivanje ("trebljenje") grešaka – debagera (engl. debugger).

Posebna vrsta jezičkih procesora su **interpreteri**, koji prevode, liniju po liniju, program sa višeg programskog jezika na mašinski jezik, i svaki prevedeni iskaz izvršavaju pre nego što pređu na prevođenje sledećeg iskaza. Izvršavanjem programa (napisanog na višem programskom jeziku) uz pomoć interpretera, tzv. interpretiranjem programa, ne proizvodi se program na mašinskom jeziku. Interpreteri su lakši za korišćenje od prevodilaca, omogućuju brže otkrivanje grešaka i zahtevaju manje memorijskog prostora, ali su manje efikasni od prevodilaca, jer zahtevaju prevođenje programa pri svakom izvršavanju.

2.3 Funkcionisanje računarskog sistema

Pošto se napiše program na višem programskom jeziku, unese u računar, prevede na mašinski jezik i poveže sa drugim prevedenim programskim celinama (i tako dobije izvršni program), može se uneti u operativnu memoriju i izvršavati. Izvršavanje mašinskog programa odvija se instrukcija po instrukcija.

Izvršavanje pojedinačne mašinske instrukcije odvija se u **mašinskom ciklusu**. On se sastoji od dve faze: prvo, kontrolna jedinica procesora donosi instrukciju iz operativne memorije u procesorski registar instrukcija, dekodira je i identifikuje operaciju koja treba da se izvrši i adrese operanada. Zatim se potrebni podaci donose sa identifikovanih adresa memorije u registre podataka, ALJ izvršava operaciju, a rezultat privremeno smešta u akumulator, do njegovog smeštanja u memoriju. Zatim se izvršava sledeća instrukcija, a adresa na kojoj se ona nalazi izračunava se na osnovu adrese prethodne izvršene instrukcije, dodavanjem broja bajtova koje je zauzimala prethodna instrukcija. Mašinski ciklusi se ponavljaju do instrukcije zaustavljanja tj. kraja programa.

Ovakva metoda izvršavanja programa naziva se **sekvencijalnom** ili **serijskom** obradom, i odgovara konceptu fon Nojmanove mašine. Osim ovakvog, postoji i **paralelni** način obrade, pri čemu se program "razbija" na delove koji se istovremeno izvršavaju na desetine, stotine ili čak hiljade procesora (tzv. masovno paralelni nizovi procesora).

3

Karakterski skup i kodne sheme

3.1 Karakterski skup

Za tekstualnu komunikaciju čoveka sa računarom (npr. preko tastature i ekrana terminala), tj. za saopštavanje programa i podataka računaru i dobijanje izlaznih informacija iz računara, potrebno je, pre svega, imati precizno definisanu "spoljašnju" azbuku tj. konačni skup nedeljivih slova (znakova, simbola) – npr. {A,B, ... Z,1,2, ..., 9, ?}. U računaru se različiti simboli spoljašnje azbuke kodiraju različitim varijacijama sa ponavljanjem nula i jedinica fiksne dužine, tzv. kodnim rečima ili kôdom fiksne dužine nad azbukoma {0, 1} (binarnom azbukom).

Svaki računar ima svoj skup karaktera za komunikaciju sa korisnikom, i taj skup nije standardizovan. Karakteri imaju svoj spoljašnji oblik (glif, šara, grafički oblik koji prepoznajemo kao znak u određenom sistemu pisanja – može biti slovo, cifra, interpunkcijski ili specijalni znak, itd.) koji vidimo na ekranu (npr. A, a, +, }, [, 0, itd.) i svoju unutrašnju reprezentaciju u binarnom obliku, koja odgovara nekoj standardnoj kodnoj shemi.

3.2 Kodne sheme

Objasnimo ukratko principe, strukturu i svojstva kodnih shema (pronaći na Internetu izvore informacija o karakterima i raznim načinima za njihovo kodiranje).

Danas su najčešće u upotrebi kodne reči dužine 7 odnosno 8 bitova (ukupno 128 odnosno 256 kodnih reči) i 16 bitova (ukupno 65536 kodnih reči). Kodne reči u jednom kodu fiksne dužine nazivaju se *karakteri*. Ako se za registrovanje jednog znaka spoljašnje azbuke u računaru koristi jedan bajt (kao što je to slučaj sa 7-bitnim odnosno 8-bitnim kodnim rečima), onda važi i da sadržaju jednog bajta, bez obzira da li on predstavlja deo binarnog zapisa grafičke, zvučne ili tekstualne informacije, odgovara jedan znak spoljašnje azbuke sa grafičkim likom ili informacija o kontroli

uređaja ili toka podataka (na primer, kraj reda, kraj datoteke, prelaz u novi red, zvono i sl.), tzv. kontrolni karakter.

3.2.1 7-bitne kodne sheme

Najrasprostranjeniji je 7-bitni kôd koji je 1983. godine standardizovan međunarodnim (ISO – International Standard Organization) standardom. Najpoznatija je njegova nacionalna američka verzija koju je definisao ANSI – American National Standards Institute 1968. godine u dokumentu American Standard Code for Information Interchange po kome se ovaj kôd i zove ASCII-kôd. Ova kodna shema je i bila osnov za definisanje standarda, ali i za niz drugih nacionalnih verzija.

U ASCII kôdu se, na primer, znak 'A' zapisuje kao 1000001, znak 'B' kao 1000010, znak '0' kao 0110000, znak '9' kao 0111001, itd. Kako se vrednost ovih binarnih brojeva može sračunati u dekadnom sistemu (npr. $1000001_2 = 1\times2^6+0\times 2^5+0\times2^4+0\times2^3+0\times2^2+0\times2^1+1\times2^0 = 64_{10}+1_{10} = 65_{10}$, to se i ASCII kodovi obično zapisuju (za komunikaciju među ljudima) u dekadnom sistemu, pa je kôd za znak 'A' – 65, za znak 'B' – 66, za znak '0' – 48, za znak '9' – 57, itd. Takođe se koristi i zapis u heksadekadnom sistemu (sistemu sa osnovnom 16) u kome je kôd za znak 'A' – 41_{16}, za znak '0' – 30_{16}, ili 0x41, 0x30, redom.

Navedimo neke principe na kojima je struktuiran ASCII kôd, i njegove osobine.

Prva 32 karaktera (kodovi 0–31) i poslednji karakter (kôd 127) su kontrolni karakteri. To su karakteri bez grafičkog lika, kao CR (Carriage Return) – vraćanje na početak reda (kôd 13), LF (Line Feed) – prelaz na novi red (kôd 10) itd.

Skup simbola spoljašnje azbuke je uređen prema uređenju njihovih karaktera - unutrašnjih kodova u odgovarajućoj kodnoj shemi. U ASCII kodu, skup velikih slova A–Z (kodovi 65–90), kao i skup malih slova a–z (kodovi 97–122), kodiran je u abecednom poretku (65<66 prema tome 'A' <'B'). Skup cifara 0-9 (kodovi 48–57) je u rastućem brojčanom poretku ($49 < 50$, prema tome '1'<'2'). Skup velikih slova, skup malih slova i skup cifara su uzastopni, tj. između slova A i slova Z nema drugih karaktera osim onih koji odgovaraju velikim slovima engleske abecede, slično važi za mala slova, odnosno za dekadne cifre. Sve cifre prethode svim velikim slovima, sva velika slova prethode svim malim slovima. Specijalni i interpunkcijski znaci su izmešani između njih. Kôd svakog velikog slova je za 32 (odnosno 2^5) manji od kôda odgovarajućeg malog slova. Na primer, za kôd slova **E** (69) i **e** (101) važi da je 69+32=101, odnosno, 01000101+00100000=01100101.

Međunarodni – ISO standard 7-bitnog kôda čija je ASCII nacionalna američka verzija, ostavlja prostor i za druge nacionalne verzije, u kojima postoje ne-engleska slova. Tako ovaj standard propisuje da se pozicijama (kodovima, karakterima) 64, 91–94, 96 i 123–126 ne pridružuje obavezan grafički lik, već da se one u nacionalnim verzijama standarda i u određenim aplikacijama mogu slobodno koristiti. Jugoslovenski standard (tzv. YUSCII, koji je definisao Jugoslovenski zavod za standarde 1986. godine) koristi ovih 10 pozicija za kodiranje slova specifičnih za

srpsku latinicu na sledeći način (u zagradama su navedeni znaci koji su u ASCII kodu predstavljeni odgovarajućim kodovima tj. karakterima):

<table>
<tr><th>velikoSlovo</th><th>kôd</th><th>maloSlovo</th><th>kôd</th></tr>
<tr><td>Ž(@)</td><td>64</td><td>ž(‘)</td><td>96</td></tr>
<tr><td>Š([ )</td><td>91</td><td>š({)</td><td>123</td></tr>
<tr><td>Đ(\)</td><td>92</td><td>đ(|)</td><td>124</td></tr>
<tr><td>Ć(])</td><td>93</td><td>ć(})</td><td>125</td></tr>
<tr><td>Č(ˆ)</td><td>94</td><td>č(˜)</td><td>126</td></tr>
</table>

YUSCII skup zadržava sve navedene osobine ASCII kôda osim jedne, a ta je da ni velika ni mala slova nisu u abecednom poretku. Na primer, 64 < 65, dakle 'Ž' < 'A' što ne odgovara abecednom poretku.

3.2.2 Proširenja 7-bitnog kôda

Sedmobitna standardna kôdna shema ostavlja svega 95 karaktera (pozicija) za kodiranje karaktera spoljašnje azbuke sa glifom (grafičkim likom). To je nedovoljno. Zato sledeći ISO standard uvodi 8-bitno prošrenje 7-bitnog kôda, i to kao uniju "donje" i "gornje" kodne stranice: donja kodna stranica uključuje karaktere sa nepostavljenim najvišim bitom (0) - ona odgovara ASCII kodnoj shemi, dok gornja kodna stranica uključuje karaketere sa postavljenim najvišim bitom (1). Ovim proširenjem se dobija još 96 pozicija (karaktera) za kodiranje znakova sa glifom (od 128 karaktera u gornjoj stranici, 32 se opet rezerviše za kontrolne karaktere i to su karakteri 128 - 159). Dakle, na raspolaganju je 95 (iz donje kodne stranice) +96 (iz gornje kodne stranice) karaktera za vidljive znake (znake sa glifom), što nije ni blizu dovoljno za kodiranje raznih jezika i pisama. Zbog toga ovaj 8-bitni ISO standard predstavlja jednu seriju standarda, nazvanu serijom ISO 8859, koja uključuje čitav niz verzija 8-bitne kodne sheme tj. čitav niz *karakterskih skupova*: ISO 8859-1 (Latin-1), za kodiranje većine zapadnoevropskih jezika, ISO 8859-2 (Latin-2), za kodiranje većine srednjeevropskih jezika i slovenskih jezika sa latiničnim pismom - tu su kodovi i za slova srpske latinice, 8859-3 (Latin-3), za kodiranje esperanta, turskog, malteškog, 8859-4 (Latin-4) za kodiranje estonskog, letonskog, litvanskog, 8859-5 (Latin/Cyrillic) za kodiranje većine slovenskih jezika koji koriste ćirilično pismo, uključujući i srpsku ćirilicu, itd.

Slova specifična za srpsku latinicu u Latin-2 nalaze se na sledećim pozicijama:

velikoSlovo	kôd	maloSlovo	kôd
Ž	174	ž	190
Š	175	š	191
Đ	208	đ	240
Ć	198	ć	230
Č	200	č	232

Može da se uoči da se karakteri (kodovi) za parove veliko-malo slovo specifično za srpsku latinicu u Latin-2, razlikuju ili za 16 - bit na poziciji 5 ili za 32 - bit na poziciji 6. Ostalim (nespecifičnim) slovima srpske latinice odgovaraju isti karakteri kao i u osnovnom ASCII skupu.

Pomenimo i karakterski skup 8859-5 (Latin/Cyrillic): Prvih 128 karaktera opet je identično osnovnom ASCII skupu, dok slovima srpske ćirilice A–Ш (odnosno a–ш) odgovaraju karakteri 176-200 (odnosno za po 32 veći – 208-232, sa nekoliko ubačenih slova ruske ćirilice), uz izuzetak ćiriličnih slova Ђ (162), Ј, Љ, Њ, Ћ(168-171) i Џ – 175, i odgovarajućih malih slova čiji su kodovi veći za po 80.

3.2.3 Industrijski standardi

Razni proizvođači računara i softvera izgradili su sopstvene kodne sheme koje nazivaju i *kodnim stranama*. Neke od tih kodnih shema postale su i tzv. industrijski standardi.

Jedan od prvih industrijskih standarda kodiranja bio je IBM-ov 8-bitni EBCDIC kôd (Extended Binary Coded Decimal Interchange Code), uveden još davne 1963. godine za računare IBM System/360. Korišćen je na svim IBM mainframe računarima ali je patio od ozbiljnih nedostataka: susedna slova nisu bila na susednim pozicijama, neki značajni interpunkcijski znaci (prisutni u ASCII) su nedostajali. Kritičari ga nazivaju i ”karakterskim skupom za IBM dinosauruse”.

Jedna značajna familija industrijskih standarda je skup Microsoft Windows kodnih strana poznatih kao ANSI kodne strane. To su 8-bitna proširenja ASCII kôda – analogoni ISO 8859 standardnim kodnim stranama. Prva od njih, 1252, bila je bazirana na nepotvrđenom ANSI nacrtu standarda ISO 8859-1, ali koristi svih 128 karaktera gornje kodne stranice za znakove sa grafičkim likom, tj. ukupno je na raspolaganju 223 karaktera u svakom karakterskom skupu; to omogućuje kodiranje i nekih specijalnih simbola koji nisu predviđeni 8859 standardom, kao na primer znak za evro, ligatura oe, i sl.

Neki iz ove familije standarda su:

- CP1250 (WinLatin2) - centralno evropska i istočno evropska latinica
- CP1251 (WinCyrillic) - ćirilica
- CP1252 (WinLatin1) - zapadnoevropska latinica
- CP1253 - grčko pismo
- CP1254 - tursko pismo
- CP1255 - hebrejsko pismo
- CP1256 - arapsko pismo, itd.

3.2.4 Unicode

Već je utvrđeno da su kodne sheme sa 8-bitnom kodnom reči nedovoljne za kodiranje raznih jezika i pisama, pa 8-bitni ISO standard zato i nije jedinstveni standard već predstavlja čitavu familiju kodnih shema.

Krajem 80-tih i početkom 90-tih godina međunarodna organizacija za standardizaciju (ISO) odlučila je da uvede novi, jedinstveni standard kodiranja, koji mora da bude višebajtni da bi podržao sva pisma i jezike na Zemlji. Tako je nastao *univerzalni karakterski skup - UCS* Universal Multiple-Octet Coded Character Set 4 (ISO 10646), standard 4-bajtovskog kodiranja, dovoljan za sva postojeća pisma, uključujući i slikovna kinesko-japansko-koreanska (CJK) pisma, drevne jezike, naučne notacije itd. Radna verzija ovog standarda objavljena je 1990.g, a prvih 256 karaktera bilo je identično ISO 8859-1 karakterskom skupu.

Nekako u isto to vreme, konzorcijum sastavljen od predstavnika Xerox, Apple, Sun MicroSystems, Microsoft, NeXT i drugih industrijskih korporacija, počeo je da razvija univerzalni karakterski skup – industrijski standard višebajtnog kodiranja, dovoljan za kodiranje živih jezika. Kao više nego dovoljan odabran je dvobajtni (16-bitni) kôd (obezbeđuje 65536 različitih karaktera). Novi sistem kodiranja nazvan je Unicode, s obzirom na tri "UNI"-cilja:

- Univerzalan (UNIversal) da "pokrije" sve savremene jezike sa pismom
- Jedinstven (UNIque), bez dupliranja karaktera - kodiraju se pisma a ne jezici
- Uniforman (UNIform) – svaki karakter kodira se istim brojem bitova: 16

Godine 1991. objavljen je standard Unicode 1.1.

Potreba za jedinstvenim usmeravanjem tržišta rezultovala je koordinacijom ISO standarda i industrijskog Unicode standarda višebajtnog kodiranja, od 1993. godine. ISO je usaglasio "osnovnu multilingvalnu ravan" svog kôda (prvih 2^{16} tj. 65536 karaktera) sa repertoarom Unicode-a (razvijena je standardna (ISO) verzija dvobajtnog kodiranja, UCS-2), a sam Unicode, pošto je ustanovljeno da 16-bitni kôd ipak nije dovoljan, proširio je svoj repertoar na oko 1000000 kodova. Uvedena je kodna šema UTF-16 koja sve karaktere UCS-2 tj. Unicode-a kodira na identičan način (njih 2^{16} tj. 65536, sa po dva bajta), a još 16 takvih skupova od po 2^{16} karaktera kodira sa po 2 16-bitne kodne jedinice (tj. 4 bajta).

Jedna zanimljiva ali loše realizovana mogućnost predviđena ISO standardom - neuniformno kodiranje različitih grupa karaktera (najčešće korišćeni – jednim bajtom, ređe korišćeni – sa dva bajta, itd. do 5 bajtova), doživela je svoju široko prihvaćenu, efikasnu implementaciju kroz tzv. UTF-8 (Unicode Transformation Format) sistem kodiranja promenljive dužine kodova.

UTF-8 je kodiranje karaktera Unicode-a promenljivom dužinom bajtova. Može da predstavi sve karaktere Unicode-a, a kompatibilan je sa ASCII. U UTF-8, ASCII skup se kodira jednim bajtom (na identičan način), sledećih 1920 karaktera Unicode-a zahteva 2 bajta za kodiranje u UTF-8 – to su latinice, ćirilice, grčko, arapsko, hebrejsko pismo i sl. Ostatak osnovne multilingvalne ravni (do 65536

karaktera) kodira se sa 3 bajta. Četiri bajta koristi se za ostale "ravni" proširenog Unicode-a (skupove od po $2^{16} = 65536$ karaktera). UTF-8 kodiranje je takvo da se na osnovu prvih nekoliko bitova jednog bajta može jednoznačno odrediti da li je reč o karakteru koji se kodira jednim ili većim brojem bajtova.

Repertoar Unicode-a Osnovni Unicode ima $2^{16} = 65536$ karaktera. Prve 8192 pozicije (u heksadekadnom zapisu to su pozicije 0x0000 0x2000) rezervisane su za standardne alfabete, kao što su latinice, ćirilice, grčko, arapsko, hebrejsko, tai, sirijsko pismo i sl. Pri tome, prvih 256 karaktera identično je sa ISO 8859-1.

Kodovi slova specifičnih za našu latinicu u Unicode su:

velikoSlovo	kôd	maloSlovo	kôd
Ć	262	ć	263
Č	268	č	269
Đ	272	đ	273
Š	352	š	353
Ž	381	ž	382

Kodovi velikih slova srpske ćirilice smešteni su od 0x410-0x428 (u dekadnom sistemu 1040–1064) sa izuzetkom (kao i u ISO 8859-5) ćiriličnih slova Ђ, Ј, Љ, Њ, Ћ i Џ koja im prethode, dok su kodovi malih ćiriličnih slova za po 32 manji, osim pomenutih 6 kod kojih su kodovi malih slova veći za 80 od kodova odgovarajućih velikih slova.

Sledećih 4096 pozicija (u heksadekadnom zapisu to su pozicije 0x2000 0x3000) rezervisano je za specijalne karaktere (npr. razne vrste kvadratića, trouglića, zvezdica, interpunkcijskih znakova, oznaka valuta, matematičkih simbola i sl).

CJK ideografsko (slikovno) pismo zauzima pozicije 0x4E00-0x9FFF, ukupno 20924 karaktera.

Unicode konzorcijum neprestano dopunjuje Unicode repertoar.

4

Azbuka, reč, jezik, jezički procesori

4.1 Azbuka, niska

Svaki jezik za komunikaciju čoveka i računara koristi konačni skup simbola (znakova) – *azbuku.* Simbol (znak, slovo) je nedeljiva jedinica jezika. Nizanjem slova azbuke dobiju se *niske* nad azbukom. Dužina niske je broj slova u nisci. Ako je w – niska, njena dužina obično se obeležava sa $|w|$. Niska koja ne sadrži nijedno slovo zove se *prazna niska* i obeležava se sa e (engl. empty – prazan). Ako je a – slovo azbuke, sa a^i označava se niska $aa \ldots a$ koja sadrži i slova a. Tako a^0 označava praznu nisku, $a^1 = a$, $a^2 = aa$, itd.

Formalno, niska nad azbukom A definiše se na sledeći način:

1. e je niska nad A;
2. ako je w niska nad A i a – slovo iz A, onda je wa niska nad A;
3. sve niske nad azbukom A mogu se dobiti primenom pravila 1. i 2.

Osnovna operacija nad niskama je dopisivanje (konkatenacija) i ona se definiše na sledeći način:

Ako su w, u – niske, $|w|{=}x$, $|u|{=}y$, onda je rezultat *dopisivanja* (*konkatenacije*) niske u na nisku w niska wu dužine $|wu|{=}x+y$, i kojoj je i-to slovo jednako i-tom slovu niske w za $i \leq x$, tj. jednako $i-x$-tom slovu niske u za $x < i \leq x+y$.

Ako je A – azbuka sa n slova, $A = \{a_1, a_2, \ldots, a_n\}$, onda oznaka A^* označava beskonačni skup svih mogućih niski nad azbukom A ($A^* = \{e, a_1, a_2, \ldots, a_n, a_1a_1, a_1a_2, \ldots, a_1a_n, a_2a_1, a_2a_2, \ldots, a_1a_1a_1, \ldots\}$).

Oznaka A^+ označava skup svih mogućih niski nad azbukom A dužine ≥ 1 ($A^+ = A^* \setminus e$).

4.2 Jezik

Jezik L nad azbukom A je neki podskup skupa A^* svih niski nad azbukom. Kako

je skup A^* beskonačan (za nepraznu azbuku), on ima i beskonačno mnogo podskupova, pa nad azbukom A ima i beskonačno mnogo jezika. U zadavanju jezika L potrebno je upravo tačno definisati taj podskup skupa A^* koji predstavlja niske nad azbukom A koje pripadaju jeziku L – *reči* jezika L (na primer, reč "jezika" celih brojeva je 123, kao i -123, ali ne i 12-3). Kada su konstrukcije jezika složene, nazivaju se rečenicama (na primer, u prirodnom jeziku kao što je srpski), a njihove komponente - rečima. Isti je slučaj i sa programskim jezikom (u kome je svaki ispravni program - jedna rečenica tog jezika). Definisanje reči (tj. rečenica) jezika (i njihovo razlikovanje od niski nad istom azbukom koje ne pripadaju jeziku) obično se vrši primenom nekih pravila. Skup pravila kojima se opisuju sve ispravne rečenice jezika, zove se *sintaksa jezika*, dok se skup pravila kojima se opisuju njihove komponente – reči zove *leksika jezika*.

Semantika jezika je skup pravila kojima se definiše značenje reči jezika. *Pragmatika jezika* odnosi se na upotrebu jezika, tj. na mogućnost jezika da na više načina izrazi isti pojam ili akciju (na primer, razni iskazi ciklusa u programskom jeziku).

Jezik je sredstvo za komunikaciju između dva ili više korisnika i mora da bude prihvatljiv za sve korisnike. Ako su svi korisnici jezika – ljudi, onda je jezik komunikacije obično **prirodni jezik** (srpski, engleski, francuski, itd.). Prirodni jezici su ekspresivni (univerzalni), i njima se može izraziti svaka informacija, ali su nejednoznačni i neprecizni u predstavljanju specifičnih informacija kao što su matematičke. Zato se izgrađuju veštački jezici u specifičnim oblastima, npr. jezik matematičkih formula u matematici, jezik hemijskih formula u hemiji, jezik saobraćajnih znakova u saobraćaju, itd.

Posebna klasa veštačkih jezika, u slučaju da je jedan od korisnika jezika – računar, jesu programski jezici. Osnovni motiv za uvođenje programskih jezika je potreba da se premosti razlika u nivou apstrakcije čoveka i računara. Dok čovek ima potrebu i mogućnost da razmišlja na različitim, ponekad visokim nivoima apstrakcije, računar je ograničen mogućnostima koje su definisane u trenutku njegove izgradnje i izgradnje njegovog operativnog sistema. Programski jezici omogućuju čoveku da formuliše poruku na način koji je za njega relativno prirodan i jednostavan. S druge strane, precizna definicija dopuštenih oblika i njihovog značenja omogućuje izgradnju jezičkih procesora (prevodilaca i interpretera) – programa pomoću kojih se čovekova poruka prevodi na jezik "razumljiv" mašini – mašinski jezik.

Postoje razne definicije programskog jezika. Jedna od njih je definicija Američkog instituta za standarde (ANSI) koja glasi: "Programski jezik je jezik koji se koristi za pripremanje računarskih programa". Ova definicija, pored toga što je veoma opšta, ne odražava tekuću upotrebu jezika, tj. ne pravi razliku između raznih vrsta programskih jezika. Ta razlika u filozofiji, strukturi i upotrebi raznih programskih jezika najbolje se sagledava kroz klasifikaciju programskih jezika.

Programski jezici kao način komunikacije između čoveka i računara primenjuju se od druge polovine pedesetih godina 20. veka, kada je definisana prva verzija programskog jezika FORTRAN (FORmula TRANslation) i napravljen prevodilac

za njega (Džon Bekus – John Backus, SAD, 1957). Od tada pa do danas razvijeno je više stotina programskih jezika (http://en.wikipedia.org/wiki/List_of_programming_languages).

Za prirodni jezik (na primer srpski) postoje sintaksna pravila za izgradnju ispravnih konstrukcija (rečenica), ali ta pravila ne pokrivaju sve ispravne konstrukcije jezika. To nije ni čudno jer je prirodni jezik dinamičan, on živi i menja se u vremenu i prostoru. Za veštački jezik, i posebno programski jezik, potrebno je da se veoma precizno opiše njegova sintaksa, tj. sve i samo one konstrukcije koje pripadaju jeziku (npr. konstanta, identifikator, izraz, iskaz, funkcija, program, itd.), da bi se omogućila izgradnja sredstava koja će automatski, bez pomoći čoveka, ustanoviti da li je neka konstrukcija ispravna konstrukcija programskog jezika. Ta sredstva su programi – jezički procesori.

Mada obezbeđuje izgradnju i prepoznavanje ispravne konstrukcije jezika, sintaksa ne govori ništa o smislu te konstrukcije. Tako sintaksno ispravna rečenica u prirodnom jeziku može biti potpuno besmislena (ili višesmislena), ali se svakoj sintaksno ispravnoj konstrukciji programskog jezika može (jednoznačno) pridružiti njeno značenje.

4.3 Jezički procesori

Precizna formulacija sintakse (i semantike) programskih jezika omogućuje izradu programa za obradu jezika – *jezičkih procesora*. Ovi programi analiziraju sintaksnu ispravnost programa na programskom jeziku i, ako je program ispravan, transformišu ga u binarni (mašinski) oblik koji može da se izvrši na računaru.

U zavisnosti od toga da li se ceo program proanalizira i transformiše, pre nego što može da se izvrši, ili se analiza i izvršenje programa obavljaju naizmenično – deo po deo programa (npr. naredba po naredba), jezički procesori se dele u dve vrste: *kompilatore* i *interpretere*.

Izvršavanje programa P na jeziku L može da se ostvari, primenom interpretera, na način prikazan slikom 4.1.

Način rada interpretera opisuje se sledećim postupkom: uzima se instrukcija programa, vrši se njena sintaksna analiza, zatim semantička obrada, i prelazi se na sledeću instrukciju. Postupak se ponavlja sve dok se ne naiđe na instrukciju kraja programa. Semantička obrada izvršnih instrukcija svodi se na njihovo izvršavanje. Opisne instrukcije se obrađuju tako što se informacije iz njih pamte u tabelama i koriste pri obradi izvršnih instrukcija. Dakle, analiza i izvršavanje programa uz pomoć interpretera su vremenski povezani.

Ulazni podaci interpretera su program P i ulazni podaci D programa P. Interpreter uzima podatke onda kada instrukcije koje interpretira to zahtevaju.

Rezultat rada interpretera je rezultat izvršavanja programa P (nema prevedenog programa P na mašinskom jeziku).

U toku rada interpreter mora da održava sve radne podatke programa P.

Jezici za koje se grade interpreteri nazivaju se *interpreterskim (interaktivnim)* jezicima.

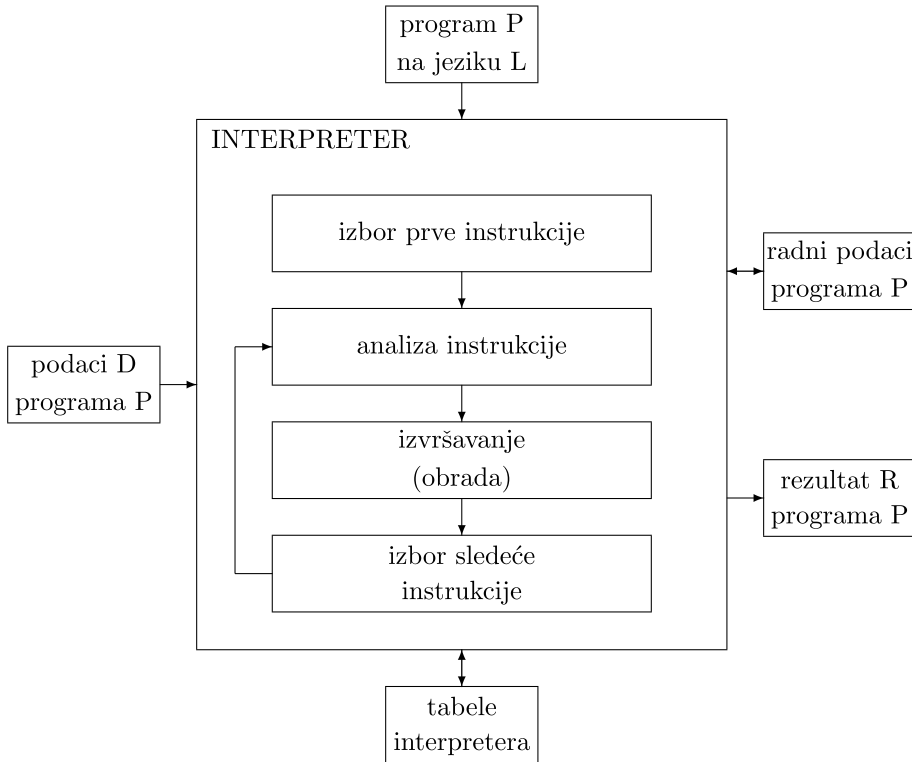

Figure 4.1: Izvršavanje programa uz pomoć interpretera

S druge strane, izvršavanje programa P na jeziku L može se ostvariti primenom kompilatora na način prikazan na slici 4.2.

Kompilatori su jezički procesori koji prevode program sa jednog (*izvornog*) jezika na drugi (*izlazni, ciljni*). Izvršavanje programa uz pomoć kompilatora odvija se u dva koraka:

1. Program P (u izvornom jeziku L) transformiše se u program P' na izlaznom jeziku L', pod kontrolom kompilatora. Dakle, ulazni podatak kompilatora je program P a rezultat rada kompilatora je program P'. U okviru ovog koraka (kompilacije) razlikuje se veći broj faza. *Leksičkom analizom* utvrđuje se korektnost osnovnih elemenata rečenice jezika – reči ili *leksema* (u programskom jeziku to su identifikatori, operatori, konstante i sl; na primer, u izrazu *i+1*, lekseme su "i" (identifikator), "+" (operator) i "1" (konstanta)). *Sintaksnom analizom* utvrđuje se da li su reči ispravno složene u rečenicu jezika, tj. u programskom jeziku – da li su lekseme ispravno složene u program (ili neku manju programsku celinu – izraz, naredbu, funkciju; na primer, izraz *i+1* je dobro komponovan od leksema, tj. sintaksno ispravan, dok izraz *i+1+* nije

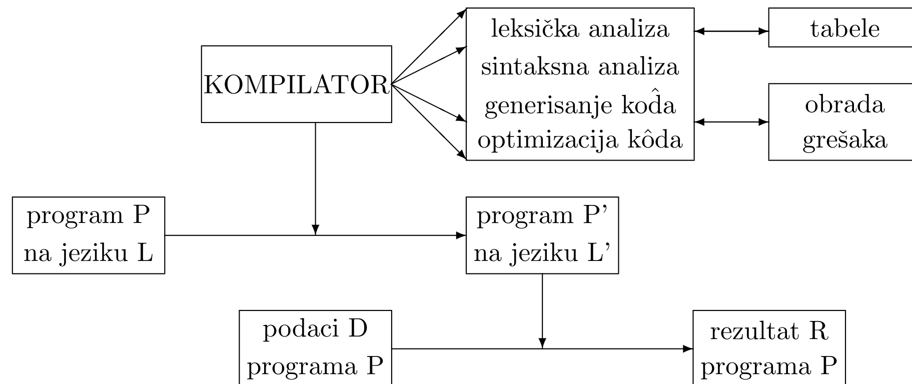

Figure 4.2: Izvršavanje programa uz pomoć kompilatora

dobro komponovan – nije sintaksno ispravan). Generisanje koda proizvodi ekvivalentan program na nekom nižem, jednostavnijem jeziku (može da bude i mašinski), dok optimizacija koda obezbeđuje da se eliminišu nepotrebno proizvedeni delovi transformisanog koda.

2. Dobijeni program P' se interpretira, tj. pod kontrolom programa P' ulazni podaci D programa P se transformišu u rezultat R programa P.

   Kada je izlazni jezik L' – mašinski jezik računara, interpretaciju vrši neposredno računar i ona se sastoji u izvršavanju izlaznog programa P'. U tom slučaju postoji i međukorak između prevođenja i interpretacije – *povezivanje* (engl. link) izlaznog programa P' sa procedurama koje se pozivaju iz izvornog programa P.

   Kada je izlazni jezik L' neki međujezik (nižeg nivoa od jezika L a višeg od mašinskog), interpretaciju vrši odgovarajući interpreter jezika L' (prema prethodno opisanom postupku izvršavanja programa uz pomoć interpretera).

   Dakle, pri izvršavanju programa uz pomoć kompilatora faze prevođenja i izvršavanja su razdvojene i vremenski mogu biti udaljene.

   Jezici za koje se grade kompilatori nazivaju se *kompilatorskim* jezicima tj. jezicima za paketnu obradu.

4.3.1 Prednosti i nedostaci interpretera i kompilatora

Interpretacija i kompilacija programa su dva komplementarna pristupa.

Osnovna prednost interpretera je što omogućuje neposredni pristup korisnika procesu računanja, izvornom programu i podacima, jer su i program i podaci prisutni u toku izvršavanja. Uočena greška može odmah da se ispravi i da se nastavi sa interpretacijom. Interpeteri su zato izuzetno pogodni u fazi rada na programu (u toku razvoja programa).

Interpreteri analiziraju i izvršavaju jednu po jednu instrukciju (naredbu, iskaz) izvornog programa. Ako postoje ciklusi, ista instrukcija se analizira više puta i usporava izvršavanje programa. Pri svakom izvršavanju programa (za nove ulazne podatke), program se ponovo interpretira, što uključuje, pored ponovnog izvršavanja programa, i ponavljanje sintaksne analize. Sporost izvršavanja programa koja je rezultat analize pri izvršavanju programa uz pomoć interpretera je osnovni nedostatak interpretera.

Interpreteri se, zbog svojih svojstava, primenjuju na jezike kod kojih izvorni program ima samo jednu instrukciju (npr. jezici u sastavu tekst editora), ili kada se svaka instrukcija programa izvršava samo jedanput (npr. jezici u sastavu operativnih sistema). Interpreteri su pogodni i za jezike kod kojih je vreme analize instrukcije zanemarljivo u odnosu na vreme njenog izvršavanja (npr. problemski orijentisani jezici) kao i za jezike visokog nivoa kod kojih nema izraženih upravljačkih struktura kao ni izražene tipiziranosti, pa je moguće redom analizirati i izvršavati komponentu po komponentu programa (npr. PROLOG, BASIC, LISP).

Kompilatori, posebno kada je izlazni jezik mašinski, imaju osnovnu prednost u brzini svakog izvršavanja jednom prevedenog programa. Osnovni nedostatak izvršavanja programa uz pomoć kompilatora je što prevedeni program gubi svaku vezu sa izvornim programom. Svaka (i najmanja) izmena u izvornom programu zahteva ponovno prevođenje celog programa. Zato se kompilatori primenjuju na jezike u kojima se pišu programi sa intenzivnom numeričkom obradom (za koje je važno da se izvršavaju što brže). Takođe, kompilatori su pogodni za jezike sa veoma izraženom strukturom i strogom tipiziranošću, s obzirom da jedna instrukcija može da se protegne na veliki segment programa.

Čest je slučaj da se za jedan izvorni (programski) jezik gradi i interpreter i kompilator. Intepreter se tada koristi u fazi razvoja programa da bi omogućio interakciju korisnika sa programom, dok se u fazi eksploatacije (korišćenja) kompletno razvijenog i istestiranog programa koristi kompilator koji proizvodi program sa najefikasnijim izvršavanjem.

Pristup koji kombinuje prednosti interpretera i kompilatora je prevođenje izvornog jezika L na međujezik L' (koji nije mašinski ali je niskog nivoa, obično jezik neke apstraktne virtuelne mašine) i koji se zatim interpretira. Time se održava veza programa koji se interpretira sa ulaznim podacima izvornog programa (mada ne i sa samim izvornim programom), a izlazni jezik je obično takav da je njegova interpretacija dovoljno brza (znatno brža od interpretacije samog izvornog programa). I izmene u izvornom programu tada se lakše izvode, jer je prevođenje na međujezik L' znatno brže od prevođenja na mašinski jezik. Ovaj pristup se primenjuje, npr. kod programskih jezika Java, C#.

5

Rešavanje problema uz pomoć računara

Rešavanja problema uz pomoć računara podrazumeva proizvodnju (pisanje) programa na nekom programskom jeziku, čijim izvršavanjem će se rešavati svaki pojedinačni zadatak tog problema. Da bi se program proizveo, potrebno je odabrati metodu za rešavanje problema, poznavati (ili razviti) algoritam za rešavanje problema, poznavati programski jezik, zapisati algoritam na programskom jeziku, uveriti se u korektnost programa njegovim izvršavanjem za razne kombinacije ulaznih podataka. Jedini jezik na kome računar može da izvršava program jeste mašinski jezik tog računara. To je jezik najnižeg nivoa, u kome su sve "naredbe" (tzv. instrukcije) i podaci predstavljeni binarnom azbukom, i koji je zbog toga neodgovarajući za čoveka. Da bi se program izvršavao, potrebno je imati programske alate koji će omogućiti rad sa našim programom, od unošenja u računar, preko prevođenja na mašinski jezik, do testiranja programa i otkrivanja grešaka. Ako su u programu otkrivene greške, program se "ispravlja" (ponavlja se faza pisanja programa), a ako je potrebno, primenjuje se (ili razvija) drugi algoritam, odnosno menja metoda rešavanja problema. Pošto se izradi korektan i efikasan program za rešavanje zadatog problema, on se primenjuje kraće ili duže vreme, uz eventualne povremene modifikacije koje odgovaraju promenama u postavci problema. Dakle, u rešavanju problema uz pomoć računara javlja se veći broj poslova koje treba obaviti. Ovi poslovi mogu se, koncepcijski, podeliti u nekoliko faza, koje čine *životni ciklus* programa. Životni ciklus programa je vreme od početka razvoja programa (od početka rešavanja problema uz pomoć računara), do kraja korišćenja programa, a sam termin asocira na ponovljeno (ciklično) izvršavanje tih faza, što i odgovara prirodi ovog procesa. Te faze su sledeće:

1. specifikacija
2. projektovanje
3. realizacija

4. testiranje
5. izrada dokumentacije
6. eksploatacija i održavanje

Specifikacija je postupak opisivanja problema za koji se traži programsko rešenje. Specifikacijom je potrebno što preciznije opisati problem, prirodu ulaznih podataka i oblik u kome se žele rešenja – izlazni rezultati. Specifikacija programa bavi se pitanjem šta program treba da uradi, kojom brzinom, koja mu je maksimalna dozvoljena veličina, itd.

Specifikacija se može izraziti raznim jezicima sa različitim sintaksama. Mada postoje i posebni jezici za pisanje formalne specifikacije, ona se može (u jednostavnijim slučajevima) izraziti i prirodnim jezikom, npr. "smestiti u z proizvod brojeva a, b, pod pretpostavkom da su a, b – nenegativni celi brojevi".

Projektovanje predstavlja razumevanje problema, izradu matematičkog modela i izbor ili razvoj globalnog algoritma.

Realizacija predstavlja implementaciju, ili ostvarenje rešenja. Ona uključuje detaljnu razradu algoritma, izbor struktura podataka i medija na kojima će se podaci držati (sa kojih će se unositi ili na koje će se izdavati), izradu sintaksno ispravnog programa na izabranom programskom jeziku (programa koji je napisan prema pravilima kojima je definisan taj jezik). Da bi se utvrdila sintaksna ispravnost programa, potrebno je uneti program u računar (npr. preko tastature, korišćenjem programa – editora), prevesti program na mašinski jezik – pomoću programa – prevodioca (pri čemu se ispituje sintaksna ispravnost) i izvršiti ga (npr. uz pomoć programa za otkrivanje grešaka), ili ga interpretirati (analizirati sintaksnu ispravnost programa i prevoditi ga deo po deo uz istovremeno izvršavanje analiziranih delova). Program na mašinskom jeziku treba uspešno da bude povezan sa drugim programskim celinama koje poziva. Ako se pri prevođenju programa ili njegovom interpretiranju i izvršavanju ustanovi neka greška, potrebno je ispraviti program, ponovo ga prevesti i izvršiti. Greške koje se pri prevođenju programa mogu otkriti jesu sintaksne greške (npr. naredba nije završena znakom ";" ili je ispušten operand, npr. $a + -c$ umesto $a + b - c$ i sl). Greške pri povezivanju uključuju, npr. nepostojanje programske celine (npr. funkcije) koja se poziva u nekom programu, ili nesaglasnost poziva funkcije sa njenom definicijom (npr. različit broj argumenata ili tipovi argumenata u definiciji i pozivu).

Testiranje predstavlja, pre svega, proces utvrđivanja semantičke ispravnosti programa, tj. uveravanja, u što je moguće većem stepenu, da program izvršava željenu funkciju, tj. da odgovara specifikaciji, tj. da rešava postavljeni problem. Ukoliko je formalno dokazana korektnost programa, ili je program razvijen formalnim metodama iz formalne specifikacije, za njega se može garantovati da je semantički korektan, do na tipografske greške. Tada je testiranje maksimalno pojednostavljeno ili nepotrebno. U svim drugim slučajevima, testiranje je važna faza razvoja programa. Testiranje se obavlja višestrukim izvršavanjem programa za raznovrsne pripremljene ulazne podatke, za koje su nam poznati očekivani rezultati. Značajno

je da pripremljeni ulazni podaci budu dobro odabrani tako da se za njihove razne kombinacije, pri raznim izvršavanjima programa, izvršavaju (testiraju) sve grane programa. Takvi ulazni podaci nazivaju se *test primeri*, i oni omogućuju proveru ispravnosti u što je moguće većoj meri, tako da program ima veliku verovatnoću ispravnosti u primeni. Ovde je bitno naglasiti da se test primerima nikako ne može dokazati korektnost programa, ali se možemo, u visokom stepenu, uveriti da je program ispravan. Pri testiranju programa mogu se otkriti greške pri izvršavanju (npr. deljenje nulom), koje otkriva i prijavljuje sam računar tj. njegov operativni sistem, ili semantičke greške koje može otkriti samo čovek – programer, poređenjem dobijenih rezultata izvršavanja programa sa očekivanim rezultatima. To mogu biti greške nastale pri pisanju programa (npr. umesto uvećanja promenljive i za 1 omaškom smo naveli uvećanje promenljive i za 10 – u sintaksi C-a, $i+=10$), ali to mogu biti i greške u algoritmu (npr. sa brojačem ciklusa, i, pošli smo od vrednosti 0 umesto od vrednosti 1), ili greške u samom matematičkom modelu ili primenjenoj metodi (npr. za rešavanje sistema nehomogenih algebarskih jednačina primenjena je metoda kojom se rešava samo sistem homogenih algebarskih jednačina). Ako je greška ustanovljena na nivou programa, potrebno je vratiti se na korak realizacije. Ako je greška ustanovljena na nivou algoritma ili metode (modela), potrebno je vratiti se na fazu projektovanja, pri čemu se ponavljaju sve naredne faze. Pri testiranju programa pomaže izdavanje tekućih vrednosti promenljivih na karakterističnim mestima u programu, ili testiranje ulaznih podataka (tzv. logička kontrola podataka) da bismo se uverili da će program, kao ispravne ulazne podatke, prihvatiti samo podatke određenog (predviđenog) tipa, oblika i opsega.

Izrada dokumentacije

Kada je program dobro i uspešno istestiran, pristupa se izradi dobre, pregledne i detaljne dokumentacije. Dokumentacija je neophodna za uspešno korišćenje programa, i posebno za fazu održavanja programa koja prati njegovo korišćenje. Dokumentacija treba da sadrži specifikaciju problema, algoritam (globalni i detaljni), čitljivo napisan program (uz dobru meru komentara), način zadavanja ulaznih podataka i način izdavanja rezultata, značenje korišćenih imena (promenljivih, datoteka, programa, potprograma, itd.), rezultate testiranja, test primere, uputstva za korišćenje i održavanje programa.

Eksploatacija i održavanje je faza korišćenja programa koja može da traje i duži niz godina. Održavanje programa podrazumeva izvesne modifikacije programa u toku njegove eksploatacije, koje su posledica promena samog problema u toku vremena, ili potrebe da se kvalitet programa (prostorna i vremenska efikasnost, primenljivost) poveća. Održavanje programa je lakše ako je program preglednije napisan i ako je dokumentacija kompletnija, posebno zbog toga što održavanje programa po pravilu ne vrši lice koje je program pisalo. Svaku izmenu u programu nastalu u fazi održavanja programa potrebno je takođe dokumentovati, tj. pored održavanja programa potrebno je održavati i njegovu dokumentaciju.

5.1 Specifikacija

5.1.1 Dvovalentna logika

Jedan od osnovnih pojmova u programiranju uopšte, i posebno u zadavanju formalne specifikacije, jeste pojam logičkog izraza. U ovoj tački ćemo ukratko objasniti njegovo značenje.

Logička vrednost je istinitosna vrednost iz skupa {T, F} ({true, false}, tj. {tačno, netačno}, tj. {0,1}). Logička konstanta je logička vrednost (razni jezici ih označavaju na razne načine, npr. 0 za netačno, broj $\neq 0$ za tačno u C-u), a logička promenljiva je promenljiva koja uzima logičke vrednosti (i samo njih). U programskim jezicima govorimo o promenljivoj **logičkog tipa**.

Logički izraz se gradi od logičkih konstanti, logičkih promenljivih, poređenja, logičkih funkcija i logičkih operatora, izračunava se po određenim pravilima (vezanim za prioritet i tzv. asocijativnost logičkih operatora) a izračunata vrednost je logička vrednost (T ili F). Najjednostavniji logički izrazi su logička konstanta ili logička promenljiva. Svako poređenje uporedivih vrednosti (brojeva, znakova, itd.) je takođe logički izraz i ima vrednost T ili F.

Nad jednostavnim logičkim izrazima može se graditi složeniji logički izraz primenom logičkih operacija $\neg$ (negacije), $\vee$ (disjunkcije), i $\wedge$ (konjunkcije).

Logičke operacije $\neg$, $\vee$ i $\wedge$ nad logičkim vrednostima p, q (logičkim promenljivim, konstantama, izrazima), definisane su sledećim tablicama istinitosnih vrednosti:

p	q	$\neg\ p$	$p \wedge q$	$p \vee q$
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F

U logičkom izrazu $p \wedge q$, p i q nazivaju se **konjunktima**. Slično, u logičkom izrazu $p \vee q$, p i q nazivaju se **disjunktima**.

Logički izraz se naziva i **uslov** ili **predikat**. Ako je vrednost logičkog izraza T, kaže se da je odgovarajući uslov zadovoljen, tj. predikat tačan, inače uslov nije zadovoljen, tj. predikat je netačan.

Operacija $\neg$ je najvišeg prioriteta, zatim operacija $\wedge$, a najnižeg prioriteta je operacija $\vee$. Primenom ove tri logičke operacije mogu se izraziti i sve ostale binarne logičke operacije (ima ih 16 jer ima 4 različita para logičkih vrednosti operanada p i q, pa dakle ima 2^4 različitih rasporeda vrednosti T, F na 4 mesta). Na primer, implikacija $p \Rightarrow q$ može se zapisati kao $\neg\ p \vee q$, ekvivalencija $p \Leftrightarrow q$ može se zapisati kao $(\neg\ p \vee q) \wedge (\neg\ q \vee p)$, što se može prikazati sledećom tablicom:

p	q	$p \implies q\ (\neg p \vee q)$	$p \equiv q\ ((\neg p \vee q) \wedge (\neg q \vee p))$
T	T	T	T
T	F	F	F
F	T	T	F
F	F	T	T

Kako i svaka od relacijskih operacija ($\{=, <, >, <>, <=, >=\}$) proizvodi logičku vrednost kao rezultat i kada se primeni na logičke argumente (jer postoji uređenje $F < T$, tj. $0 < 1$), binarne logičke operacije mogu se izraziti i pomoću relacijskih operacija nad logičkim vrednostima. Tako se implikacija može izraziti kao $p <= q$, ekvivalencija kao $p = q$, itd.

5.1.2 Formalna specifikacija

Formalna specifikacija problema je zapis oblika

$\{Q\}S\{R\}$,

gde su Q, R logički izrazi, tvrđenja čija je vrednost ili tačno (T) ili netačno (F), a S je algoritam (tj. program).

Interpretacija ovog zapisa specifikacije programa je sledeća:

"Ako izvršenje S počinje sa vrednostima ulaznih promenljivih ("u stanju") koje zadovoljavaju uslov Q, onda se garantuje da će se S završiti u konačnom vremenu sa vrednostima programskih promenljivih ("u stanju") koje zadovoljavaju uslov R".

Uslov Q naziva se **preduslov** za algoritam (program, iskaz) S, a uslov R naziva se **postuslov** za algoritam (program, iskaz) S. Preduslov i postuslov precizno definišu sve ulazne i izlazne promenljive, pa se formalna specifikacija može predstaviti kao transformacija preduslova u postuslov, tj.

{ preduslov} $\rightarrow$ **{postuslov}**

Primer (swap funkcija – razmena vrednosti dveju promenljivih x, y)

$\{x = X \wedge y = Y\} \rightarrow \{x = Y \wedge y = X\}$

(ovde su oznake x, y upotrebljene da označe promenljive, a oznake X, Y – proizvoljne ali fiksirane vrednosti tih promenljivih).

5.2 Algoritam – intuitivni pojam

Pojam algoritma pripada osnovnim pojmovima matematike. Smisao reči (ne definicija pojma) je da je algoritam tačan **propis** o izvršenju, **određenim redosledom**, nekog **niza operacija** za rešavanje svih zadataka zadatog tipa. Propis u ovoj formulaciji predstavlja konačni skup pravila, tj. opis postupka kojim se izvršavaju operacije. Pojedina operacija algoritma naziva se **algoritamski korak**, a određeni redosled podrazumeva preciznu informaciju o tome koji je algoritamski korak prvi, koji je poslednji (kojim se algoritam završava), i koji algoritamski korak sledi za posmatranim.

Redosled izvršavanje algoritamskih koraka pri jednom izvršavanju algoritma određuje njegovu **algoritamsku strukturu**. Algoritamska struktura može biti **linijska**, kada se posle jednog algoritamskog koraka može izvršavati samo algoritamski korak koji se još nije izvršavao pri tom izvršavanju algoritma, i **ciklična**, kada se posle jednog algoritamskog koraka može izvršavati i algoritamski korak koji se već izvršavao pri tom izvršavanju algoritma. U linijskoj algoritamkoj strukturi, dakle, svaki algoritamski korak izvršava se najviše jedanput pri jednom izvršavanju

algoritma, dok se kod ciklične algoritamske strukture jedan algoritamski korak može izvršavati i više puta pri jednom izvršavanju algoritma.

Iako neformalna, prethodna formulacija odražava pojam algoritma koji se stihijski oblikovao tokom vekova.

Navedimo ovde i definiciju algoritma prema Webster's Ninth New Collegiate rečniku: algoritam je procedura za rešavanje nekog matematičkog problema (kao što je nalaženje najvećeg zajedničkog delioca) u konačnom broju koraka, koja često uključuje ponavljanje neke od operacija; ili šire, korak po korak određena procedura za rešavanje problema ili dolaženje do nekog cilja.

Sama reč **algoritam** dolazi od imena srednjevekovnog arapskog matematičara Al Horezmi, koji je još u IX veku formulisao pravila po kojima se izvršavaju četiri aritmetičke operacije u dekadnom sistemu. To su prvi i najjednostavniji algoritmi.

U razumevanju pojma algoritma potrebno je jasno razlikovati konkretni (pojedinačni) zadatak od skupa svih zadataka jednog, istog tipa. Na primer, traženje rešenja algebarske jednačine $x^2 - 4 = 0$ je konkretni zadatak čije je rešenje skup brojeva $\{-2, 2\}$, kao što je i sabiranje dva zadata cela dekadna broja, npr. 1256 i 157989, konkretni zadatak sa konkretnim rešenjem 159245. S druge strane, skup svih zadataka istog tipa formuliše se kao **problem**. Na primer, naći postupak koji omogućuje rešavanje proizvoljne kvadratne jednačine, ili izračunavanje zbira bilo koja dva cela broja u dekadnom sistemu. Za problem je karakteristično prisustvo parametara (promenljivih) kojima se postiže konačna formulacija moguće beskonačnog skupa zadataka posmatranog tipa. Zato je rešenje problema jedan zajednički propis (metoda, algoritam) koji omogućuje rešavanje bilo kog konkretnog zadatka datog problema, čim su poznate konkretne vrednosti promenljivih za taj zadatak. Konkretne vrednosti promenljivih nazivaju se ulaznim podacima algoritma, dok se rešenje konkretnog zadatka za zadate ulazne podatke naziva rezultatom rada algoritma.

Za rešavanje jednog problema moguće je, u opštem slučaju, primeniti veći broj različitih metoda, i pritom doći do različitih algoritama, različitih po svojoj složenosti, ali i po svojoj efikasnosti. Izgradnja algoritma, kada je moguća, povezana je, u opštem slučaju, sa suptilnim i složenim rasuđivanjem koje zahteva kreativnost, visoku stručnost i veliku dovitljivost. Osim što treba da bude korektan (da rešava problem za koji je izgrađen), dobar algoritam treba da bude i efikasan, tj. da izvršavanjem što manjeg broja, što jednostavnijih operacija za dati ulaz proizvede rešenje zadatka. Zato se proces izgradnje algoritma može podeliti u četiri faze: konstrukcija, dokaz korektnosti, analiza efikasnosti, implementacija (u programskom jeziku). Problemi koji mogu da nastanu u svakoj od ovih faza utiču na modifikaciju i ponovno izvršenje prethodnih faza. Konstrukcija i dokaz korektnosti mogu se objediniti u proces **izvođenja algoritma** iz formalne specifikacije, pri kojem se izgrađuje algoritam koji garantovano zadovoljava specifikaciju, tj. garantovano korektan algoritam.

S druge strane, izvršavanje algoritma, tj. proces rešavanja pripadnih konkretnih zadataka, od trenutka kada je algoritam već izgrađen, potpuno je "automatski", u smislu da se od onoga ko tu radnju izvodi (čovek ili mašina) zahteva samo da je

u stanju da obavlja **izračunavanje** i **upravljanje**, tj. da izvršava elementarne operacije od kojih se proces sastoji, i da se tačno pridržava postavljenog propisa (algoritma).

Procesi koji teku po strogo određenom algoritmu susreću se pre svega u matematici, ali i u drugim područjima ljudske delatnosti.

Algoritmi se mogu zapisati u različitim notacijama – prirodnim jezikom, dijagramima toka, pseudojezikom, programskim jezikom, itd.

U sledećoj tački biće predstavljen jedan primer algoritma za koji se dokazuje da odgovara zadatoj specifikaciji. Ovaj primer će da ilustruju tehniku za dokazivanje korektnosti.

5.2.1 Algoritam razmene – swap

Reč je o razmeni vrednosti dve promenljive, x, y. Ako se sa X, Y označe proizvoljne ali fiksirane vrednosti promenljivih x, y, redom (npr. X je 3, Y je 5), onda je specifikacij algoritma swap

$\{x = X \wedge y = Y\} \rightarrow \{x = Y \wedge y = X\}$

Postoji veći broj algoritama razmene koji zadovoljavaju ovu specifikaciju. Na primer, za swap algoritam koji uključuje treću promenljivu, t, sledećeg oblika:

swap: $t \leftarrow x; x \leftarrow y; y \leftarrow t$, važi

$\{x = X \wedge y = Y\}$ swap $\{x = Y \wedge y = X\}$.

Interpretacija ovog zapisa je da

$\forall(X, Y, x, y)\ \{x = X \wedge y = Y\}$ swap $\{x = Y \wedge y = X\}$.

Tačnost ovog tvrđenja sledi iz sledećeg razmatranja:

$\{x = X \wedge y = Y\}\ t \leftarrow x\ \{x = X \wedge y = Y \wedge t = X\}$

$\{x = X \wedge y = Y \wedge t = X\}\ x \leftarrow y\ \{x = Y \wedge y = Y \wedge t = X\}$

$\{x = Y \wedge y = Y \wedge t = X\}\ y \leftarrow t\ \{x = Y \wedge y = X \wedge t = X\}$

Drugi swap algoritam (bez treće promenljive) ima sledeće korake:

swap: $x \leftarrow x + y; y \leftarrow x - y; x \leftarrow x - y$, i opet važi

$\{x = X \wedge y = Y\}$ swap $\{x = Y \wedge y = X\}$.

Ovo sledi iz sledećeg razmatranja:

$\{x = X \wedge y = Y\}\ x \leftarrow x + y\ \{x = X + Y \wedge y = Y\}$

$\{x = X + Y \wedge y = Y\}\ y \leftarrow x - y\ \{x = X + Y \wedge y = X + Y - Y = X\}$

$\{x = X + Y \wedge y = X\}\ x \leftarrow x - y\ \{x = X + Y - X = Y \wedge y = X\}$

6

Pregled programskog jezika C

U ovoj tački biće izložen kratki pregled programskog jezika C kroz prikaz njegovih osnovnih karakteristika.

6.1 Filozofija jezika

Programski jezik C je programski jezik opšte namene koji potiče od jezika BCPL i B, koji su korišćeni za pisanje operativnih sistema i jezičkih procesora. Osnovni doprinos C-a, u odnosu na ove jezike, jesu tipizirane promenljive.

Programski jezik C implementirao je 1972. godine Denis Riči (Dennis Ritchie, SAD), kao jezik za programiranje aplikacija pod UNIX operativnim sistemom na DEC PDP-11 seriji računara. Godine 1977. izgrađena je verzija programskog jezika C (Portable C compiler) nezavisna od vrste računara, koja je omogućila prenosivost programa napisanih u C-u. Neformalna specifikacija jezika prikazana je u knjizi Brajana Kernigena i Denisa Ričija "The C Programming Language", 1978. godine. Prvi formalni standardi jezika pojavili su se 80-tih godina (američki standard 1989. – ANSI standard, međunarodni ISO standard 1990.). ISO standardi iz 1999. i 2011. proširili su jezik konstrukcijama novijih jezika. Verzija C jezika koja se ovde opisuje uglavnom je bazirana na specifikacija jezika iz Kernigen–Riči knjige.

S obzirom na svoju filozofiju i namenu, programski jezik C blizak je operacijama mašinskog nivoa, omogućuje rad sa pojedinačnim bitovima i memorijskim adresama, fleksibilan je. Ova fleksibilnost i odsustvo ograničenja čini ga primenljivim u raznovrsnim domenima programiranja. Zbog odsustva ograničenja i malog broja tipova, programi napisani u programskom jeziku C mogu da budu manje pouzdani i teže čitljivi. Jezik ne uključuje svojstva kao što su ulaz/izlaz, matematičke (elementarne) funkcije – sin, cos, exp, itd., ali se ove komponente "ugrađuju" u C iz odgovarajućih biblioteka.

6.2 Struktura programa

Sledeća svojstva karakterišu strukturu programa na programskom jeziku C:

1. Program može da se uključi veći broj funkcija koje se nalaze u većem broju izvornih datoteka.

2. Jedna izvorna datoteka sastoji se od spoljašnjih (globalnih) objekata – spoljašnjih podataka (to su podaci koji su definisani izvan funkcija) i funkcija (u C-u su sve funkcije spoljašnje, tj. ne mogu da budu definisane unutar druge funkcije). Spoljašnji objekti mogu da budu vezani za tu datoteku (to su statički –"static" objekti) ili za sve datoteke tog programa. Jedan spoljašnji objekat dostupan je svim funkcijama u istoj izvornoj datoteci koje slede za definicijom tog objekta. Pristup spoljašnjim objektima definisanim u jednoj izvornoj datoteci, od strane funkcija iz druge datoteke, postiže se navođenjem *deklaracije* "extern" koja samo navodi ime i tip takvog objekta ali ne rezerviše memorijski prostor.

3. Svi izvršni iskazi sadržani su u *blokovima* koji mogu da budu ugnježdeni do proizvoljne dubine. Promenljive mogu da budu deklarisane na početku bilo kog bloka i vide se samo u tom bloku (to su untrašnje ili lokalne promenljive). Svaki iskaz završava se tačka-zapetom (**;**), a blokovi su uokvireni zagradama ({, }).

4. Većina izvornih datoteka C-programa počinje uvodnim delom sa tzv. *preprocesorskim* instrukcijama. Preprocesorska instrukcija počinje znakom #.

5. Po konvenciji, svaki program ima funkciju *main* kojom počinje izvršavanje.

6. Sve C-ovske rutine (potprogrami) su funkcije. Svaka funkcija može da vraća vrednost, koja može biti bilo kog tipa osim nizovskog. C-ovske funkcije navode se jedna za drugom unutar jedne izvorne datoteke.

7. Parametri funkcija prenose se po vrednosti (funkcija ne može da promeni vrednost svog parametra), osim za nizove. Vrednost nizovskog parametra je *adresa* prvog elementa niza.

8. Izvršavanje funkcije završava se izvršenjem poslednjeg iskaza funkcije, ili posle izvršenja *return* iskaza. Funkcija ne mora da izračunava (i vraća) vrednost, a ako to čini, onda se iskaz *return* koristi i za vraćanje vrednosti funkcije.

9. Za razliku od "return" iskaza kojim se završava izvršavanje funkcije i program nastavlja sa radom, za prekid rada po programu (zaustavljanje programa) može da se koristi poziv funkcije "exit(status)" koja zaustavlja program i vraća celobrojnu vrednost status (obično 0 za uspešno izvršen program).

10. Komentari se pišu između para oznaka "/*", "*/" i ne utiču na izvršavanje programa.

6.3 Primer programa

Razmotrimo prvo najjednostavniji program u programskom jeziku C koji na standarni izlaz ispisuje poruku "Ovo je moj prvi program".

```
#include <stdio.h>
int main()
{
    printf("Ovo je moj prvi program\n");
    return 0;
}
```

Prva linija počinje znakom # i to je preprocesorska instrukcija kojom se uključuje standardna datoteka stdio.h u kojoj se nalaze prototipovi – deklaracije svih funkcija za ulaz i izlaz podataka – u našem primeru potrebna je zbog funkcije printf koja se poziva u programu. Program ima jednu funkciju – glavnu funkciju – main, od koje počinje izvršavanje programa. Svi njeni iskazi navedeni su između para zagrada { i } – to su samo poziv funkcije printf za ispisivanje poruke *Ovo je moj prvi program*, i iskaz return kojim se vrši povratak iz funkcije main u okruženje programskog jezika C sa vrednosšću 0 koja ukazuje na uspešno izvršenje programa.

Da bismo ilustrovali kompletniju strukturu programa na programskom jeziku C razmotrimo sada jedan složeniji primer. Pre svega, razmotrimo primer funkcije za tzv. binarno pretraživanje niza. Neka su elementi niza – slogovi sa dva polja – celobrojnim poljem "ključ" i poljem tipa niske karaktera – "vrednost". Neka su slogovi sortirani po polju "ključ" (ključnom polju). Funkcija uzima taj niz slogova i traži slog sa zadatom vrednošću ključnog polja; vraća indeks sloga čija je vrednost ključnog polja jednaka traženoj, ako takav postoji, i indikator neuspeha, u suprotnom. Ako je ime niza A a vrednost ključa koja se traži smeštena u promenljivoj k, algoritam za ovu funkciju ima sledeći oblik:

1. Postaviti l na najmanji indeks a d na najveći indeks niza A.
2. Dok je $l \leq d$, ponoviti korake 3–5. Ako je $l > d$, vratiti "neuspeh".
3. Postaviti s na indeks na sredini između l i d.
4. Ako je ključ sloga A_s jednak k, vratiti indeks s.
5. Ako je ključ sloga A_s manji od k, preći na desnu polovinu niza, tj. postaviti l na $s + 1$, inače preći na levu polovinu niza tj. postaviti d na $s - 1$.

Ovaj algoritam se u C-u može zapisati funkcijom BinarPret koja ima tri argumenta – niz slogova A, broj slogova u tom nizu n i ključ k koji se traži. Funkcija vraća celobrojnu vrednost – indeks sloga sa ključem k, ako takav postoji, ili -1 ako takav slog u nizu A ne postoji. Program ima sledeći oblik:

```
#include <stdio.h>
```

```
#define NEUSPEH −1
#define MAXELEM 100

typedef struct {
      int kljuc;
      char *vrednost;
} slog;
slog A[MAXELEM];
int BinarPret(slog A[], int n, int k);

main()
{
      int i,n,x;

      printf("unesi broj elemenata i celobrojne elemente niza u rastucem
poretku\n");
      scanf("%d", &n);
      for(i=0;i<n; )
            scanf("%d", &(A[i++].kljuc));
      printf("unesi celobrojnu vrednost kljuca koji se trazi\n");
      scanf("%d", &x);
      printf("indeks trazenog elementa je %d", BinarPret(A,n,x));
}

int BinarPret(slog A[ ], int n, int k)
/* vraca indeks elementa niza A sa kljucem k,
            vraca NEUSPEH ako k nije u A */
{
      int l=0, d=n-1,s;
      while(l <= d) {
            s = (l + d)/2;
            if(k < A[s].kljuc) d=s-1;
            else if(k > A[s].kljuc) l=s+1;
            else return(s);
      }
      return(NEUSPEH);
}
```

Struktura tipa slog ima polja kljuc i vrednost, pri čemu se polje kljuc koristi za poređenje, a polje vrednost se i ne koristi – ono je prisutno kao predstavnik drugih informacija koje se pamte o pojedinom slogu.

6.4 Preprocesor C-a

I u ovom primeru program počinje instrukcijama preprocesoru – tzv. preprocesorskim direktivama tj. linijama kojima prethodi znak #. Rad preprocesora prethodi prevođenju programa sa C-a na mašinski jezik.

Postoje tri vrste preprocesorskih direktiva (sve počinju znakom #): instrukcija #define kojom se vrši zamena stringova (u našem primeru string "NEUSPEH" zamenjuje se, gdegod se pojavi u programu, stringom "−1"), instrukcija #include kojom se uključuje tekst, obično tekstuelna datoteka koja sadrži standardni skup deklaracija, tzv. standardno zaglavlje (u našem primeru na početak programa uključuje se tekstuelna datoteka stdio.h), i instrukcije za uslovnu kompilaciju (više u poglavlju 10 o funkcijama).

Direktivom #define zadaje se tzv. "makro" identifikator (prvi string) koji može da ima dva oblika: bez parametara, kada se zadaje zamena tipa objekta, i sa parametrima, kada se zadaje zamena tipa funkcije. U našem primeru imali smo zamenu tipa objekta.

6.5 Tipovi podataka

Deklaracija promenljivih u C-u ima oblik

vrsta *tip* lista-promenljivih

gde je *vrsta* – oznaka vidljivosti i trajanja promenljivih, memorijskog prostora koji promenljiva zauzima (auto, static, extern, register, typedef), *tip* – identifikator tipa (predefinisanog, korisnički definisanog ili strukturnog), a lista promenljivih – spisak imena promenljivih, razdvojenih zapetama, koje imaju istu vrstu i tip.

Na primer,

int v[50];

deklariše 50-elementni celobrojni niz v (vrsta nije navedena i zavisi od mesta gde je niz definisan).

Deklaracija promenljivih može da bude ujedno i definicija (kada se za promenljivu rezerviše memorijski prostor) – takve su sve promenljive u našem primeru.

Korisnički tip može se uvesti definicijom tipa oblika

typedef *tip* id-tipa;

gde je *tip* – neki već prethodno definisani tip, a id-tipa – novi identifikator (novo ime) tog tipa.
Na primer

typedef int[50] vector;

(uvodi novo ime "vector" za tip 50-elementnog celobrojnog niza; niz je predefinisani tip u C-u).

Niz v sada može da se deklariše i kao
vector v;

U našem primeru, definiše se tip strukture i dodeljuje mu se ime slog.

Oblast važenja promenljive (doseg, gde se ona "vidi", gde joj se može pristupiti radi čitanja ili upisa vrednosti) zavisi od toga gde je deklarisana i koje je *vrste*, i

može biti blok, funkcija, izvorna datoteka, ili sve izvorne datoteke jednog programa. *Trajanje* (ili životni vek) promenljive je nezavisno od njene oblasti važenja, zavisi od njene *vrste*, i može biti trajanje izvršavanja bloka, funkcije ili programa. Na primer, promenljiva lokalna za blok, sa trajanjem izvršenja bloka, zove se *automatska* (auto) promenljiva, dok je promenljiva lokalna za blok sa trajanjem izvršenja celog programa – *statička* (static) promenljiva.

U našem primeru, niz A sa elementima tipa slog je spoljašnji (globlni) objekat koji se "vidi" i u funkciji main i u funkciji BinarPret, i čije je trajanje – trajanje celog programa; celobrojne promenljive i,n,x su automatske promenljive (unutrašnje za funkciju main sa trajanjem te funkcije), dok su promenljive l,d,s – automatske promenljive funkcije BinarPret.

Postoji i vrsta automatske promenljive koja se zove *registarska* (register) promenljiva za koju se sugeriše kompilatoru da je smesti u procesorski registar kome se mnogo brže pristupa nego memorijskom registru. Kao registarske deklarišu se promenljive koje se veoma često koriste u programu.

Promenljive se mogu inicijalizovati definicijama (u našem primeru int l=0).

Osnovni tipovi. U C-u postoje tri osnovna tipa podataka: znakovni (char), celobrojni i realni. Celobrojni tip se javlja u tri veličine (int, short, long), realni u dve (float, double). Celi brojevi mogu biti i neoznačeni (*unsigned*) za manipulisanje bitovima i adresama.

Nabrojivi tipovi. U C-u postoje i nabrojivi tipovi koji se sastoje od nabrojivih konstanti sa celobrojnim vrednostima (od 0 do broja konstanti−1, što se podrazumeva, ili sa vrednostima koje su tim konstantama eksplicitno dodeljene). Na primer,

enum boolean {NE, DA};

je nabrojivi tip nazvan boolean sa dve nabrojive konstante od kojih prva (NE) ima vrednost 0 a druga (DA) vrednost 1. Ali, u primeru

enum meseci { JAN=1, FEB, MAR, APR, MAJ, JUN, JUL, AVG, SEP, OKT, NOV, DEC };

nabrojiva konstanta JAN ima vrednost 1 a svaka sledeća – za 1 veću od prethodne.

Nabrojivi tip je alternativni vid definisanja imenovanih konstanti (pored #define).

Strukturni tipovi. U C-u su prisutna dva mehanizma za struktuiranje podataka: *niz* i *struktura*. Deklaracija niza ima oblik

[*vrsta*] *tip_elementa ime* [*veličina*];

Vrsta može imati iste vrednosti kao i za druge promenljive (auto, static, extern, register, typedef), i najčešće se podrazumeva i ispušta, *tip_elementa* može biti bilo koji tip, ali *veličina* mora biti konstantni celobrojni izraz. Elementi niza *ime* imaju

indekse od 0 do *veličina* -1 (u našem primeru niz A ima elemente tipa slog a indekse od 0 do MAXELEM -1, tj. 99). U C-u parametar funkcije može biti niz sa proizvoljnim brojem elemenata (npr. niz A kao parametar funkcije BinarPret).

Struktura u C-u je objekat koji se sastoji od jednog ili više polja (tzv. *članovi* strukture), gde je svako polje – objekat proizvoljnog tipa. U našem primeru, struktura tipa slog sastoji se od dva polja – jedno je kljuc i celobrojnog je tipa, a drugo je vrednost i tip mu je pokazivač na objekte tipa karakter (zbog tesne veze između pokazivača i nizova u C-u, "vrednost" je zapravo ime niza karaktera, tj. adresa prvog elementa niza karaktera – v. sledeći paragraf "Pokazivači"). Obraćanje polju strukture vrši se navođenjem imena strukture, tačka, ime polja – u našem primeru, A[s].kljuc.

Dve varijante tipa strukture u C-u su unija i bit-polja (o ovim varijantama biće reči kasnije).

Pokazivači. Pokazivač u C-u pokazuje na objekat nekog specifičnog (*baznog*) tipa, tj. predstavlja memorijsku adresu tog objekta. Deklaracija

char ∗vrednost;
slog ∗sp;

deklariše *vrednost* kao pokazivač na objekte tipa karakter a *sp* kao pokazivač na objekte tipa slog. Operator ∗ (tzv. operator dereferenciranja, primenjuje se na objekat tipa pokazivač) daje kao rezultat vrednost promenljive na koju pokazuje pokazivač – argument ovog operatora, tj. promenljivu čija je adresa – argument ovog operatora. Tako, *vrednost* i *sp* su adrese, ∗*vrednost* je karakter, ∗sp je struktura tip slog a (∗sp).kljuc (ili, ekvivalentno, $sp \rightarrow kljuc$) je ceo broj. Operator & (tzv. operator referenciranja, primenjuje se na objekat – promenljivu proizvoljnog tipa) inverzan je operatoru ∗ i daje rezultat – adresu objekta. Tako, ako je promenljiva *primerak_sloga* tipa slog (slog *primerak_sloga*), onda *sp* može da bude inicijalizovano sa

$sp = \&primerak_sloga;$

Nad adresama (pokazivačima) definisani su operatori poređenja, oduzimanja, dodavanja i oduzimanja celih brojeva na pokazivač (od pokazivača). Operacija sabiranja pokazivača nije dopuštena. Račun nad adresama izvodi se u jedinicama veličine memorije potrebne za smeštanje vrednosti baznog tipa za pokazivač. Zato, $vrednost + 1$ pokazuje na sledeći karakter u memoriji a $sp + 1$ pokazuje na sledeći primerak sloga.

Pokazivači i nizovi su tesno povezani u C-u. Ime niza je pokazivač na prvi element niza. Tako, ako je A niz, njegovom prvom elementu može se pristupiti bilo sa $A[0]$ ili $*A$. Slično, $vrednost[0]$ označava karakter na koji pokazuje $vrednost$, $vrednost[-1]$ je prethodni karakter a $vrednost[1]$ je sledeći karakter.

U C-u pokazivači mogu da pristupe eksplicitno deklarisanim promenljivim (u našem primeru, npr. &n).

Komentar o tipovima. Filozofija tipova podataka u C-u je obezbeđenje pogodnog načina da se upravlja hardverom. Mada je nivo instrukcija u C-u obično viši od mašinskog, objekti kojima operiše su nivoa sličnog mašinskom. U C-u se može jednostavno pristupiti specifičnim fizičkim adresama, kao i specifičnim bitovima u reči.

Zbog mešanja buleanskog i celobrojnog tipa (svaka celobrojna vrednost $\neq 0$ u C-u se može interpretirati kao TRUE, a 0 – kao FALSE; logički izraz dobija vrednost 1 ako je tačan, i 0 ako je netačan), C kompilatori ne mogu da otkriju izvesne greške. Na primer, u C-u su sintaksno korektni relacijski izrazi

$1<2<3$ i $3<2<1$

i oba se izračunavaju na 1 (TRUE). Naime, izračunavanje izraza ide sleva na desno. U prvom izrazu, $1<2$ daje tačno (1), a $1<3$ daje opet tačno (1). U drugom izrazu, $3<2$ daje vrednost netačno (0), a $0<1$ daje opet tačno (1).

6.6 Ključne reči

C jezik sadrži i rezervisane (ključne) reči koje ne mogu da se koriste kao identifikatori. Ključne reči se odonose na nazive tipova, vrste (promenljivih) i iskaza. To su: auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while. Ključne reči ne mogu da se koriste u druge svrhe, npr. kao identifikatori (imena) objekata – promenljivih, konstanti, funkcija i sl. Ključne reči se pišu malim slovima. S obzirom da se u C-u razlikuju mala od velikih slova, AUTO, FOR, INT i sl. nisu ključne reči i mogu da se koriste kao identifikatori objekata. Slično važi i za Auto, FoR, iNT i sl.

6.7 Izrazi

Izrazi se grade od konstanti, promenljivih, poziva funkcija i operacija. Programski jezik C uključuje (između ostalih) sledeće skupove operacija:

1. aritmetičke (+,−,∗, /, % (ostatak pri celobrojnom deljenju – moduo); deljenje (/) nad celim brojevima daje celobrojni deo količnika)
2. logičke (&&, ||, !)
3. relacijske (==, ! =, <, <=, >, >=)
4. manipulisanje bitovima (&, |, ˆ, . . .)
5. uvećanja i umanjenja za jedan (++, −−)
6. promena tipa ((*tip*) *izraz*)
7. dodele (=, +=, − =, ∗=, /=, %=, . . .)

U C-u postoji veći broj prioriteta operacija (npr. operacije poređenja na jednakost i različitost, == i ! =, su nižeg prioriteta od ostalih relacijskih operacija, a sve one su višeg prioriteta od logičke operacije konjunkcije && koja je višeg prioriteta u odnosu na disjunkciju, ||).

Operatori dodele. Za C je karakteristično da operator dodele (=) može da učestvuje u izrazu kao i svaki drugi operator. Ovim operatorom se izračunava vrednost izraza na desnoj strani ovog operatora, konvertuje se u tip promenljive na levoj strani i dodeljuje toj promenljivoj. Pri tome i ceo izraz dodele dobija vrednost pridruženu promenljivoj na levoj strani. Na primer, izraz (x=sledeci())==KRAJ dobija vrednost 1 (tačno) ili 0 (netačno), jer funkcija sledeci() svoju vrednost dodeljuje promenljivoj x, i pri tome izraz x=sledeci() dobija vrednost promenljive x; zato se ta vrednost može dalje porediti sa konstantom KRAJ, a vrednost tog poređenja je 1 (za tačno) ili 0 (za netačno). Na levoj strani operatora dodele mora biti tzv. l-vrednost (l-value), a to je ime promenljive, element niza ili memorijska adresa. Dodela vrednosti promenljivoj na levoj strani operatora dodele zapravo je bočni efekat operatora dodele, dok je njegov glavni efekat – značenje – izračunavanje vrednosti izraza.

Pored uobičajenog (osnovnog) operatora dodele, C ima i operatore dodele sa dodavanjem, oduzimanjem, množenjem, deljenjem, i sl, tj. +=, −=, ∗=, /=, %=, . . . , kao i operatore uvećanja odnosno umanjenja za 1, ++, −−. Na primer, v+=a+5; je ekvivalentno sa v=v+a+5; ++v je uvećanje vrednosti promenljive v pre korišćenja njene vrednosti, a v++ uvećanje vrednosti te promenljive *posle* korišćenja njene vrednosti.

Dopuštanjem dodela unutar izraza programski jezik C doprinosi bočnim efektima, koji čine program manje preglednim, manje pouzdanim i teže izmenljivim. Sledeći iskaz u C-u ilustruje ova svojstva:

while((s[i++]=getchar())! =EOF);

Logički izraz. Logički izrazi izračunavaju se uslovno, sleva nadesno, samo dok se ne utvrdi njihova istinitosna vrednost; to je tzv. *lenjo izračunavanje* logičkog izraza. Tako, desni operand operacija AND (&&), OR (||) izračunava se samo ako vrednost izraza nije utvrđena izračunavanjem levog operanda (u suprotnom ne mora da bude čak ni definisan, npr. može da sadrži deljenje nulom).

Uslovni operator. Ovaj operator, u oznaci ?:, uzima tri operanda: uslov, tačno-deo, netačno-deo. Vrdenost izraza jednaka je tačno-deo ako je uslov tačan (ima vrednost $\neq 0$), u suprotnom je vrednost izraza netačno-deo. Na primer, vrednost izraza

y ! = 0 ? 1/y : BESKONACNO

jeste 1/y ako y nije 0, i BESKONACNO, inače.

Operator ”,”. Operator ”,” omogućuje kombinovanje dva izraza u jedan. Vrednost rezultujućeg izraza je vrednost desnog izraza. Na primer, izraz

i=0, j=1

dobija vrednost 1, dok kao bočni efekat dodeljuje 0 promenljivoj i a 1 promenljivoj j.

6.8 Iskazi

U programskom jeziku C, svaki izraz može biti ”pretvoren” u iskaz ako mu se na kraj doda tačka-zapeta (”;”); izraz se tada izračunava da bi se izvršili njegovi bočni efekti, i vrednost koju izračunava se ne koristi.

Dodela. Serijska kompozicija. Iskaz dodele u C-u je izraz dodele sa tačka-zapetom na kraju. Dodela se može izvršiti između svih tipova osim nizovskog. Višestruke dodele imaju smisla jer je dodela – izraz. Na primer,

i=j=k=0

jeste izraz (a i=j=k=0; iskaz) koji se izračunava zdesna ulevo tako što promenljiva k dobija vrednost 0, podizraz $k = 0$ dobija vrednost 0, ta vrednost se dodeljuje promenljivoj j, podizraz $j = k = 0$ dobija vrednost 0, a ta vrednost se dodeljuje promenljivoj i i ceo izraz dobija vrednost promenljive i (0).

Niz iskaza grupiše se u *blok* – jedan serijski komponovani iskaz uokviravanjem zagradama ({, }).

Selekcija. Osnovna konstrukcija za selekciju (izbor) iskaza u C-u je oblika

if(uslov) *iskaz1*
else *iskaz2*

Uslov je izraz čija je istinitosna vrednost netačno ako je 0, a inače je tačno. Semantika iskaza je ista kao i u drugim programskim jezicima: ako je vrednost uslova $\neq 0$, izvršava se iskaz1, inače se izvršava iskaz2 .

Za višestruki izbor prema vrednosti izraza programski jezik C ima iskaz *switch*:

switch (*izraz*) {
case *konstantni_izraz*1: $lista_iskaza_1$;
case *konstantni_izraz*2: $lista_iskaza_2$;
⋮
default: *podrazumevana_lista_iskaza*;
⋮
}

Posle izvršenja liste iskaza pridruženih vrednosti izraza, izvršavanje se nastavlja izvšavanjem svih lista iskaza koje slede, osim ako se kontrola ne prenese eksplicitno (iskazom *break*) na drugi deo programa.

Iteracija (ponavljanje). Programski jezik C ima iterativne konstrukcije (petlje) sa proverom uslova na početku ili na kraju:

while (*uslov*) *iskaz*

odnosno

do *iskaz* while (*uslov*)

U oba slučaja iskaz se izvršava sve dok se logički izraz *uslov* izračunava na vrednost tačno.

Ciklus oblika
inicijalizacija_petlje;
while (uslov) {
    iskaz;
    reinicijalizacija_petlje;
}

zapisuje se u C-u sintaksom koja objedinjuje inicijalizacije i uslov u zaglavlje petlje:

for(*inicijalizacija_petlje; uslov; reinicijalizacija_petlje) iskaz*;

Komponente zaglavlja for iskaza su izrazi, i svaki može da bude prazan (prazan uslov ima vrednost tačno). Na primer, for(i=0; i<n; i++)A[i]=0; For-iskaz u C-u ne mora garantovano da se završava, smer iteracije ne mora da bude određen iz samog teksta programa, a broj iteracija ne mora da bude poznat na početku izvršavanja iskaza; s druge strane, on ima veliku fleksibilnost (npr. vrednost kontrolne promenljive ne mora da se menja za $(+|-)1$).

Bezuslovni prenos kontrole. U C-u postoje tri iskaza bezuslovnog prenosa kontrole upravljanja: *break*, *continue* i *goto*. *Break* izlazi iz unutrašnje petlje ili *switch* iskaza. *Continue* prelazi na sledeću iteraciju unutrašnje petlje, dok *goto* prenosi upravljanje na obeležje (koje ima oblik identifikatora). Prenos je moguć između blokova i upravljačkih struktura (selekcije, iteracije), ali ne između funkcija.

6.9 Ulaz-izlaz

Operacije ulaza i izlaza nisu deo programskog jezika C, već se realizuju funkcijama iz standardne C biblioteke (pripadaju tzv. okruženju C-jezika). Standardno zaglavlje stdio.h uključuje funkcije za unošenje i izdavanje podataka.

Najjednostavniji mehanizam za unošenje jeste čitanje jednog po jednog karaktera sa *standardnog ulaza*, obično tastature, funkcijom getchar bez argumenata (void označava odsustvo argumenata) koja vraća celobrojnu vrednost (int). Prototip ove funkcije je

int getchar(void);

Ona vraća sledeći učitani karakter ili EOF (-1) kada pročita karakter za kraj ulaza (CTRL-Z pod Windows) ili "naiđe" na kraj datoteke (ako je ulaz iz datoteke).

Slično, funkcija

int putchar(int)

koristi se za izdavanje: putchar(c) prikazuje karakter c na *standardnom izlazu*, obično ekranu.

Mehanizam formatiranja se koristi da konvertuje vrednost proizvoljnog tipa iz memorije u string proizvoljne forme na izlazu (pri izdavanju podataka – npr. realni broj u string sa tri decimalna mesta), ili string u odgovarjuću vrednost u memoriji (pri učitavanju podataka).

Funkcija izlaza, printf, prevodi interne vrednosti (iz memorije) u **niske karaktera** (stringove). Ona ima promenljivi broj argumenata i zaglavlje oblika

int printf(char *format, arg_1, arg_2, ...)

Ova funkcija konvertuje, formatira i izdaje vrednosti svojih argumenata arg_1, arg_2, ... na standardni izlaz, pod kontrolom zadatog formata. Funkcija vraća broj izdatih karaktera.

Pri pozivu funkcije printf, argument format (s obzirom da je string) piše se između navodnika. Pored običnih karaktera koji se izdaju kako su napisani, format uključuje i objekte za specifikaciju konverzije, tj. objekte koji govore kojim tipom interpretirati i u koji oblik konvertovati svaki od argumenata. Zato format mora da ima tačno onoliko objekata za specifikaciju konverzije koliko argumenata ima printf. Ovi objekti počinju znakom % i uključuju (pored ostalog) neki od karaktera "d", "i" (za izdavanje celog broja), "c" za izdavanje jednog karaktera, "s" za izdavanje stringa (niske karaktera), "f", "e", "g" za izdavanje realnog broja, itd.

Na primer, poziv funkcije

printf("indeks trazenog elementa je %d", BinarPret(A,n,x));

iz našeg primera, izdaje tekst "indeks trazenog elementa je", a zatim pod kontrolom objekta specifikacije konverzije %d (dakle, u obliku celog broja), izdaje vrednost svog jedinog argumenta – vrednosti funkcije BinarPret(A,n,x). Ako je, na primer, vrednost te funkcije ceo broj 3, funkcija printf izdaće tekst

indeks trazenog elementa je 3

Funkcija scanf je analogon funkcije printf za formatirani unos. Ova funkcija uzima nisku karaktera sa standardnog ulaza, pronalazi (prepoznaje) podniske čija sintaksa odgovara tipovima (objektima specifikacije konverzije) iz formata, konvertuje te podniske karaktera u odgovarajuće tipove (npr. cele, realne brojeve) i dodeljuje ih promenljivim – svojim argumentima arg_1, arg_2, ... Preciznije, dodela se vrši promenljivim čije su **adrese** arg_1, arg_2, ..., jer, za razliku od funkcije printf, svi argumenti funkcije scanf osim prvog (formata) moraju da budu pokazivači.

Na primer, poziv funkcije

scanf("%d", &x);

iz našeg primera, učitava nisku karaktera (koja se sastoji od dekadnih cifara), konvertuje je u ceo broj i vrednost tog celog broja dodeljuje celobrojnoj promenljivoj x u memoriji.

Funkcija scanf vraća broj uspešno prepoznatih i dodeljenih vrednosti.

Kako operacije ulaza i izlaza nisu deo samog C jezika, neslaganje u tipu ili broju komponenti formata i odgovarajućih argumenata ne može da bude otkriveno u vreme kompilacije, nego tek u vreme povezivanja ili izvršavanja programa.

7

Osnovni tipovi podataka

7.1 O tipovima

U matematici se promenljive klasifikuju po svojim bitnim svojstvima kao realne, kompleksne, logičke, skupovne, skupove skupova, funkcije, funkcionale, skupove funkcija, itd. U računarskoj obradi podataka klasifikacija se pokazuje još važnijom.

U matematici se tip podatka prepoznaje iz konteksta (npr. 5 je ceo broj) ili se o tipu podatka uvodi pretpostavka (npr. "neka je x realna promenljiva..."), ili se on izvodi iz definicije operacije (npr $\sqrt{x}$ – je realni broj). U programiranju se tip eksplicitno deklariše, i ukazuje na dimenziju prostora koji kompilator određuje za podatak tog tipa. Pri tome važi:

- tip određuje skup vrednosti
- tip se može odrediti iz oblika (npr. konstante) ili deklaracije (npr. promenljive), bez izvršavanja procesa računanja
- vrednosti jednog tipa imaju određen – isti način predstavljanja u memoriji računara
- svaka operacija ili funkcija "očekuje" argumente određenog tipa i proizvodi rezultat određenog tipa. Ako operacija uzima argumente različitih tipova (npr. "+" nad tipom float, int), tip rezultata se određuje iz jezičkih pravila.

U većini slučajeva, novi tip podataka definiše se nad prethodno definisanim tipovima. Vrednosti takvih tipova su kolekcije **komponentnih vrednosti** prethodno definisanih **konstituentnih tipova**, i takav, novi tip naziva se **strukturanim** (ili strukturnim) tipom.

Ako se tip sastoji samo od jednog konstituentnog tipa, on je **primitivni** (ili osnovni, ili bazni) tip.

Broj različitih vrednosti tipa je njegova **kardinalnost**. Kardinalnost osnovnog tipa T obezbeđuje meru prostora potrebnog za predstavljanje vrednosti tipa T.

Komponentni tipovi u okviru strukturnog tipa mogu biti opet struktuirani, ali su krajnje komponente obavezno atomične (primitivnih tipova). Zato je neophodno da programski jezik obezbedi mogućnost uvođenja primitivnih (nestruktuiranih) tipova.

Jedan način za uvođenje primitivnih tipova jeste **nabrajanje** (enumeracija). Ovo su **nabrojivi** tipovi i njih definiše programer, npr. u C-u

enum mesec {JAN=1, FEB, MAR, APR, MAJ, JUN, JUL, AVG, SEP, OKT, NOV, DEC};

Drugi način za uvođenje primitivnih tipova su predefinisani, **standardni** tipovi podataka. Ovi tipovi obično uključuju brojevne i znakovne (a često i logičke) vrednosti.

Programski jezik opšte namene mora da ponudi i više **metoda struktuiranja**, kojima se grade struktuirani tipovi. U programerskom smislu oni se razlikuju po operatorima za konstruisanje vrednosti i izbor komponenti tih vrednosti. Osnovne metode struktuiranja tipova u programiranju su **niz**, **slog** i **sekvenca** (datoteka), a u C-u **niz** i **struktura**. Kompleksnije strukture podataka se obično ne definišu kao "statički" tipovi (za čije vrednosti se rezerviše memorijski prostor i čuva za sve vreme izvršenja programa ili neke programske celine), već se "dinamički" generišu za vreme izvršavanja programa (prostor se rezerviše i oslobađa po potrebi). Ove strukture uključuju liste, stabla (drveta), i uopšte, **konačne grafove**.

Najznačajniji primitivni (osnovni) operatori programskog jezika su **poređenje** i **dodela**, tj. test jednakosti (i uređenja ako je definisano), i komanda za **nametanje** jednakosti. Razlika između ova dva operatora najčešće je vidljiva i u notaciji. Na primer, u C-u test jednakosti ima oblik $x == y$, dok osnovna dodela ima oblik $x = y$. Ovi operatori su definisani za skoro sve tipove podataka.

Standardni primitivni tipovi podataka zahtevaju i standardne primitivne operatore. Na primer, brojevni i logički tipovi podataka zahtevaju aritmetičke i logičke operatore.

Svakom standardnom tipu pridružene su dopuštene operacije i standardne funkcije. Neke standardne funkcije imaju isti tip kao i argument (na primer, funkcije koja izračunavaju apsolutnu vrednost broja u programskom jeziku C su funkcije abs(i) − za ceo broj i fabs(x) − za realni broj), a mogu transformisati i vrednost jednog tipa u drugi tip (na primer, funkcija koja vraća celobrojnu vrednost niske cifara s u C-u je atoi(∗s)).

Struktuirane vrednosti se generišu tzv. **konstruktorima**, a komponente se izdvajaju tazv. **selektorima**. Svaki struktuirani tip ima pridružen par ovakvih operatora transfera.

7.2 Osnovni tipovi u C-u

Standardnih osnovnih tipova u C-u je veoma malo. To su:

znakovni ili karakterski tip – char – zauzima jedan bajt, može da primi jedan karakter u lokalnom skupu karaktera;

celobrojni tip – int – obično veličine celog broja na pripadnom računaru (npr. 2 ili 4 bajta);

realni broj jednostruke preciznosti – float – zauzima 4 bajta i ima preciznost od oko 7 dekadnih cifara;

realni broj dvostruke preciznosti – double – zauzima 8 bajtova i ima preciznost od oko 16 dekadnih cifara.

7.2.1 Znakovni tip

Znakovni tip je konačni i uređeni skup karaktera.

Skup karaktera je uređen prema uređenju njihovih unutrašnjih kodova u odgovarajućoj kodnoj šemi (npr. ASCII kodu).

Osnovni objekti podataka nad kojima program vrši operacije jesu *konstante* i *promenljive*. *Znakovna konstanta* je **ceo broj**, zapisan u obliku jednog karaktera između jednostrukih navodnika, npr. 'x'. Vrednost znakovne konstante je vrednost kôda tog karaktera u pripadnom karakterskom skupu odnosno kodnoj šemi. Na primer, vrednost znakovne konstante '0' je 48, i ta dva zapisa imaju identičnu vrednost – ASCII kôd znaka '0'; ipak, zapis '0' je pouzdaniji jer ne zavisi od pripadne kodne šeme. Znakovne konstante mogu da učestvuju u aritmetičkim izrazima baš kao i drugi celi brojevi.

Neki karakteri predstavljaju se kodnom sekvencom, na primer \n (karakter za novi red), \t (tabulator), \b (karakter unazad – engl. backspace), \? (znak pitanja), \\ (kosa crta unazad – engl. backslash), itd; ove sekvence izgledaju kao dva karaktera, ali predstavljaju jedan karakter.

Znakovna konstanta '\0' predstavlja karakter sa vrednošću 0, tj. ima vrednost 0. To je tzv. nula-karakter.

Znakovna konstanta može da bude zapisana i u obliku oktalnog broja, na primer: '\101' (dekadna vrednost 65 tj. konstanta 'A'), '\61' (dekadna vrednost 49 tj. konstanta '1'), odnosno heksadekadnog: '\x41' (dekadna vrednost 65 tj. konstanta 'A'), '\x31' (dekadna vrednost 49 tj. konstanta '1').

Znakovnoj konstanti (kao i konstantama drugih tipova) može se dodeliti ime (identifikator) preprocesorskom direktivom #define, npr.

#define GRANICNIK '$'

a one se zatim mogu koristiti u izgradnji konstantnih izraza kojima se, na primer, deklarišu nizovi:

#define MAXLINE 1000

char linija[MAXLINE+1]; /* niz znakova linija ima MAXLINE+1 element */

Uvođenjem identifikatora za konstante postiže se pouzdanije i lakše programiranje i modifikacija programa.

Niska karaktera, ili *literal*, ili *string*, jeste niz od 0 ili više karaktera navedenih između **dvostrukih** navodnika, kao, na primer, "ovo je string". String je zapravo niz karaktera sa nula-karakterom na kraju. Standardna biblioteka funkcija sadrži funkciju strlen(s) koja se deklariše u standardnom zaglavlju <string.h> i koja vraća dužinu stringa s, ne računajući nula-karakter.

Dakle, bitna razlika između znakovne konstante i stringa je što je znakovna konstanta **ceo broj**, a string je niz karaktera koji se završava nula-karakterom.

Promenljiva ima ime (identifikator), tip, adresu i vrednost. Identifikator se sastoji od slova i cifara (prvi karakter je slovo) pri čemu se i podvlaka (_) računa kao slovo, a mala i velika slova su različiti karakteri. Dužina identifikatora zavisi od vrste promenljive – identifikatori unutrašnjih promenljivih mogu da budu i do 31 karakter dugački, a spoljašnjih, kao i imena funkcija, do 6.

Definicijom promenljivih uvode se nove promenljive u program, tj. imena promenljivih koje će se koristiti nabrajaju se, navode se njihovi tipovi (za znakovne promenljive – tip char), rezerviše se memorijski prostor za te promenljive (za promenljive tipa char 1 bajt) i eventualno dodeljuju početne vrednosti. *Deklaracijom* promenljivih nabrajaju se promenljive, sa svojim tipovima, koje će se koristiti u određenoj programskoj celini (npr. datoteci) a koje su definisane na drugom mestu u programu. Deklaracijom se ne dodeljuje memorijski prostor niti se vrši inicijalizacija promenljive.

Na primer,

char c, linija[1000];

je definicija znakovne promenljive c i znakovnog niza linija, dok je

extern char c;

deklarcija promenljive c koja omogućuje njeno korišćenje u datoteci u kojoj se deklaracija navodi.

Promenljiva znakovnog tipa može da bude i takva da joj se jednom dodeljena vrednost ne menja, a može da bude i parametar (funkcije) kome funkcija ne može da menja vrednost. Na primer,

const char poruka[ ] = "warning: ";

int strlen(const char[ ]);

Na promenljive tipa char može da se primeni kvalifikator *signed* (označen) ili *unsigned* (neoznačen), pri čemu se neoznačeni tip interpretira kao skup pozitivnih brojeva ili 0. Na primer, ako je tip char dužine 8 bita, onda promenljive tipa unsigned char imaju vrednosti od 0 do 255 (reprezentacija 00000000 predstavlja vrednost 0 a reprezentacija 11111111 predstavlja vrednost 255 tj. $1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$). S druge strane, tip signed char uključuje vrednosti od -128 do 127 (ima ih takođe $2^8 = 256$, ali je interpretacija zapisa – ceo broj u tzv. potpunom komplementu: negativan broj se dobija kada se u binarnom zapisu odgovarajućeg pozitivnog broja binarne cifre dopune do 1 – 0 se zamenjuje sa 1 a 1 sa 0, a zatim se tako transformisani zapis sabira sa 1. Tako najveću vrednost ima zapis 01111111 – vrednost je 127, dok zapis 10000001 (10000000+1) ima vrednost -127. Broj za 1 manji ima zapis 10000000 pa se njime predstavlja

broj −128). Za zapis označenih (signed) brojeva vcija sabiranja izvodi na isti način kao i za označene brojeve, pri čemu se prenos sa najviše pozicije zanemaruje.

Znakovnom tipu odgovaraju funkcije iz standardnog zaglavlja <**string.h**>, kojima se stringovi kopiraju, dopisuju, porede, pronalaze podstringovi, određuje dužina, itd. Za uzimanje karaktera sa standardnog ulaza (izdavanje karaktera na standardni izlaz) koriste se najjednostavnije funkcije ulaza/izlaza int getchar(void) odnosno int putchar(int) (iz standardnog zaglavlja <stdio.h>).

Primer: brojanje razmaka (karaktera ' ') i ostalih karaktera, sve do pojave karaktera '.'

```
#include <stdio.h>
#define BLANKO ' '
#define TACKA '.'
main(){
    int blanko = 0, ostali = 0;
    char c;
    c = getchar();
    while( c != TACKA ) {
        if( c == BLANKO ) blanko = blanko+1;
        else ostali = ostali + 1;
        c = getchar();
    }
    printf("Tekst sadrzi %d razmaka i %d drugih karaktera\n" blanko, ostali
);
}
```

7.2.2 Celobrojni tip

U računaru se može predstaviti samo konačno mnogo celih brojeva, tj. celobrojni tip (u C-u **int**) je konačni podskup skupa celih brojeva. Taj podskup je definisan implementacijom, i zavisi od dužine (u bajtovima tj. bitovima) kojom se ceo broj predstavlja. Ako se podatak tipa int predstavlja sa dva bajta (16 bitova), onda se mogu predstaviti celi brojevi u intervalu

$[-2^{15}, 2^{15} - 1]$, tj. [-32768; 32767].

Upotreba brojeva izvan ovog intervala dovodi do greške.

Na celobrojni tip int mogu se primeniti i dodatni kvalifikatori, short i long (i long long od standarda iz 1999.). Oni označavaju veličinu celog broja, tj. broj bitova za njegovu reprezentaciju; u nekim implementacijama, short int (kratki ceo broj) je 16 bita (2 bajta) dugačak, long int (dugački ceo broj) je 32 bita (4 bajta), a "obični" int je 2 ili 4 bajta i važi da je short int $\leq$ int $\leq$ long int.

Unarni operator **sizeof(tip)** (ili **sizeof izraz**) vraća broj bajtova potrebnih za skladištenje vrednosti tog tipa (odnosno izraza). Na primer sizeof(char) je 1, sizeof(int) je 2 ili 4. Operator može da se primeni na sve tipove odnosno izraze.

U standardnom zaglavlju <**limits.h**> definisane su konstante-ograničenja tipa int, zavisna od implementacije.

Celobrojni tip, slično znakovnom, može biti kvalifikovan kao signed (označen) ili unsigned (neoznačen), sa istom interpretacijom i memorijskom reprezentacijom kao i odgovarajući znakovni tip.

Tip konstante očigledan je iz samog njenog zapisa – celobrojna konstanta ima uobičajenu sintaksu (npr. 1234) i njen tip je int. Ako se celobrojna konstanta završava slovom l (ili L), ili ako je suviše velika da bi bila tipa int, konstanta je tipa long int; ako se završava slovom u (U), odnosno ul (UL), konstanta je neoznačen, odnosno dugački neoznačen ceo broj. Celobrojna konstanta može da bude zapisana i u oktalnom odnosno heksadekadnom zapisu (sistemu) – ako počinje cifrom 0 odnosno cifrom 0 i slovom x (X) (cifre oktalnog sistema su 0–7, a heksadekadnog – cifre 0–9 i slova 'a'–'f', odnosno 'A'–'F'). Tako je, na primer, 31 dekadni ceo broj sa istom vrednošću kao i oktalni ceo broj 037 tj. heksadekadni ceo broj 0x1f (tj. 0X1F).

Nad operandima celobrojnog tipa dopuštene su operacije + (sabiranje), − (oduzimanje), ∗ (množenje), / (izračunavanje celobrojnog dela količnika) i % (izračunavanje ostatka pri celobrojnom deljenju). Sve operacije nad celobrojnim argumentima proizvode celobrojne rezultate.

Za operacije / i % važi sledeća relacija:
$a \ \% \ b = a - ((a \ / \ b) * b)$.

Znak – može biti i unaran (npr. –50).

Operatori uvećanja i umanjenja Pored prethodnih, na promenljive celobrojnog tipa u C-u mogu da se primene i dva specifična operatora, dodavanje 1 na argument (uvećanje za 1), "++", i oduzimanje 1 od argumenta (umanjenje za 1), "−−".

Na primer, if(c=='\n') ++n;

Ovi operatori mogu da se primenjuju pre ili posle korišćenja vrednosti promenljive. Tako, na primer, iskaz

x = n++;

dodeljuje vrednost promenljive n promenljivoj x, a zatim povećava vrednost promenljive n za 1, dok iskaz

x = ++n;

povećava vrednost promenljive n za 1 a zatim tako povećanu vrednost dodeljuje promenljivoj x.

Ove operacije mogu da se primene samo na l-vrednosti (v. 6.7, Operatori dodele) (na memorijsku adresu dobijenu, npr, operatorom referenciranja &, ili na promenljivu ili element niza, odnosno objekat koji ima fiksnu adresu u memoriji koja može da se dobije primenom operatora referenciranja, &). Operacije "++" i "−−" ne mogu da se primene na izraz, npr. "3+9", jer njegova vrednost "12" nema fiksnu adresu u memoriji. Izraz je primer tzv. "r-vrednosti" (**"r-values"** – sve ono što može da se nađe na desnoj strani operatora dodele, a uključuje l-vrednosti i

ne-l-vrednosti). Značajno je napomenuti da, mada promenljiva, posle primene operatora ++ ili −− i dalje predstavlja l-vrednost, izraz koji se tom prilikom dobije nema fiksnu adresu tj. predstavlja ne-l-vrednost (r-vrednost). Zato je korektan zapis, na primer, {i = j−−; i= ++j;} ali nije korektan izraz ++j−− pa ni iskaz i= ++j−−.

Prioritet operacija nad celobrojnim tipovima je sledeći: unarni + i −, uvećanje (++) i umanjenje (−−) – najvišeg prioriteta, zatim operatori umnožavanja ∗, /, %, i najzad operatori sabiranja i oduzimanja + i −.

Nad celobrojnim tipom, kao i nad ostalim primitivnim tipovima, dopuštene su i relacijske operacije == (jednakost), ! = (različitost), <, >, <=, >=, čiji je rezultat logičkog tipa. Tako je vrednost poređenja 3<4 tačno (1), a 4<3 netačno (0). Relacijske operacije su nižeg prioriteta od aritmetičkih. Još nižeg prioriteta su operacije dodele (npr. =).

Standardne funkcije za rad sa celim brojevima u C-u deklarisane su u standardnim zaglavljima <**stdlib.h**> i <**math.h**> standardne biblioteke. Sa rezultatom celobrojnog tipa su, na primer, abs(n) – apsolutna vrednost (n – celobrojnog tipa), atoi(s) – celobrojna vrednost niske cifara s, (zaglavlje <**stdlib.h**>**)**. Funkcije za izračunavanje najmanjeg celog broja većeg ili jednakog od x, najvećeg celog broja manjeg ili jednakog x – ceil(x), floor(x), pri čemu su i x i vrednost funkcije tipa realnog broja dvostruke tačnosti (double), na primer, nalaze se u standardnom zaglavlju <**math.h**>, zajedno sa drugim funkcijama realnog argumenta.

Primer: Neka je potrebno sabrati dva cela broja zadata njihovim nizovima cifara. Ako su to, na primer, trocifreni brojevi, onda program na C-u može imati sledeći oblik:

```
#include ¡stdio.h¿
char c11, c12, c13, c21, c22, c23;
int b1, b2, zbir;
main()
{
    scanf("%c, %c, %c, %c, %c, %c", &c11, &c12, &c13, &c21, &c22, &c23);
    b1 = ((c11 – '0')*10 +(c12 – '0'))*10 + c13 – '0';
    b2 = ((c21 – '0')*10 + (c22 – '0'))*10 + c23 – '0';
    zbir = b1+b2;
    printf("%d\n", zbir);
}
```

Jedna implementacija funkcije konverzije stringa (koji se sastoji od karaktera – cifara) u ceo broj (sa prototipom int atoi(char [ ])) biće prikazana u tački 11.1.

Međutim, ako su brojevi dugi (imaju po n cifara, npr. za n=100), oni mogu izaći iz dijapazona celih (pa i dugačkih celih) brojeva u C-u, pa se zadatak mora rešavati na drugi način. Jedan takav način je da se imitira "pismeno" sabiranje – zdesna ulevo, a da se i zbir predstavlja kao niska znakova (karaktera) (v. tačku 11.1).

7.2.3 Logički tip

Logički tip nije standardni tip u C-u, prema njegovim prvim standardima. Može se uvesti kao nabrojivi tip, ali, bez obzira na to, konstanta 0 u okviru logičkog izraza ponaša se kao logička vrednost netačno, dok se konstante različite od 0 ponašaju kao logička vrednost tačno. Slično, tačan logički izraz (na primer, poređenje 3<4) proizvodi vrednost 1, dok netačan logički izraz proizvodi vrednost 0.

Za logički tip koristi se i termin "buleanski", kao sinonim.

Za logički tip karakteristični su operatori konjunkcije, (&&), disjunkcije (||) i negacije (!).

Izrazi buleanskog tipa (sa istinitosnom vrednošću) veoma su česti u programiranju. Svako poređenje ima vrednost koja je buleanskog tipa. Razmotrimo sledeći primer.

Program koji ponavlja čitanje znaka sa ulaza i izdavanje tog znaka na izlazu, sve dok se na ulazu ne pojavi znak '$', može se zapisati na sledeći način:

```
char ch;
main()
{
    for(; (ch=getchar())!='$'; ;)
        putchar(ch);
}
```

Izraz (ch=getchar())!='$' uključuje čitanje karaktera sa standardnog ulaza, dodelu tog karaktera promenljivoj ch i poređenje (na različitost) tog karaktera sa karakterom '$'. Ovo poređenje je jednostavni logički izraz koji može da ima vrednost tačno ili netačno. Sve dok je njegova vrednost tačno (karakter je različit od '$'), ponavljaju se radnje izdavanja karaktera na standardni izlaz i čitanja novog karaktera sa standardnog ulaza; kada vrednost poređenja postane netačno (učitani karakter nije različit od '$') – sa tim radnjama se završava.

Nad jednostavnim logičkim izrazima može se graditi složeniji logički (buleanski) izraz (kao što je opisano u tački 5.1.1, "Dvovalentna logika"). Vrednost logičkog izraza izračunava se "lenjo": samo dok se ne utvrdi njegova istinitosna vrednost. Na primer, pri izračunavanju vrednosti logičkog izraza

```
(3<5)||(a/b>5)
```

ceo izraz dobija vrednost 1 ("true") već po izračunavanju prvog disjunkta, pa se drugi disjunkt i ne izračunava i može biti i nedefinisan (npr. ako je b=0).

Neka je definisan nabrojivi tip boolean i deklarisane sledeće promenljive (u C-u):

```
enum boolean {N, T};
int kolicina, kol1;
float duzina, visina;
boolean kraj;
```

Tada su primeri logičkih (buleanskih) izraza:

```
kolicina < kol1
```

kolicina == kol1 && duzina >= visina && kraj
kolicina % kol1 ==0 || kolicina <= 100.

U odnosu na matematičku notaciju, uočimo dve bitne razlike u zapisu logičkih izraza u programiranju:

1) zapis, npr. minimum $\leq$ vrednost $\leq$ maximum, na koji smo toliko navikli, može se u C-u zapisati u sličnom obliku: minimum <= vrednost <= maximum, ali, kao što je već naglašeno u preglednom delu, ovaj izraz ne mora da dobije očekivanu logičku vrednost – izračunava se sleva nadesno, pa, na primer, 5<= 2 <=1 dobiće vrednost 1 (tačno), mada se to ne očekuje. Zato se pripadnost intervalu po pravilu zapisuje kao konjunkcija dva poređenja: minimum <= vrednost && vrednost <= maximum;

2) realne brojeve je u programiranju bolje ne upoređivati na jednakost (==), zbog zaokrugljivanja u računaru, već na dovoljnu bliskost. Na primer, umesto poređenja a==b (a,b – realne vrednosti), bolje je porediti ih sa fabs(a–b)< ϵ, za dovoljno malo ϵ, npr. fabs(a–b)< fabs(a∗1E–6) (razlika na milioniti deo od a, v. sledeću tačku, "Realni tip").

7.2.4 Realni tip

Realna konstanta u C-u može da bude zapisana u neeksponencijalnom i eksponencijalnom obliku. Ona mora da ima bar jednu cifru u celom delu, ili u razlomljenom delu, i mora da ima decimalnu tačku ili eksponent; ako je u eksponencijalnom obliku, onda mora da ima bar jednu cifru i iza slova E (e). Zato su niske .5 ili 2. ili 1.E-2 ili 1E5, 0.0005, 9.10956E–28 (masa elektrona), 123.4, 1e-2, 123.4e-2, ispravne realne konstante u C-u dok niska 3.0E nije. Konstanta u eksponencijalnom obliku – npr. 123.4e-2 ima vrednost 123.4×10^{-2}. Tzv. *sintaksni dijagram* na slici 7.1 prikazuje sve načine na koje može da bude izgrađena sintaksno ispravna realna konstanta. Svaki prolaz kroz dijagram, od njegovog levog do njegovog desnog kraja, opisuje jedan takav način.

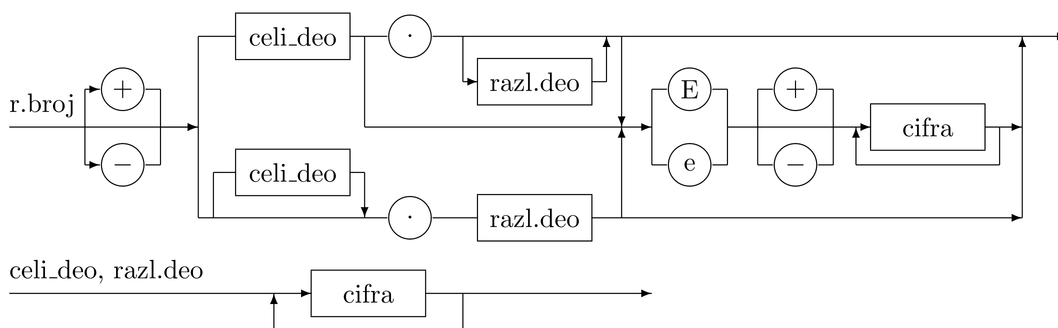

Figure 7.1: Sintaksni dijagram realne konstante

Tip realne konstante je double (realni broj dvostruke tačnosti), ona ima do 16 značajnih cifara i zauzima 8 bajtova.

Realni broj može biti i dugački realni broj dvostruke tačnosti (long double, od standarda iz 1999), čime se postiže povećana preciznost (broj značajnih cifara). Ako se realna konstanta završava slovom f (F), ona je realni broj jednostruke tačnosti (do 7 značajnih cifara, 4 bajta u memorji); ako se završava slovom l (L), konstanta je dugački realni broj dvostruke tačnosti (broj značajnih cifara i veličina potrebne memorije zavisi od implementacije).

Kvalifikator long (dugački) može se primeniti na promenljive tipa double. Prava veličina (broj bajtova) za long double zavisi od računara, ali, kao i za tip int, važi da je veličina tipa float ne veća od veličine double, a ona ne veća od veličine tipa long double.

Vrednost realnog tipa (tipa real) je element konačnog podskupa skupa realnih brojeva koji je definisan implementacijom. Implementacija definiše najmanji, najveći realni broj i gustinu tj. tačnost sa kojom se mogu registrovati realni brojevi (standardno zaglavlje <**float.h**> sadrži definicije konstanti za realni tip koje zavise od implementacije). Na primer, ako je realni x broj predstavljen sa 4 bajta, onda je on (ako je $\neq 0$) u intervalu $10^{-38} \leq |x| \leq 10^{38}$, a tačnost sa kojom se registruje je 7 značajnih cifara. Ako dva realna broja imaju po više od 7 značajnih cifara, i ako su im prvih 7 značajnih cifara jednake (i na jednakim pozicijama u oba broja), onda će ta dva broja biti registrovana kao jednaki brojevi (osim ako se zbog zaokrugljivanja nekom od njih poveća poslednja cifra). Tako, na primer, program

```
#include <stdio.h>
float x,y;
main()
{
    scanf("%f, %f", &x, &y);
    printf("x=%f, y=%f", x, y);
}
```

za dva različita ulazna broja:

1.2345678, 1.2345679

izdaje isti rezultat (sa zaokrugljivanjem na šest cifara u razlomljenom delu – sve cifre su značajne tj. tačne ili korektno zaokrugljene):

x=1.234568, y=1.234568

Kako svaki, proizvoljno mali interval na realnoj osi sadrži beskonačno mnogo vrednosti (realna osa je kontinuum), to realni tip u programskom jeziku jeste konačni skup predstavnika intervala na realnom kontinuumu. Izračunati rezultati nad realnim brojevima su aproksimacije pravih rezultata. Procenom greške pri približnom računanju sa realnim brojevima bavi se numerička matematika.

Ipak, neku predstavu o tačnosti računanja sa realnim brojevima možemo dobiti iz načina na koji se realni broj predstavlja u savremenom elektronskom računaru.

Uobičajena reprezentacija realnog broja u računaru je tzv. reprezentacija sa pokretnom tačkom (floating point representation). U ovoj reprezentaciji realni broj

x predstavljen je parom celih brojeva, e i m, tzv. **eksponent** i **mantisa**, svaki sa konačno mnogo cifara, tako da je

$x = m * B^e, -E < e < E, -M < m < M.$

U ovom zapisu m se zove **koeficijent** ili **mantisa**, a e – **eksponent**. B, E, M su celobrojne konstante koje karakterišu reprezentaciju. B se zove **baza** reprezentacije sa pokretnom tačkom, i obično nije 10 već neki mali stepen od 2 (obično $2^1, 2^3, 2^4$).

Na primer, ako se realni broj predstavlja u četiri bajta (32 bita), mantisa može biti predstavljena sa 24 bita (24-cifreni binarni broj odgovara 7-cifrenom dekadnom broju pa to određuje 7 značajnih cifara u dekadnom zapisu), a eksponent može da bude predstavljen sa 8 bitova, pa je binarni eksponent e_b (za bazu $B = 2$) $-128 \leq e_b \leq 127$, a odgovarajući dekadni eksponent e_d (t.d. $10^{e_d} \approx 2^{e_b}$) $-38 \leq e_d \leq 38$.

Za odabranu bazu B, dati broj x može biti predstavljen mnogim parovima (m, e). Na primer, za bazu B=2 i $x = 1.5$, x može da se predstavi kao $x = 3 * 2^{-1}$ $(m = 3, e = -1)$, ili $x = 6 * 2^{-2}$ $(m = 6, e = -2)$, ili $x = 12 * 2^{-3}$ $(m = 12, e = -3)$, itd. **Kanonički** ili **normalizovani** oblik realnog broja x definiše se uslovom koji mantisu svodi, deljenjem bazom, na najveći razlomljen broj maji od 1:

$\frac{1}{B} \leq m < 1.$

U prethodnom slučaju, $m = 0.75, e = 1$, tj. $x = 0.75 * 2^1$.

Ako se koristi samo ovaj oblik, onda gustina predstavnika intervala realne prave opada eksponencijalno sa povećanjem $|x|$. Na primer, interval $[0.1; 1]$ sadrži približno isto onoliko predstavnika kao i interval $[10000; 100000]$, a tačno isto toliko ako je $B = 10$ (zato što se i broj iz intervala $[10000; 100000]$ sada predstavlja kao broj iz intervala $[0.1; 1]$ – normalizovana mantisa, puta baza na odgovarajući eksponent koji je sada 5). Među predstavnicima realnih brojeva uvek su 0 i 1.

Rad sa realnim brojevima u programiranju je dosta problematičan. Probleme prave kako sabiranje i oduzimanje (skoro jednakih brojeva), tako i deljenje, zbog "odsecanja" značajnih cifara. Na primer, ako su brojevi predstavljeni sa četiri značajne cifre,

$x = 9.900, y = 1.000, z = -0.999$

1. $(x + y) + z = 10.90 + (-0.999) = 9.910$

2. $x + (y + z) = 9.900 + 0.001 = 9.901$

pa je prekršen zakon asocijativnosti.

Mera **preciznosti** date aritmetike sa pokretnom tačkom, ϵ, približno je 10^{-n}, ako računar predstavlja realni broj sa n dekadnih cifara. Preciznost ϵ (ili "epsilon mašine") definiše se kao najmanji pozitivni broj takav da 1 i $1 + \epsilon$ imaju različite predstavnike (npr. $\epsilon = 10^{-7}$).

Nad realnim tipom dopuštene su operacije +, –, ∗, ++, – / i relacijske operacije (kao i nad celobrojnim tipom). Ako je bar jedan operand realnog tipa, operacije +, ∗, –, / daju rezultat realnog tipa. Prioritet ovih operacija je kao i kod celobrojnog tipa.

Standardne funkcije koje se primenjuju nad realnim tipovima nalaze se u standardnom zaglavlju math.h. To su trigonometrijske funkcije sin, cos, tan, sinh, cosh, tanh, . . . , eksponencijalna (exp) i logaritamske funkcije – log (prirodni logaritam),

log10 (logaritam sa osnovom 10), pow(x,y) (x^y), sqrt (kvadratni koren), ceil, floor (ranije pomenute), itd. Za sve ove funkcije i argumenti i vrednost funkcija su tipa double. U ovom zaglavlju nalaze se i definicije nekih često korišćenih iracionalnih konstanti kao što su broj Pi (imenovana konstanta PI ili M_PI), broj e (E ili M_E) i sl.

Pri formatiranom učitavanju ili izdavanju realnog broja, potrebno je preneti podatak sa jednog medija na drugi (sa tastature u memoriju odnosno iz memorije na ekran), i transformisati reprezentaciju podatka (iz zapisa niske – stringa) u zapis realnog broja i obratno). Obe radnje obavlja za nas programski sistem. Standardna funkcija transfera stringa koji se sastoji od karaktera u zapisu realnog broja bez eksponenta, u realni broj, može imati sledeću definiciju:

```
#include <ctype.h>
/* atof: konvertuje string s u realni broj dvostruke tacnosti */
double atof(char [ ])
{
    double vred, stepen;
    int i, znak;
    for (i=0; s[i]=='\b' || s[i] == '\t'; i++)
    /* preskoci sve beline i tabulatore */
    ;
    znak = (s[i] == '–')? –1:1;
    if (s[i]=='+' || s[i]=='–') i++;
/* celobrojni deo */
    for(vred=0.0; '0'<=s[i] && s[i] <=' 9'; i++)
        vred = 10.0 * vred + (s[i]-'0');
    if(s[i]=='.')i++;
/* razlomljeni deo */
    for(stepen=1.0; '0'<= s[i]&& s[i]<='9'; i++)
    {
        vred = 10.0 * vred + (s[i] – '0');
        stepen *=10.0;
    }
    return znak * vred / stepen;
}
```

7.3 Izrazi

Izraz kombinuje promenljive, konstante i operacije i tako proizvodi novi objekat sa novom vrednošću i pripadnim tipom.

Izračunavanje izraza zasniva se na dve vrste pravila:

1. pravila o prioritetu i tzv. asocijativnosti operatora, i
2. pravila o tipu podizraza i izraza

Na primer, ako su zadate dodele x=15; y = x/2; onda će vrednosti promenljivih x i y zavisiti od njihovog tipa, tj. ako su x, y (ili bar y) tipa int, y će biti 7, a ako su x, y tipa float, y će biti 7.5 (primenjena su pravila druge vrste). Takođe, ako su promenljive i,j,x,y definisane sledećim definicijama

int i, j:

float x, y;

onda je izraz i-j tipa int, a izraz x*y (kao i x*i, x*j) je tipa float.

Prioriteti operatora uglavnom su već objašnjeni. Asocijativnost se odnosi na "smer" primene operatora istog prioriteta, ukoliko ih je uzastopno više - "sleva na desno" ili "zdesna na levo". Na primer, operatori +, -, *, /, % - svi su levo asocijativni, tj. $a - b + c$ računa se kao $(a - b) + c$ a ne $a - (b + c)$.

Primer desnoasocijativnog operatora je uslovni operator ?: – ako ih ima više u jednom izrazu, primenjuju se zdesna na levo. Tako, izraz

$x > y?x > z?x : z : y > z?y : z$

ima isto značenje kao

$(x > y?(x > z?x : z) : (y > z?y : z))$

i izračunava maksimum tri broja x, y, z.

Primeri desnoasocijativnih operatora su i unarni operatori (unarni +, −, kao i ++, −−).

7.3.1 Operatori nad bitovima

Pored uobičajenih (prethodno pomenutih) aritmetičkih, relacijskih i logičkih operatora, programski jezik C ima i operatore nad bitovima koji mogu da učestvuju u izgradnji izraza. Operandi bitovskih operatora mogu da budu int ili char tipa.

Operatori za manipulisanje pojedinačnim bitovima u C-u su:

& – AND nad bitovima (u rezultatu postavlja 1 samo u one bitove u kojima oba operanda imaju 1);

| – inkluzivni OR nad bitovima (postavlja 1 u sve one bitove u kojima bar jedan od operanada ima 1);

ˆ – ekskluzivni OR nad bitovima (postavlja 1 samo u one bitove u kojima tačno jedan operand ima 1);

<< – levi šift (pomeranje levog operanda ulevo za broj bitova sadržan u desnom operandu; u oslobođene bitove upisuje se 0; analogno sa množenjem odgovarajućim stepenom dvojke)

>> – desni šift (pomeranje levog operanda udesno za broj bitova sadržan u desnom operandu; najviši bit se ili ponavlja (aritmetički šift) ili se u njega upisuje 0 – logički šift).

˜ – jednostruki komplement (bitovi 1 zamenjuju se sa 0, a 0 sa 1).

Operator AND nad bitovima koristi se za maskiranje nekih bitova. Na primer,

n = n & 0177;

postavlja na 0 sve bitove osim najnižih 7 bitova od n.

Dodela

x = x & ˜077;

izvršava se tako što se prvo operatorom jednostrukog komplementa u konstanti 077 postavljaju 0-bitovi na 1 a 1-bitovi na 0, a zatim se u x ostavljaju samo 1-bitovi iz gornjih 10 pozicija (donjih 6 su bili 1-bitovi i postali su 0-bitovi).

Primer: razmena i-tog i j-tog bita celog broja koji se učitava sa standardnog ulaza

```
#include <stdio.h>
unsigned Trampa(unsigned n, int i, int j);
main()
{
    unsigned x; /*broj sa standardnog ulaza ciji se bitovi razmenjuju*/
    int i,j; /*pozicije bitova za trampu*/
    /*ucitavanje broja i pozicija za razmenu sa standarnog ulaza*/
    scanf("%u %d %d", &x, &i, &j);
    printf("\n Nakon trampe vrednost unetog broja je %u\n", Trampa(x,i,j));
}
unsigned Trampa(unsigned n, int i, int j)
{
//ako se bit na poziciji i razlikuje od bita na poziciji j, treba ih invertovati
    if ( ((n>>i)&1) != ((n>>j)&1) ) n^= (1<<i) | (1<<j);
    return n;
}
```

7.3.2 Konverzija tipova

Do konverzije tipova dolazi u nekoliko situacija:

1. Kada se uz aritmetički binarni operator navedu operandi različitog tipa, oni se konvertuju na zajednički tip koji se određuje prema sledećem skupu pravila:

   Ako je jedan od operanada tipa long double, konvertovati drugi operand u long double,

   inače, ako je jedan od operanada tipa double, konvertovati i drugi u tip double,

   inače, ako je jedan od operanada tipa float, konvertovati i drugi u tip float,

   inače, konvertovati tipove char i short u tip int.

   Ako je jedan od operanada tipa long int, konvertovati i drugi u tip long int.

   Automatske konverzije vrše se iz "užih" u "šire" tipove pri čemu se ne gubi informacija (npr celobrojni u realni tj. int u float). Ovakva konverzija naziva se *promocijom tipa.* Konverzija vrednosti izraza kojom može da se izgubi informacija (npr. iz long int u int ili short int, iz float u int kao u izrazu i=f) je dopuštena i naziva se *democijom tipa.*

   Karakteri su celi brojevi i sa njima se može izvoditi kompletna aritmetika.

Pri konverziji karaktera (jednog bajta) u ceo broj (bar dva bajta), karakter (tj. njegova celobrojna vrednost) upisuje se u niži bajt celog broja. U slučaju da se pri toj konverziji karakteri interpretiraju kao neoznačeni – unsigned, viši bitovi celog broja popunjavaju se nulama i odgovarajući ceo broj je uvek pozitivan. Ako se karakter interpretira kao označeni (signed) broj, karakter koji u najvišem bitu ima 1 prevodi se u negativan broj (i viši bitovi se popunjavaju jedinicama).

2. Pri dodeli (=), tip desne strane u dodeli konvertuje se u tip leve strane dodele. Pri tome, karakter se konvertuje u ceo broj, dugi celi brojevi konvertuju se u kraće odbacivanjem potrebnog broja viših bitova. Dodela realnog broja celobrojnoj promenljivoj odseca razlomljeni deo.

3. Najzad, konverzija tipova se može vršiti i eksplicitno, unarnim operatorom podešavanja tipova (engl. cast) oblika

   (*ime-tipa*) *izraz*

   Tada se *izraz* konvertuje u navedeni tip primenom prethodnih pravila. Na primer, funkcija sqrt definisana u standardnoj biblioteci u standardnom zaglavlju <math.h>, ima argument tipa double i proizvodi besmisleni rezultat ako se primeni na argument drugog tipa (jer se reprezentacija argumenta tog drugog tipa interpretira kao tip double). Zato se ova funkcija može pozvati za celobrojni argument tako što se on prvo eksplicitno konvertuje u tip double:

   sqrt((double) n)

   Ili, razlomljeni količnik celih brojeva i,j:

   x=(float) i /j;

7.4 Prioritet operatora.

Programski jezik C ima veliki broj prioriteta operatora zbog čega su zagrade gotovo nepotrebne (osim da povećaju čitljivost).

Dosad opisani oparatori mogu se grupisati po prioritetima u sledeće grupe i poređati na sledeći način (prioritet opada sa porastom rednog broja; poslednja kolona se odnosi na asocijativnost):

<table>
<tr><td>1.</td><td>(), [ ]</td><td>sleva nadesno</td></tr>
<tr><td>2.</td><td>!,˜, ++,−−, +, −, ∗, &, (tip)</td><td>zdesna nalevo</td></tr>
<tr><td>3.</td><td>∗, /, %</td><td>sleva nadesno</td></tr>
<tr><td>4.</td><td>+, −</td><td>sleva nadesno</td></tr>
<tr><td>5.</td><td><<, >></td><td>sleva nadesno</td></tr>
<tr><td>6.</td><td><, <=, >, >=</td><td>sleva nadesno</td></tr>
<tr><td>7.</td><td>==, ! =</td><td>sleva nadesno</td></tr>
<tr><td>8.</td><td>&</td><td>sleva nadesno</td></tr>
<tr><td>9.</td><td>ˆ</td><td>sleva nadesno</td></tr>
<tr><td>10.</td><td>|</td><td>sleva nadesno</td></tr>
<tr><td>11.</td><td>&&</td><td>sleva nadesno</td></tr>
<tr><td>12.</td><td>||</td><td>sleva nadesno</td></tr>
<tr><td>13.</td><td>?:</td><td>zdesna ulevo</td></tr>
<tr><td>14.</td><td>=, +=, −=, ∗=, /=, %=, &=, ˆ=, |=, <<=, >>=</td><td>zdesna ulevo</td></tr>
<tr><td>15.</td><td>,</td><td>sleva udesno</td></tr>
</table>

Unarni operatori +, −, ∗ & imaju viši prioritet od svojih binarnih analogona.

Na primer,
x = y==z isto je što i x = (y==z)
-a*b+c isto je što i ((-a)*b)+c
a=b=c=d isto je što i a=(b=(c=d))

8

Formalni opis konstrukcija jezika

Postoji više načina da se definišu sintaksno ispravne konstrukcije veštačkog jezika, na primer identifikatora, konstante, izraza, i sl. Ako je jezik konačan (i mali), moguće je formirati spisak svih niski (reči) jezika. Jedan primer takvog jezika je jezik saobraćajnih znakova. Drugi takav primer je jezik nad azbukom $A = \{a, b\}$ predstavljen sledećim skupom reči: $\{a, b, aa, bb, ab, ba, aaa, bbb\}$.

Ako je jezik veliki, i posebno ako je beskonačan, potrebno je zadati konačni skup pravila kojima se grade sve sintaksno ispravne konstrukcije jezika, tj. sve reči jezika.

U matematici postoji oblast koja izučava tzv. formalne jezike, koji se, prema određenim pravilima, izgrađuju nad datom azbukom. Programski jezici su samo jedan podskup skupa formalnih jezika. U opštem slučaju formalnih jezika, način za izgradnju ispravnih konstrukcija – reči jezika zadaje se formalizmom koji se zove *formalna gramatika*. Ovaj formalizam, pored azbuke samog jezika (skupa završnih simbola), koristi još jedan skup simbola, disjunktan sa azbukom jezika, tzv. pomoćnih simbola pomoću kojih se izgrađuju ispravne reči jezika. Tu je, naravno, i skup pravila, koja opisuju kako se ispravne konstrukcije jezika grade.

8.1 Formalna gramatika

Ilustrujmo formalizam formalnih gramatika prvo jednim primerom.

Primer 1.

Posmatrajmo azbuku koja se sastoji od cifara dekadnog brojnog sistema i znakova aritmetičkih operacija $+, -$, tj. $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\}$, i posmatrajmo cele brojeve nad ovom azbukom. Tako, na primer, niske +1205, 1205 i −1205 jesu celi brojevi, a niske 12+05, 1205+, 12−05, 1205− to nisu. Jezik celih

dekadnih brojeva (sa dopuštenim nevažećim nulama) može se izgraditi nad ovom azbukom primenom sledećih pravila:

1. **broj** je neoznačen ceo broj, + neoznačen ceo broj, ili − neoznačen ceo broj;

2. **neoznačen ceo broj** je cifra, ili neoznačen ceo broj na koji je dopisana cifra;

3. **cifra** je svaki simbol iz skupa $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

U izgradnji reči ovog jezika (celih dekadnih brojeva) učestvuju pojmovi "broj", "neoznačen ceo broj", "cifra" (ili, možemo ih označiti simbolima B, b, c), koji čine pomoćni skup simbola. Reči se izgrađuju primenom navedenih pravila, a u izgradnji reči polazi se upravo od pomoćnog simbola "broj" tj. B, jer je to osnovni objekat jezika koji gradimo. Umesto rečenicama prirodnog jezika pravila zapisujemo kraće – i preciznije, na sledeći način:

1'. $B \longrightarrow b| + b| - b$;

2'. $b \longrightarrow c|bc$;

3'. $c \longrightarrow 0|1|2|3|4|5|6|7|8|9$;

Polazeći od početnog simbola – ceo dekadni broj (B), i primenjujući pravila dobićemo ispravne reči iz jezika, tj. korektne cele dekadne brojeve. Na primer, broj 1234 može se izvesti iz početnog simbola B primenom nekih od navedenih pravila na sledeće niske:

$B \Rightarrow b \Rightarrow bc \Rightarrow bcc \Rightarrow bccc \Rightarrow cccc \Rightarrow 1ccc \Rightarrow 12cc \Rightarrow 123c \Rightarrow 1234$.

Proces izvođenja je završen jer smo dobili reč – broj koji se sastoji samo od slova azbuke nad kojom se jezik gradi. Ovo nije jedino izvođenje broja 1234, jer se i drugačijim redosledom zamene simbola c odgovarajućom cifrom može dobiti isti broj (npr. $\ldots cccc \Rightarrow ccc4 \Rightarrow cc34 \Rightarrow c234 \Rightarrow 1234$).

Takođe, polazeći od početnog simbola i primenjujući pravila ne može se dobiti nijedna niska koja ne pripada jeziku – npr. ne može se dobiti niska 12+23.

Primenom navedenih pravila može se dobiti svaki ceo dekadni broj. Ako je broj n-tocifren, dovoljno je jedanput primeniti pravilo $B \longrightarrow b(-b, +b)$, $n-1$ puta primeniti pravilo $b \longrightarrow bc$, jedanput pravilo $b \longrightarrow c$, i n puta pravilo oblika $c \longrightarrow$(odgovarajuća cifra).

Formalna gramatika je uređena četvorka (N, T, P, S), gde je:

N – konačni skup pomoćnih (nezavršnih, neterminalnih) simbola koji se koriste u izvođenju niski jezika (ali ne učestvuju u samim niskama jezika);

T – konačni skup završnih (terminalnih) simbola – azbuka, tj. skup slova nad kojim se definiše jezik; uslov za skup T je da je $N \cap T = \emptyset$.

P – konačni skup pravila izvođenja niski nad $N \cup T$ iz drugih niski nad $N \cup T$, tj. konačni podskup skupa $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$; element (a, b) iz skupa P zove se gramatičko pravilo i zapisuje se u obliku $a \to b$. Niska a mora da sadrži bar

jedan element iz skupa pomoćnih simbola (N), dok niska b može biti proizvoljna niska nad unijom skupova završnih i pomoćnih simbola.

S – početni simbol, $S \in N$.

Sada se za prethodni primer formalna gramatika može definisati sledećim skupovima:

$N = \{c, b, B\}$;

$T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\}$;

$P = \{c \to 0, c \to 1, c \to 2,\ c \to 3,\ c \to 4,\ c \to 5,\ c \to 6,\ c \to 7,\ c \to 8,\ c \to 9, b \to c, b \to bc, B \to b, B \to +bB \to -b\}$;

$S = B$.

Za datu gramatiku $G(N, T, P, S)$, *gramatička forma* se definiše na sledeći način:

1. početni simbol S je gramatička forma;

2. ako je xyz gramatička forma i $y \longrightarrow u$ gramatičko pravilo, onda je xuz takođe gramatička forma (to su sve niske koje se mogu dobiti iz S primenom pravila iz P).

Završna niska ili *niska generisana gramatikom* je gramatička forma koja ne sadrži pomoćne simbole.

Jezik generisan gramatikom je skup završnih niski izvedenih iz početnog simbola S.

8.2 Meta jezici

Programski jezici čine jedan pravi podskup skupa formalnih jezika. Njihova sintaksa se zbog toga može opisati sredstvima koja su pogodnija (od formalnih gramatika) za predstavljanje i jednostavnija za razumevanje. Ta sredstva su tzv. *meta jezici.* To su jezici kojima se opisuje jezik koji se izučava. Sâm jezik koji se izučava, npr. programski jezik, zove se *objekt jezik.*

Kada je reč o prirodnom jeziku, onda se njegova sintaksa i semantika opisuju takođe prirodnim jezikom, tj. i objekt jezik i meta jezik u slučaju prirodnog jezika je – taj prirodni jezik. Na primer, u gramatikama srpskog jezika opisuju se, opet rečenicama srpskog jezika, pravila koja važe za konstrukciju pravilne rečenice, za promenu pojedinih vrsta reči, itd. U rečniku srpskog jezika može se naći značenje pojedinačnih reči i fraza srpskog jezika.

Sa veštačkim jezikom, i posebno programskim jezikom, stvar stoji bitno drugačije. Na primer, za jezik saobraćajnih znakova "sintaksa" (izgled) konstrukcija (saobraćajnih znakova), s obzirom da ih je konačno mnogo, opisuje se eksplicitnim pobrojavanjem slika koje te znakove predstavljaju, dok se njihova "semantika" (značenje) opisuje prirodnim jezikom. Programski jezik je beskonačan jer se u njemu može napisati beskonačno mnogo programa. Sintaksu programskog jezika moguće je takođe opisati prirodnim jezikom, ali bi taj opis bio glomazan i najčešće neprecizan, kao što pokazuje sledeći primer. Može se reći: program na programskom jeziku C sastoji se od preprocesorskih direktiva (npr. #include, #define), definicija globalnih promenljivih i definicija funkcija. Sledio bi opis pravila po kojima se gradi preprocesorska direktiva, definicija promenljive, definicija funkcije,

lista parametara, parametar, blok, iskaz, izraz, identifikator, itd, do najjednostavnijih konstrukcija jezika (leksema). Zato se za opisivanje sintakse programskog jezika ne upotrebljava prirodni jezik već neki namenski građen, pregledniji i jednostavniji jezik.

Za formalno opisivanje sintakse programskog jezika koriste se najčešće dva meta jezika: Bekus-Naurova forma, BNF (ili proširena – "extended" – Bekus-Naurova forma, EBNF) i sintaksni dijagrami (sintaksnim dijagramima predstavljena je sintaksa realne konstante na slici 7.1).

Formalno opisivanje semantike je dosta kompleksno, pa se po pravilu u praksi semantika opisuje vezivanjem za sintaksu, ili intuitivno, korišćenjem prirodnog jezika.

8.2.1 Bekus-Naurova forma

Bekus-Naurova forma (Džon Bekus – John Backus, SAD, Peter Naur – Danska) je jedan meta jezik za formalno opisivanje sintakse programskih jezika. U ovoj notaciji sintaksa programskog jezika opisuje se pomoću konačnog skupa *metalingvističkih formula* (MLF).

Metalingvistička formula se sastoji iz leve i desne strane razdvojene specijalnim, tzv. "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.*

Metalingvistička promenljiva je fraza prirodnog jezika u uglastim zagradama $(<, >)$, i ona predstavlja pojam, tzv. *sintaksnu kategoriju* objekt-jezika. Ona odgovara pomoćnom simbolu formalne gramatike. Na primer, u programskom jeziku, sintaksne kategorije su <program>, <ceo broj>, <cifra>, <znak broja>, <ime promenljive>, itd. U prirodnom jeziku, sintaskne kategorije su < reč>, <rečenica>, <interpunkcijski znak>, itd. Metalingvistička promenljiva ne pripada objekt jeziku.

"Sitnije" sintaksne kategorije nazivaju se **leksičkim kategorijama** ili **rečima** (imaju značenje ali ne stoje samostalno; npr. identifikator, konstanta u programskom jeziku). "Krupnije" sintaksne kategorije nazivaju se **rečenicama** (osim što imaju određeno značenje, egzistiraju samostalno; npr. iskaz u programskom jeziku).

Metalingvistički izraz se gradi od metalingvističkih promenljivih i metalingvističkih konstanti, uz korišćenje drugih specijalnih, tj. univerzalnih metalingvističkih simbola.

Metalingvističke konstante su simboli objekt jezika. To su, na primer, $0, 1, 2, +, -$, ali i rezervisane reči programskog jezika, npr. IF, THEN, BEGIN, FUNCTION, itd.

Metalingvistički izraz može se sastojati od jedne metalingvističke promenljive ili jedne metalingvističke konstante, a može se izgraditi i na jedan od sledeća dva načina:

1. na već formirani metalingvistički izraz dopiše se ML promenljiva ili ML konstanta i tako se dobije novi ML izraz;

2. konačni niz prethodno formiranih ML izraza razdvojenih specijalnim tj. univerzalnim metasimbolom "|" (čita se "ili") predstavlja novi ML izraz.

Metalingvistička formula $A ::= a$ ima značenje: ML promenljiva A po definiciji je ML izraz a.

Svaka metalingvistička promenljiva koja se pojavljuje u metalingvističkom izrazu a mora se definisati posebnom MLF.

Primer 2.

Jezik celih brojeva u dekadnom brojnom sistemu može se opisati sledećim skupom MLF:

<ceo broj >::=<neoznačen ceo broj > | <znak broja> <neoznačen ceo broj>
<neoznačen ceo broj>::=<cifra> | <neoznačen ceo broj><cifra>
<cifra>::= 0|1|2|3|4|5|6|7|8|9
<znak broja>::= +|−

Navedene MLF mogu se pročitati na sledeći način:

Ceo broj je po definiciji neoznačen ceo broj ili znak broja za kojim sledi neoznačen ceo broj.

Neoznačen ceo broj je po definiciji cifra ili neoznačen ceo broj za kojim sledi cifra.

Cifra je po definiciji 0 ili 1 ili 2 ili 3 ili 4 ili 5 ili 6 ili 7 ili 8 ili 9.

Znak broja je po definiciji + ili −.

Upotreba univerzalnog metasimbola "|" samo skraćuje zapis MLF a ne povećava izražajnu moć notacije. Tako, na primer, prva MLF ima značenje ekvivalentno značenju sledeće dve MLF:

<ceo broj> ::= <neoznačen ceo broj>
<ceo broj> ::= <znak broja><neoznačen ceo broj>

Radi kraćeg zapisa i pojednostavljenja MLF uvode se razne modifikacije Bekus-Naurove forme. One se postižu uvođenjem novih metasimbola (simbola meta jezika). Jedna takva modifikacija, poznata kao proširena Bekus-Naurova forma, (EBNF – Extended Backus Naur Form) uključuje male, srednje i velike (vitičaste) zagrade, sa sledećim značenjem:

– navođenje konstrukcije između otvorene i zatvorene vitičaste zagrade {. . . } definiše ponavljanje navedene konstrukcije 0 ili više puta; na primer,

<neoznačen ceo broj>::=<cifra>{<cifra>}

(neoznačen ceo broj sastoji se od jedne cifre za kojom može da sledi proizvoljni niz cifara);

– navođenje konstrukcije između otvorene i zatvorene srednje zagrade [. . .] definiše opciono pojavljivanje konstrukcije (njeno pojavljivanje 0 ili 1 put); na primer,

<ceo broj>::= [<znak broja>]<neoznačen ceo broj>

(ceo broj je neoznačen ceo broj ispred koga se može (ali i ne mora) navesti znak broja);

– navođenje konstrukcija između otvorene i zatvorene male zagrade (...) definiše grupisanje konstrukcija; na primer,

<ceo broj> ::= [(+|−)] <cifra>{<cifra>}

(ceo broj je cifra za kojom sledi proizvoljni niz cifara, a kojoj može (ali i ne mora) da prethodi znak + ili −).

Ako simboli zagrada (malih, srednjih, velikih) mogu da pripadaju i objekt jeziku, njihovo korišćenje kao metasimbola se označava njihovim podvlačenjem.

8.2.2 Još primera

Primer 3. Identifikator

Identifikatori (imena objekata) koji se sastoje od slova, cifara i podvlake i počinju slovom ili podvlakom mogu da se opišu sledećim MLF:

<identifikator >::=<slovo>{<slovo> | <cifra>}
<slovo >::=<veliko slovo> | <malo slovo> | _
<veliko slovo>::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<malo slovo>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<cifra>::= 0|1|2|3|4|5|6|7|8|9

Primer 4. Sintaksa izraza

Prioriteti operatora odraženi su u sintaksi izraza. Sledeće tri (pojednostavljene) metalingvističke formule Bekus-Naurovom formom opisuju sintaksu aritmetičkog izraza u C-u, koja odslikava različite prioritete binarnih aritmetičkih operatora:

<arit.izraz>	::=	<multip.izraz> |	
		<arit.izraz> + <multip.izraz> |	(niži prioritet od
		<arit.izraz> – <multip.izraz>	operatora umnožavanja)
<multip.izraz>	::=	<prim.izraz> |	
		<multip.izraz> ∗ <prim.izraz> |	
		<multip.izraz> / <prim.izraz> |	
		<multip.izraz> % <prim.izraz>	
<prim.izraz>	::=	<identifikator> |	
		<konstanta>	
		<string>	
		(<arit.izraz>)	

Prema ovoj sintaksi, identifikator a kao i konstanta 5 jesu primitivni izrazi. Oni su takođe i multiplikativni izrazi, ali su multiplikativni izrazi i $a * 5$, i $a * 5 * 5$ i sl. Svi ovi multiplikativni izrazi su, sami za sebe, i aritmetički izrazi, ali su aritmetički izrazi i $a * 5 + 5$, i $a * 5 + 5 - a$ i sl. S obzirom na ovako zadatu sintaksu, pri analizi sintaksne korektnosti aritmetičkog izraza prvo će se tražiti (prepoznavati, pa prema tome i izračunavati) multiplikativni izrazi, što upravo odražava prioritet aritmetičkih operatora.

9

Iskazi – upravljanje

Iskazima se opisuje suštinski koncept programskog jezika – radnja, akcija. Iskazom se saopštava računaru da treba da uradi nešto sa podacima.

U programskom jeziku C, svaki izraz postaje iskaz kada se dopuni karakterom ';', npr.

x=0;

i++;

printf(...);

Strukturni iskazi programskog jezika su oni koji se sastoje od drugih, **komponentnih iskaza**. Iskazi koji nisu strukturni jesu **jednostavni** (primitivni, prosti) iskazi.

Strukturni iskazi predstavljaju, pre svega, upravljačke ili kontrolne strukture, tj. iskaze koji upravljaju (ili kontrolišu) izvršenjem iskaza od kojih se sastoje.

Osnovne upravljačke strukture su one koje obezbeđuju da se njihovim kombinovanjem zapiše proizvoljni algoritam. To su serijska struktura, koja proizvodi redno (serijsko) izvršenje komponenti, selektivna struktura koja proizvodi izvršenje jedne od komponenti u zavisnosti od ispunjenosti zadatog uslova, i repetitivna struktura koja proizvodi ponovljeno izvršenje komponentnih iskaza.

9.1 Jednostavni iskaz: iskaz dodele

Osnovni iskaz proceduralnih programskih jezika je **iskaz dodele:**

$a \leftarrow \Psi$,

gde je a – identifikator promenljive, a Ψ – izraz čija se vrednost izračunava, transformiše u tip promenljive a i dodeljuje kao nova vrednost promenljivoj a. Pri tome se stara vrednost promenljive a gubi. S leve strane iskaza dodele mora biti l-vrednost (v. 7.2.2).

Iskaz dodele u programskom jeziku C je zapravo izraz dodele za kojim sledi znak ";":

$a = \Psi;$

Iskaz dodele može se okarakterisati svojim **preduslovom** i svojim **postuslovom**.

Na primer, formalna specifikacija iskaza dodele $i = j + 1;$ može se zapisati korišćenjem preduslova i postuslova u obliku:

$\{j > 1\} i = j + 1; \{i > 2\}$

Prethodna formalna specifikacija ima značenje: ako pre izvršenja iskaza dodele $i = j + 1;$ važi uslov $\{j > 1\}$ (preduslov), onda će posle izvršenja tog iskaza važiti uslov $\{i > 2\}$ (postuslov).

Može se uočiti da i preduslov $\{i > 0, j > 1\}$ obezbeđuje isti postuslov $\{i > 2\}$ u slučaju prethodnog iskaza dodele, tj. $\{i > 0, j > 1\} i = j + 1 \{i > 2\}$. Razni preduslovi jednog iskaza dodele obezbeđuju važenje zadatog postuslova.

Da bi važio postuslov R iskaza dodele $a = \Psi$, neophodno je da pre izvršenja iskaza dodele važi tzv. **najslabiji preduslov** $Q = R^a_\Psi$, tj. mora biti tačan predikat R u kome je svako pojavljivanje promenljive a zamenjeno izrazom Ψ. Ovo tvrđenje se zapisuje $\{R^a_\Psi\} a \leftarrow \Psi \{R\}$, i naziva se **aksiomom dedele**. Na primer,

$\{j > 1\} i = j + 1 \{i > 2\}$, jer je $R^i_{j+1} \equiv j + 1 > 2 \equiv j > 1 \equiv Q$.

I drugim iskazima programskog jezika moguće je dodeliti odgovarajuće aksiome tj. formalne specifikacije.

U programskom jeziku C, pored osnovnog iskaza dodele postoje i prošireni iskazi dodele koji odgovaraju proširenim opertorima dodele (dodela sa aritmetičkim operatorima +=, −=, ∗=, /=, %= i bitovskim operatorima &=, ˆ=, |=, <<=, >>=). Takvi iskazi dodele imaju oblik a += Ψ;, a −= Ψ;, i sl, gde je a – promenljiva a Ψ – izraz. Ovim iskazima dodele promenljivoj a dodeljuje se vrednost izraza koji se dobije kada se binarni operator uz dodelu (+, −, ∗, . . .) primeni na vrednost promenljive a kao levi argument i vrednost izraza Ψ – kao desni. Dakle, ovi iskazi dodele ekvivalentni su iskazima osnovne dodele oblika $a = a + (\Psi)$; $a = a - (\Psi)$; $\ldots a = a >> (\Psi)$.

9.2 Serijski komponovani iskaz

Serijski komponovani iskaz je oblika

S_1
S_2
. . .
S_n

gde su $S_1, S_2, \ldots, S_n$ – iskazi.

Komponentni iskazi se "zagrađuju" nekim parom oznaka, u C-u zagradama "{", "}":

{
S_1
S_2
...
S_n
}
U C-u, komponovani serijski iskaz zove se i *blok*, i u njemu se mogu definisati promenljive lokalne za taj blok.

Važenje preduslova Q_0 serijskog komponovanog iskaza obezbeđuje važenje njegovog postuslova Q_n, ako važenje preduslova svakog komponentnog iskaza obezbeđuje važenje njegovog postuslova.

Na primer, na ovaj način dobija se zaključak o korektnosti swap-algoritma (programa) – programa za razmenu vrednosti dve promenljive x, y:
Iz
$\{x = X \wedge y = Y\} t \leftarrow x \{x = X, y = Y, t = X\}$,
$\{x = X \wedge y = Y \wedge t = X\} x \leftarrow y \{x = Y, y = Y, t = X\}$,
$\{x = Y \wedge y = Y \wedge t = X\} y \leftarrow t \{x = Y, y = X, t = X\}$,
$(X, Y$ su konstante, x, y, t – promenljive),
sledi
$\{x = X \wedge y = Y\} t \leftarrow x; x \leftarrow y; y \leftarrow t \{x = Y, y = X\}$

9.3 Uslovni iskazi (selektivna, alternirajuća upravljačka struktura)

Potpuni uslovni iskaz je oblika
ako p **onda** S_1 **inače** S_2, ili, u sintaksi C-a,
if (p)
S_1
else
S_2
gde je p – logički izraz , a S_1, S_2 su iskazi. Deo else može da se izostavi i onda je reč o nepotpunom uslovnom iskazu. Ako je vrednost logičkog izraza p – tačno (tj. u C-u ima vrednost različitu od 0), izvršava se iskaz S_1, a ako je vrednost logičkog izraza p – netačno (tj. u C-u ima vrednost 0), i ako postoji else deo, izvršava se iskaz S_2.

Komponentni iskazi S_1, S_2 u uslovnom iskazu mogu biti proizvoljni iskazi, pa i drugi uslovni iskazi. Posebno, može se dogoditi da je prvi komponentni iskaz (S_1) potpunog uslovnog iskaza – nepotpuni uslovni iskaz, ili da je jedini komponentni iskaz nepotpunog uslovnog iskaza – potpuni uslovni iskaz, tj. može se dobiti iskaz koji se u sintaksi C-a zapisuje u obliku

if (p) if (p_1) S_3 else S_4

pri čemu nije jasno da li "else" grana pripada "unutrašnjem" ili "spoljašnjem" if iskazu.

Ovakva dvoznačnost rešava se specifičnim konvencijama (dogovorima) u okviru specifičnog programskog jezika. Na primer, u C-u, ovaj iskaz pridružuje jedino else najbližem if, tj. ima značenje

if (p)
{
if (p_1) S_3 else S_4
}

Drugo značenje se može ostvariti eksplicitnim zagrađivanjem tj. sintaksom oblika

if (p)
{
if (p_1) S_3
}
else S_4

Posebno čest slučaj višestruke "odluke" pri grananju je drugi if-iskaz u else delu if-iskaza, tj. višestruki if-iskaz ima oblik

if (p_1)
S_1
else if (p_2)
S_2
else if (p_3)
...
else
S

Tačnost jednog od uslova p_i rezultuje izvršenjem odgovarajućeg iskaza S_i čime se završava izvršenje celog višestrukog if-iskaza.

Na primer, u funkciji BinarPret (v. 6.3), višestruki if-iskaz koristi se za odluku da li je vrednost ključa u sredini sortiranog (pod)niza manja, veća ili jednaka traženoj vrednosti:

```
int BinarPret(slog A[ ], int n, int k)
{
    int l=0, d=n-1,s;
    while(l <= d) {
        s = (l + d)/2;
        if(k < A[s].kljuc) d=s-1;
        else if(k > A[s].kljuc) l=s+1;
        else return(s);
    }
    return(NEUSPEH);
}
```

Za sve oblike selektivnog iskaza moguće je formulisati aksiome tog iskaza izvođenjem preduslova iz postuslova koji opisuje dejstvo tog oblika iskaza. Međutim, time se nećemo baviti (kao ni u slučaju sledećih kontrolnih struktura).

9.3.1 Uslovni iskaz sa višestrukim grananjem

Uslovni iskaz omogućuje izbor jednog od dva iskaza na osnovu dve isključive vrednosti uslova p – ili je vrednost logičkog izraza **tačno**, ili je **netačno**. U nekim slučajevima potreban je izbor jednog od konačnog broja (n) iskaza (akcija), i to na osnovu jednakosti izraza sa jednim od konačnog broja – n konstantnih celobrojnih izraza.

Tada odgovarajući selektivni iskaz višestrukog grananja (tzv. *prekidač* – engl. switch) u C-u ima oblik:

switch (i) {
  case i_1: S_1
  case i_2: S_2
  ...
  case i_n: S_n
  default: S
}

Na primer, sledeći iskaz analizira karakter c i uvećava broj pojavljivanja odgovarajuće cifre, beline ili ostalih karaktera, u zavisnosti od toga da li je karakter c cifra, belina ili nešto treće:

```
switch (c) {
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
     brcif[c-'0']++;
     break;
case ' ':
case '\n':
case '\t':
     brbel++;
     break;
default:
     brost++;
     break;
}
```

Iskaz višestrukog grananja je uopštenje uslovnog iskaza na izbor jednog od konačno mnogo slučajeva – jedne od konačno mnogo akcija – komponentnih iskaza. Iskaz se sastoji od jednog izraza – selektora i liste iskaza, od kojih svaki ima jedno ili više obeležja slučaja – konstantnih celobrojnih vrednosti. U prethodnom primeru, selektor je promenljiva c znakovnog tipa (sa celobrojnom vrednošću!), a u zavisnosti od vrednosti promenljive c u vreme izvršavanja iskaza, izvršiće se jedan od

komponentnih iskaza – onaj čije je obeležje slučaja jednako vrednosti promenljive *c*; ako takav iskaz, odnosno obeležje, ne postoji, tj. ako karakter c nije ni cifra ni belina, izvršava se podrazumevani – default – iskaz tj. uvećanje broja "ostalih" karaktera.

Posle izvršenja izabranog komponentnog iskaza, izvršenje se nastavlja na sledećem slučaju, osim ako se posle komponentnog iskaza navede iskaz bezuslovnog skoka *break*; on rezultuje momentalnim završavanjem prekidačkog (switch) iskaza.

9.4 Repetitivni iskazi

Repetitivni iskazi su iskazi kojima se realizuje kontrolna struktura ponavljanja komponentnih iskaza. Ovi iskazi sadrže jedan logički izraz – **uslov** i niz **komponentnih iskaza** (ili jedan, proizvoljno kompleksan iskaz) čije se izvršavanje ponavlja sve dok je uslov ispunjen.

U programskom jeziku C postoji iskaz ciklusa u kome se prvo proverava tačnost uslova pa se onda, ako je uslov tačan, izvršavaju komponentni iskazi (tzv. **iskaz ciklusa sa pred-proverom uslova**, **while** - iskaz), i iskaz ciklusa u kome se komponentni iskazi izvršavaju pre nego što se proverava istinitost uslova, a zatim, ako je uslov još uvek ispunjen, komponentni iskazi se ponovo izvršavaju (tzv. **iskaz ciklusa sa post-proverom uslova**, **do-while** iskaz).

Za komponentne iskaze repetitivnih iskaza koriste se termini "ciklus" i "petlja".

Jedna specifična struktura iskaza ciklusa sa pred-proverom uslova je posebno česta pa je izdvojena u poseban iskaz – **for** iskaz. To je struktura while iskaza u kome se, posle početne dodele vrednosti promenljivoj iz uslova, izračunava uslov, i sve dok je tačan ($\neq 0$), izvršava se ciklus i dodeljuje nova vrednost promenljivoj iz uslova.

9.4.1 Iskaz ciklusa sa pred-proverom uslova

Ovaj iskaz je oblika

dok je p **uradi** S

ili, u sintaksi C-a,

while (p) S

U ovom iskazu p je logički izraz a S – iskaz.

Primer – množenje dva nenegativna cela broja

Razviti program za množenje dva nenegativna cela broja bez korišćenja operacija množenja i deljenja.

Formalna specifikacija ovog problema je oblika:

$\{x, y \in N \cup \{0\}\} \to \{z = x * y\}$

Za svaki problem, pa i za ovaj koji razmatramo, postoji veći broj algoritama. Algoritam za naš problem polazi od činjenice da je množenje skraćeno sabiraje, i može da se izgradi iterativno (postupno), i to tako što će se na međurezultat,

koji je na početku = 0, dodavati x puta vrednost y. Dakle, $z = y + y + \ldots y$ (x puta). Algoritam mora da uvede pomoćnu promenljivu koja će brojati te iteracije (dodavanja), i to **ili** broj još potrebnih dodavanja (koji je na početku x a smanjuje se za po 1 sve dok ne postane 0), **ili** broj ostvarenih dodavanja (koji je na početku 0 a povećava se za po 1 dok ne postane $= x$).

Dakle, mi ćemo za naš problem razviti dva algoritma. Svaki od njih će da obezbedi da se, polazeći od zadatog preduslova ($x, y \in N \cup \{0\}$), izvršavanjem algoritma, dobija važenje postuslova $\{z = x * y\}$.

Algoritam 1

Neka je promenljiva koja broji još potrebna dodavanja u. Algoritam (program) tada ima sledeći oblik:

```
{
    z = 0;
    u = x;
    while (u > 0) {
        z = z + y;
        u = u − 1;
    }
}
```

Kada smo već, na intuitivan način, došli do ovog algoritma, možemo da ispitamo (tj. dokažemo) njegovu korektnost, tj. možemo da se uverimo da će se, za vrednosti ulaznih promenljivih (x, y) koje zadovoljavaju početni preduslov ($x, y \in N \cup \{0\}$) algoritam završiti i da će na kraju važiti traženi postuslov ($z = x * y$). Time se ovde nećemo baviti.

Algoritam 2

Neka je n promenljiva koja broji izvršena dodavanja y-a na međurezultat. Algoritam tada ima sledeći oblik:

```
{
    z = 0;
    n = 0;
    while (n < x) {
        z = z + y;
        n = n + 1;
    }
}
```

Korektnost ovog algoritma može da se dokaže na sličan način kao i za prethodni.

9.4.2 For iskaz

Iskaz ciklusa sa parametrom – for iskaz – veoma je sličan while iskazu, a izbor jednog od njih, osim što je stvar ukusa, zavisi od prirode akcije koju treba da realizuju.

Većina proceduralnih programskih jezika ima neki oblik iskaza ciklusa sa parametrom. On uključuje parametar, iskaz koji se u ciklusu ponavlja, i uslov koji treba da je ispunjen (tačan) da bi se iskaz u ciklusu ponavljao. Komponente ovog iskaza su inicijalna dodela vrednosti parametru, a zatim, ponovljeno, sve dok je uslov definisan za parametar tačan, izvršenje iskaza u ciklusu i promena vrednosti parametra.

Dakle, apstraktni oblik for iskaza je
for (i=poč_vrednost; uslov(i); i=nova_vrednost) <iskaz>

Razni programski jezici vezuju specifičnu sintaksu i semantiku za for iskaz. Neki zahtevaju obavezno prisustvo svih njegovih komponenti i strogo kontrolisanu inicijalnu dodelu i promenu vrednosti parametra. Programski jezik C dopušta fleksibilnost tako da pojedine komponente mogu da se izostave, a promena vrednosti parametra može da ima proizvoljan oblik. U programskom jeziku C, šta više, ne mora ni da bude prisutan koncept parametra kao promenljive, već su inicijalna dodela i promena vrednosti – proizvoljni izrazi opšteg tipa. Zato je for iskaz sličniji while iskazu, ne mora da se završava a i ako se završava ne može se odrediti (u opštem slučaju) broj ponavljanja iskaza u ciklusu.

U programskom jeziku C **for** iskaz je oblika

```
for(izraz1; izraz2; izraz3)
    iskaz;
```

i ekvivalentan je sledećem paru iskaza (osim što se iz for-iskaza može "prinudno" izaći iskazima bezuslovnog skoka – break, continue, goto):

```
izraz1;
while (izraz2) {
    iskaz;
    izraz3;
}
```

Izrazi *izraz1*, *izraz2*, *izraz3* su proizvoljni izrazi. Ipak, najčešće su *izraz1* i *izraz3* – dodele ili pozivi funkcija, dok je *izraz2* – logički izraz. Svaki od izraza može da se izostavi, ali znakovi ';' moraju da ostanu. Tako je iskaz

```
for (;;) {
...
}
```

iskaz beskonačne petlje (osim ako se u okviru ciklusa ne navede iskaz bezuslovnog skoka).

Dok se while iskaz koristi u slučajevima kada nema inicijalne dodele i promene vrednosti, na primer

```
while ((c=getchar()) == ' ' || c=='\n' || c=='\t');
/* preskociti beline, tabulatore i znake za novi red */
```

for iskaz se najčešće koristi kada su te aktivnosti prisutne, na primer

```
for(i=0; i<n; i++)
    blok
```

/* ponavljanje bloka iskaza za sve vrednosti i od 0 do n–1 */

Na primer, zbir prvih n prirodnih brojeva može da se izračuna sledećim iskazima:

```
s = 0;
for(i = 1; i <= n; i++ ) s = s + i;
```

ili, kraće:

```
for( s = 0, i = 1; i <= n; i++ ) s = s + i;
```

ili, još kraće:

```
for( s = 0, i = 1; i <= n; s = s + i, i++ );
```

9.4.3 Iskaz ciklusa sa post-proverom uslova

Ovo je iskaz oblika

ponavljaj S **dok** p

ili, u sintaksi C-a,

do
    S
while (p);

Opet, p je logički izraz a S – iskaz.

Iskaz S se izvršava jednom bezuslovno, a zatim, sve dok je vrednost logičkog izraza p – tačno ($\neq 0$), iskaz S se ponovo izvršava.

Iskaz do-while se ređe koristi od while iskaza ali u slučajevima kada se ciklus prema prirodi problema izvršava bar jednom – i to bezuslovno, prirodno je koristiti do-while iskaz.

Primer – Euklidov algoritam

Reč je o Euklidovom algoritmu za određivanje najvećeg zajedničkog delioca dva prirodna broja. Prikažimo prvo izvršavanje algoritma na jednom primeru. Pretpostavljamo da su elementarne operacije algoritma množenje i deljenje.

Naći najveći zajednički delilac (nzd) brojeva 14 i 5.
14:5=2(4); (14 podeljeno celobrojno sa 5 jeste 2 i ostatak pri deljenju je 4);
5:4=1(1);
4:1=4(0).
Dakle, nzd(14,5)=nzd(5,4)=nzd(4,1)=1.

Specifikacija:
$\{a, b \in N \cup \{0\}\} \rightarrow \{rezultat : nzd(a, b)\}$

Algoritam se zasniva na sledećem razmatranju: ako su brojevi a i b različiti, naći ostatak pri celobrojnom deljenju većeg manjim. Zatim, sve dok ostatak ne bude = 0, ponoviti celobrojno deljenje delioca i ostatka iz prethodnog deljenja. Ako se deljenik označi kao "prvi" broj a delilac kao "drugi", onda se algoritam zasniva na pretpostavci da je $nzd(prvi, drugi) = nzd(drugi, ostatak)$. Algoritam može da se zapiše sledećim pseudoprogrmom:

```
{
          prvi = a; drugi = b;
          if (prvi < drugi) swap(prvi, drugi);
cikl:     kol = prvi/drugi
          ost = prvi − kol ∗ drugi;
          if (ost! = 0) {
               prvi = drugi;
               drugi = ost;
               goto cikl;
          }
          else
          nzd = drugi;
}
```

Ako se, pored operacije celobrojnog deljenja, koristi i operacija određivanja ostatka pri celobrojnom deljenju, npr. % (kao u C-u), i ako se dodela promenljivim $prvi, drugi$ izvrši pre provere kraja ciklusa ($ost <> 0$), dobije se algoritam (program) sa iskazom ponavljanja sa post-proverom uslova oblika

```
{
     prvi = a; drugi = b;
     if (prvi < drugi) swap(prvi, drugi);
     do {            /∗ NZD(a, b) = NZD(prvi, drugi) ∗/
          ost = prvi % drugi;
          prvi = drugi;
          drugi = ost;
     } while (ost! = 0);
     /∗ prvi = NZD(a, b)  ∗/
}
```

(na kraju je $\mathrm{NZD}(a, b) = prvi$ a ne $drugi$ zato što je izvršena jedna suvišna izmena vrednosti promenljivih $prvi, drugi$).

I promenljiva ost može da se eliminiše, pri čemu se dobije program sledećeg oblika:

```
{
     prvi = a; drugi = b;
     do {            /∗ NZD(a, b) = NZD(prvi, drugi) ∗/
          prvi = prvi % drugi;
          swap(prvi, drugi);
     } while (drugi! = 0);
     /∗prvi = NZD(a, b)∗/
}
```

Slično, ako se Euklidov algoritam izrazi operacijom oduzimanja (umesto deljenja), on može da se opiše sledećim pseudoprogramom:

```
{
```

```
            prvi = a; drugi = b;
cikl:       if (prvi == drugi) nzd = drugi;
            else {
                  if (prvi < drugi) swap(prvi, drugi);
                  z = prvi − drugi;
                  prvi = drugi;
                  drugi = z;
                  goto cikl;
            }
}
```

Kao i u realizaciji deljenjem, i Euklidov algoritam sa oduzimanjem može se zapisati u kraćem obliku:

```
{
      prvi = a; drugi = b;
      while (prvi! = drugi) {
            if (prvi > drugi)
                  prvi = prvi − drugi;
            else
                  drugi = drugi − prvi;
            /∗ NZD(a, b)= NZD(prvi, drugi)∗/
      }
      /∗ NZD(a, b) = prvi = drugi∗/
}
```

9.4.4 Ekvivalentnost repetitivnih iskaza while i do-while

Mada jedan od iskaza while, do-while može biti adekvatniji od drugog u izražavanju upravljačke strukture ponavljanja iskaza u nekoj situaciji, ova dva iskaza su ekvivalentna u svojoj izražajnoj moći. Sve što se može izraziti jednim, može i drugim. Tako se iskaz

```
while (p)
      S
```

može zapisati i sledećim ekvivalentnim iskazom:

```
if (p)
      do
            S
      while (p);
```

Takođe, iskaz

```
do {
      S1
      S2
      ...
      Sk
```

} while (p);
može se realizovati sledećim iskazom dodele i while iskazom:

```
{
t=1;
while (t || p) {
      S1
      S2
      ...
      Sk
      t=0;
}
```

9.5 Iskazi bezuslovnog skoka

Pored iskaza bezuslovnog skoka – goto *obeležje*, u C-u postoje i iskazi break i continue. Iskaz break obezbeđuje momentalni izlazak iz unutrašnje for, while, do – petlje, kao i iz prekidačkog – switch iskaza. Iskaz continue obezbeđuje momentalni prelaz na sledeću iteraciju unutrašnje for, while ili do – petlje. U slučaju cilusa u ciklusu (dvostruke ili višestruke petlje), ovi iskazi obezbeđuju izlazak iz unutrašnje petlje odnosno prelaz na sledeću iteraciju unutrašnje petlje.

Primer – goto: nalaženje jednog zajedničkog elementa dva niza a i b, sa n odnosno m elemenata, redom.

```
      for (i=0; i<n; i++)
            for (j=0; j<m; j++)
                  if (a[i] == b[j])
                        goto nadjen;
      /* nije nadjen zajednički element */
   ...
nadjen:
      /* nadjen jedan zajednički element: a[i]==b[j] */
      ...
```

Primer – break: eliminiše završne znakove za belinu, tabulator, novi red, iz stringa s

```
int elimbel(char s[ ]) {
      int n;
      for (n=strlen(s)–1; n>=0; n−−)
            if (s[n]! =' ' && s[n] ! = '\t' && s[n]! = '\n')
                  break;
      s[n+1] = '\0';
      return n;
}
```

Primer – continue: obrada samo ne-negativnih elemenata niza a

```
for (i=0; i< n; i++) {
    if (a[i] < 0)              /* preskoci negativne elemente */
        continue;
        .../* obradi pozitivne elemente */
}
```

10

Funkcije

10.1 Definicija, deklaracija (prototip) i poziv

Da bi se omogućila dekompozicija problema koji se rešava uz pomoć računara u potprobleme, pa i dekompozicija rešenja – programa u niz elementarnijih koraka – parcijalnih programa, potrebno je u programskom jeziku koristiti koncept rutina odnosno potprograma. U C-u se ovaj koncept realizuje **funkcijama**. Korišćenje eksplicitnih potprograma donosi i druge prednosti u programiranju, kao što je mogućnost nezavisnog testiranja manjih programskih celina ili eliminisanje ponavljanja istih programskih delova u slučaju da je potrebno njihovo izvršavanje u raznim tačakama programa. Ipak, mogućnost razlaganja problema u potprobleme čija se rešenja implementiraju funkcijama predstavlja osnovni motiv za njihovo korišćenje. Dekompozicija koju obezbeđuje korišćenje funkcija olakšava rešavanje problema a u slučaju kompleksnih problema predstavlja mehanizam bez kojeg nije ni moguće rešiti problem.

Ilustrujmo poslednje tvrđenje sledećim primerom:

Primer Napisati program kojim se konstruiše niz od N znakova azbuke {1,2,3} tako "da su svaka dva susedna podniza konstruisanog niza različita". Na primer, za N=5, 12321 je dobar niz, a 23231 ili 13131 nisu.

Ideja za rešavanje problema može se opisati sledećim postupkom:

– počinje se sa praznim nizom; on zadovoljava kiterijum " ... " ali nije dovoljne dužine;

– pošto niz nije dovoljno dugačak treba ga produžiti dopisivanjem jednog znaka azbuke a zatim treba testirati dobijeni niz da li zadovoljava kriterijum "...";

– ako produženi niz nije dobar, treba pokušati sa sledećim mogućim produženjem;

– postupak produženja treba ponavljati sve dok se ne dobije niz dužine N koji zadovoljava kriterijum "...";

– ako u nekom koraku dobijemo niz nedovoljno dug a koji se ne može produžiti tako da je zadovoljen kriterijum "...", niz se mora skratiti i pokušati sa nekim drugim produženjem.

Na primer, za N=8, "dobri" delimični nizovi označeni su sa "+" a "loši" sa "-".
+ 1
- 11
+ 12
+ 121
- 1211
- 1212
+ 1213
+ 12131
- 121311
+ 121312
+ 1213121
- 12131211
- 12131212
- 12131213 (neophodno skraćenje)
- 1213122
+ 1213123
+ 12131231 (ovo je i prvi – ali ne i jedini – niz dužine 8 sa traženim svojstvom)

Ako sa m označimo celobrojnu promenljivu čija je vrednost – dužina već konstruisanog niza, sa *dobar* – logičku promenljivu čija je vrednost rezultat testa zadovljenosti kriterijuma " ..." za konstruisani niz, a sa S – konstruisani niz, globalni program koji rešava postavljeni zadatak ima oblik:

```
⋮
m =0; dobar =1;
do {
    produzi;
    proveri;
    while (!dobar) {
        promeni;
        proveri;
    }
}
while (m!=N)
```

Radnje produzi, proveri, promeni predstavljaju apstraktne iskaze kojima se nagoveštava šta treba uraditi ali ne i kako. Zato ovi iskazi treba da se konkretizuju nizom iskaza programskog jezika, tj. tim imenima (produzi, proveri, promeni) treba pridružiti odgovarajuće nizove konkretnih iskaza. Niz konkretnih iskaza koji se pridružuje imenu apstraktnog iskaza zove se u C-u **funkcija** (u opštem slučaju – rutina), ime apstraktnog iskaza zove se **ime** ili **identifikator funkcije**, pridruživanje se zove **definicija funkcije**, a izvršenje funkcije u nekom trenutku izvršavanja programa na nekom mestu u programu postiže se **pozivom funkcije**. Ako poziv funkcije prethodi njenoj definiciji, funkcija se najavljuje tzv. **deklaracijom** funkcije. Deklaracija funkcije sadrži tip povratne vrednosti funkcije, ime

funkcije i broj i tipove argumenata funkcije. Ovakav opis još se zove i *prototip* funkcije.

Funkcija može da izračunava (i vraća) vrednost i tada se tip vrednosti koju funkcija izračunava navodi ispred imena funkcije (u deklaraciji odnosno definiciji funkcije). Na primer, deklaracija (ili prototip)

int suma(void);

"najavljuje" (deklariše) funkciju suma koja izračunava i vraća vrednost celobrojnog tipa. S druge strane, deklaracija

void proveri(void);

deklariše funkciju koja ne vraća funkciji iz koje je pozvana nijednu vrednost pa zato nema tip (tip funkcije je "void").

Rezultat izvršavanja funkcije A vraća se funkciji iz koje je funkcija A pozvana, iskazom

return *izraz*

Pri tome, izraz se konvertuje u deklarisani tip funkcije, ako je potrebno, a funkcija koja poziva funkciju A može da ignoriše vraćenu vrednost.

U **return** iskazu izraz može i da se izostavi, i pri tome se ne vraća nikakva vrednost funkciji iz koje je funkcija A pozvana; isti je slučaj i ako se ceo iskaz **return** izostavi.

Rezultat izvršavanja funkcije može biti i drugog tipa (npr. double), a ako se izostavi, podrazumeva se tip int.

Funkcija može imati i parametre koji omogućuju njeno izvršavanje za različite kombinacije argumenata (v. 10.3). Tako je opšti oblik definicije funkcije

tip_rezultata ime_funkcije (*deklaracije argumenata*)
{
    definicije i iskazi
}

Definicija funkcije može se ilustrovati primerom definicije funkcije promeni:

```
void promeni(void)
{ while (S[m]=='3') { m=m-1;}
 S[m]++;
 return;
}
```

Definicija funkcije uključuje zaglavlje funkcije i telo funkcije. Primer najjednostavnijeg oblika zaglavlja funkcije jeste prvi red definicije funkcije *promeni*: sastoji se od identifikatora (imena) funkcije – *promeni* i naznake da funkcija nema argumenata (promeni(void)), kao i da ne vraća vrednost (void (promeni...)).

Telo funkcije, u najjednostavnijem slučaju, sastoji se od bloka koji sadrži samo iskaze koji su pridruženi identifikatoru funkcije – u primeru definicije funkcije *promeni* to su 2. i 3. red definicije.

Funkcija se poziva tako što se njeno ime (sa argumentima, ako ih ima) navede kao argument izraza, na mestu na kome se očekuje vrednost upravo onog tipa koji ima funkcija. Izraz u okviru kojega se poziva funkcija može biti proizvoljne složenosti (npr. i samo poziv funkcije jeste izraz), i može biti deo bilo kog iskaza čija sintaksa uključuje izraz (npr. izraz u iskazu dodele, logički izraz u uslovnom ili repetitivnom iskazu, itd).

Iz funkcije može da se izađe i pozivom funkcije **exit()** koja prekida izvršavanje programa i vraća vrednost svog parametra. Ova funkcija definisana je u standardnom zaglavlju <stdlib.h>.

Kako svaki izraz u C-u može postati iskaz tako što se završi sa ";", poziv funkcije može biti i samostalni iskaz, tzv. iskaz funkcije (u našem primeru to će biti svi pozivi funkcija u funkciji main).

10.2 Spoljašnje (globalne) i unutrašnje (lokalne) promenljive

Program na C-jeziku predstavlja skup *spoljašnjih* objekata – promenljivih i funkcija, koji su definisani izvan bilo koje funkcije i zato dostupni, u opštem slučaju, većem broju funkcija – svim funkcijama čija definicija sledi u istoj datoteci, ali i funkcijama u drugim datotekama, ispred čije definicije se deklariše definisani spoljašnji objekat. Sa druge strane, svaka funkcija može da ima promenljive definisane u telu definicije funkcije; to su *unutrašnje ili lokalne promenljive* (definicija funkcije, prema prvim standardima, ne može da sadrži definiciju druge funkcije, tj. sve funkcije su spoljašnje). Dakle, spoljašnje promenljive su glabalno dostupne, dok su unutrašnje (definisane na početku funkcije ili bloka) dostupne samo u okviru te funkcije odnosno bloka. Prostor za globalne promenljive (kao i imenovane konstante) rezerviše u delu memorije koji se naziva segment podataka i za njihovo trajanje (životni vek) kaže se da je statičko (statčki životni vek), dok za lokalne promenljive deo memorije u kome se rezerviše prostor zavisi od njihovog životnog veka.

Na primer, promenljive S, m koje se koriste u telu funkcije *promeni* jesu spoljašnje (globalne) promenljive. Pozivom funkcije *promeni* menjaju se vrednosti tih spoljašnjih promenljivih. I druge funkcije mogu da pristupaju ovim promenljivim i da im menjaju vrednosti, pa su spoljašnje promenljive jedan mehanizam (pored argumenata tj. parametara) za komunikaciju između funkcija.

Struktura datoteke programa za problem koji rešavamo može da ima sledeći oblik:

```
#include <stdio.h>
#define N 7
int m;
char S[N];
int dobar;
```

```
void promeni(void);      /* deklaracija funkcije promeni */
void proveri(void);      /* deklaracija funkcije proveri */
void produzi(void);      /* deklaracija funkcije produzi */
main()
{
    m=0; dobar=1;
    do {
        produzi();
        proveri();
        while (!dobar) {
            promeni();
            proveri();
        }
    }
    while(m!=N);
    for(m=1; m<=N; m++) printf("%c", S[m]);
}
void promeni(void)
{
  ⋮
}
void proveri(void)
{
  ⋮
}
void produzi(void)
{
  ⋮
}
```

U slučaju da se spoljašnje promenljive S, m, dobar, definišu posle definicije funkcije main, te promenljive biće nepoznate funkciji main, i program neće biti korektan.

U slučaju da se definicija funkcije main nalazi u jednoj datoteci, a definicije funkcija produzi, proveri, promeni – u drugoj, osim definicija spoljašnjih promenljivih S, m, dobar (i deklaracija funkcija produzi, promeni, proveri) ispred funkcije main u prvoj datoteci, u drugoj datoteci je neophodno navesti deklaraciju spoljašnjih promenljivih S, m, dobar, ispred definicije funkcija promeni, produzi, proveri, sa navedenom vrstom "extern" ili bez nje (ona se, zapravo, podrazumeva):

```
extern int m;
extern char S[ ];
extern int dobar;
```

Uz deklaracije nije dozvoljeno inicijalizovati promenljive (samo uz definicije!)

Dakle, jedan objekat (promenljiva, funkcija) sme se samo jednom definisati, ali se može više puta deklarisati.

Razmotrimo korišćenje unutrašnjih promenljivih na primeru funkcije proveri.

Funkcija *proveri* proverava da li konstruisani delimični niz dužine m zadovoljava kriterijum da su svaka dva susedna podniza različita, i to tako što se svojstvo različitosti proverava za susedne podnizove dužine 1, 2, ..., mpola=m / 2. Pri tome, ne proverava se različitost svih susednih podnizova već samo poslednja dva susedna podniza dužine 1,2, ..., mpola, jer je različitost svih ostalih podnizova odgovarajućih dužina utvrđena pre nego što je niz poslednji put produžen na dužinu m (niz se produžuje samo ako je "dobar").

Jednakost dva susedna podniza dužine d (d= 1, 2, ..., mpola) ispituje se tako što se porede parovi znakova na rastojanju d (za d=1, poredi se m-ti sa m-1-vim znakom; za d=2, porede se m-ti sa m-2-gim i m-1-vi sa m-3-ćim; za proizvoljno $1 \leq d \leq$ mpola, porede se m-ti sa m-d-tim, m-1-vi sa m-d-1-vim, m-2-gi sa m-d-2-gim, ..., m-d+1-vi sa m-2d+1-vim (tj. m-i-ti sa m-d-i-tim, za i=0, 1, ..., d-1). Ako se za par znakova S[m-i], S[m-d-i] ustanovi da su različiti, to je dovoljno da se utvrdi da su podnizovi dužine d različiti (tj. da je dobar==1).

Definicija funkcije *proveri* uključuje i promenljive (i definicije promenljivih) u telu funkcije, koje su od interesa za samu funkciju jer omogućuju izračunavanje koje funkcija sprovodi, ali nisu od interesa izvan same funkcije. Takve su pomenute promenljive mpola, d, i, i zato su one unutrašnje – *automatske* promenljive ove funkcije, koje se vide samo iz ove funkcije ali im je i trajanje (u memoriji) vezano za izvršavanje ove funkcije (samo koliko traje i izvršavanje ove funkcije).

Definicija funkcije *proveri* ima sledeći oblik:

```
void proveri(void)
{
 int i, d=0, mpola;
 dobar=1;
 d=0;
 mpola= m/2;
 while (dobar && (d < mpola))
 {
     d++; i=0;
     do { dobar = S[m-i]! =S[m-d-i];
        i++;
     } while(!dobar && (i<d))
 }
}
```

Radi kompletiranja programa navedimo i definiciju funkcije *produzi*:

```
void produzi(void)
{ m++; S[m] = '1';
}
```

Pozivom funkcije alocira se (dodeljuje) memorijski prostor na stek segmentu memorije za lokalne (automatske) promenljive te funkcije, u kome se čuvaju trenutne vrednosti tih promenljivih. Posle izvršenja funkcije ova memorija se oslobađa. Dakle, automatske promenljive postoje samo za vreme izvršavanja poziva funkcije, tj. za vreme "aktiviranja" funkcije. Svako aktiviranje – poziv funkcije uvodi svoje vlastite primerke automatskih promenljivih te funkcije, u sopstvenom tzv. stek okviru.

10.2.1 Statičke promenljive

Postoje situacije u kojima spoljašnje promenljive, iako treba da budu zajedničke (deljene, vidljive) za veći broj funkcija definisanih u istoj datoteci, ne treba da budu vidljive za funkcije istog programa definisane u drugim datotekama. Takve promenljive se deklarišu kao *statičke* (**static**). Isto važi i za funkcije, mada se na njih ređe primenjuje.

Svojstvo "statičnosti" ima primenu i na unutrašnje promenljive, sa drugačijim značenjem. Naime, ako se unutrašnja promenljia (definisana u okviru neke funkcije) deklariše kao **static** promenljiva, onda će se ona razlikovati od automatske promenljive u sledećem smislu: automatska promenljiva jedne funkcije (prostor i vrednost koji su joj dodeljeni) traje koliko i jedno izvršenje te funkcije; nova izvršavanja (pri novim pozivima) te funkcije kreiraju nove primerke automatskih promenljivih (sa novim prostorom i novom vrednošću)). Statička unutrašnja promenljiva zadržava vrednost i između raznih izvršavanja (pri različitim pozivima) te funkcije. Dakle, statička unutrašnja promenljiva ima statičko trajanje (statički životni vek) a automatska – automatski životni vek. Prostor za automatske i statičke lokalne promenljive rezerviše se na raznim segmentima memorije (na stek segmentu za automatske, na segmentu podataka za statičke). To treba imati u vidu pri definisanju "velikih" promenljivih (npr. nizova) kao lokalnih ili globalnih, s obzirom da je kapacitet steka obično ograničen.

10.2.2 Inicijalizacija

Mada C nije blokovski jezik u smislu definisanja funkcija unutar drugih funkcija, on ima mogućnost definisanja promenljivih u svakom bloku funkcije (blok je deo funkcije između para odgovarajućih vitičastih zagrada {, }). To su unutrašnje promenljive koje, slično unutrašnjim promenljivim na nivou funkcije, mogu biti automatske (što se podrazumeva) ili statičke (ako ima prethodi kvalifikator **static**). Automatska promenljiva na nivou bloka, kojoj je, u definiciji, dodeljena inicijalna vrednost (npr. int m=0;), dobija tu početnu vrednost pri svakom ulasku u blok, dok statička, inicijalizovana promenljiva na nivou bloka dobija početnu vrednost samo na početku izvršavanja programa.

Sve promenljive mogu da se inicijalizuju (da im se dodeli početna vrednost) eksplicitno svojom definicijom (npr. int m=0; ili int dani[ ] = { 31,28,31,30,31, 30,31,31,30,31,30,31}). Pri tome, početna vrednost globalne i statičke lokalne

promenljive mora da bude **konstantni izraz** čija se vrednost dodeljuje promenljivoj samo pre početka izvršavanja programa, dok početna vrednost automatske promenljive, s obzirom da joj se dodeljuje pri svakom pozivu funkcije ili ulasku u blok, može da uključi i promenljive (pa i pozive funkcija) sa prethodno dodeljenim vrednostima, na primer:

```
int binarpret(int x, int v[ ], int n)
{ int l =0;
  int d = n−1;
  int s = n/2;
  ...
}
```

Ako nema eksplicitne inicijalizacije, spoljašnje i statičke promenljive implicitno dobijaju početnu vrednost 0, dok je početna vrednost automatske promenljive nedefinisana.

10.2.3 Konflikt identifikatora

Svaki identifikator kome se ne vrši obraćanje izvan funkcije treba definisati kao unutrašnji – automatski identifikator funkcije. Ovi se identifikatori biraju nezavisno od spoljašnjih identifikatora pa se može dogoditi da je isti identifikator izabran i kao unutrašnji i kao spoljašnji identifikator funkcije. Ova se pojava zove **konflikt identifikatora** i razrešava se pravilima za vidljivost promenljive tj. tzv. prioritetom imena. Ilustrujmo je sledećim primerom.

Primer – konflikt identifikatora

Napisati funkciju **mult** koja za dva nenegativna cela broja x, y izračunava proizvod $z = x * y$ i program koji učitava cele brojeve x, y, u, poziva funkciju mult i izdaje vrednosti promenljivih x, y, u i rezultata z.

Prema algoritmu za množenje celih brojeva x, y iz tačke "Repetitivne strukture i razvoj algoritama" ($x = b_m * 2^{m-1} + b_{m-1} * 2^{m-2} + \cdots + b_1 * 2^0$, $b_i, i = 1, 2, \ldots, m \in \{0, 1\}$, $x * y = (b_m * 2^{m-1} + b_{m-1} * 2^{m-2} + \cdots + b_1 * 2^0) * y = b_m * 2^{m-1} * y + b_{m-1} * 2^{m-2} * y + \cdots + b_1 * 2^0 * y$), rezultat z izračunava se dodavanjem na međurezultat, koji je na početku 0, sabiraka oblika $b_i * 2^{i-1} * y$, za one sabirke za koje je $b_i = 1$.

Funkcija mult i program sada mogu da imaju sledeći oblik:

```
#include <stdio.h>
int x,y,u,z;
void mult(void);
main()
{
     scanf("%d, %d, %d", &x, &y, &u);       /* npr. x=5, y=7, u=10 */
     mult();
     printf("x=%d, y=%d, u=%d, z=%d", x, y, u, z);      /* x=5, y=7, u=10,
z=35 */
```

```
}
void mult(void)
{
 int u,v;
 u=x; v=y; z=0;
 while (u != 0) {
     if (u % 2) z=z+v;
     v*=2; u/=2;
 }
}
```

U programu postoje dva identifikatora *u*: jedan je definisan u okolini funkcije mult (izvan ijedne funkcije), a drugi je unutrašnji (lokalni) identifikator ove funkcije (definisan je unutar funkcije mult). Ova dva identifikatora označavaju dve različite promenljive: jedna je spoljašnja a druga unutrašnja za funkciju mult. U tom slučaju, kako se u funkciji mult kreira lokalna promenljiva *u* (i rezerviše memorijski prostor za nju na stek segmentu), svako pojavljivanje identifikatora *u* unutar funkcije mult odnosi se na lokalnu promenljivu, pa je za vreme izvršavanja funkcije mult spoljašnja promenljiva *u* nedostupna (nevidljiva). Izvan funkcije mult, identifikator *u* označava spoljašnju promenljivu (za nju je prostor rezarvisan na segemntu podataka). Ovo je mehanizam poznat kao *prioritet imena.*

10.3 Parametri

Prethodno definisana funkcija mult ima ograničene mogućnosti jer se može koristiti samo za množenje vrednosti dve fiksne promenljive, x i y, a rezultat je uvek pridružen promenljivoj z. Uopštenjem ovakvog oblika funkcije postiže se njena primenljivost na različite operande u različitim delovima programa. Promenljivi operandi nazivaju se **parametrima** (ili argumentima) funkcije.

U zaglavlju definicije funkcije navode se tipovi i identifikatori tzv. **formalnih parametara** funkcije koji predstavljaju opšta imena koja se zamenjuju, pri svakom pozivu funkcije, tzv. **stvarnim** (ili **aktuelnim**) parametrima (ili argumentima) (deklaracija funkcije treba da sadrži tipove parametara ali ne nužno i njihove identifikatore). Dakle, u svakom pozivu funkcije navode se aktuelni parametri, argumenti, objekti kojima se zamenjuju formalni parametri, i koji moraju biti istog tipa i u istom redosledu kao odgovarajući formalni parametri.

Formalni parametri funkcije nisu poznati izvan funkcije, pa se ponašaju kao unutrašnji objekti funkcije, definisani u samoj funkciji.

Parametri omogućuju mehanizam zamene koji dopušta ponavljanje procesa sa raznovrsnošću argumenata. Jedini mehanizam prenosa argumenata u C-u je (osim za nizove) tzv. supstitucija vrednosti (vrednosni parametar, parametar-vrednost): **vrednost** stvarnog parametra se izračunava i tom izračunatom vrednošću zamenjuje se odgovarajući formalni parametar; zato stvarni parametar može da bude

izraz. U slučaju da je parametar funkcije niz - prenosi se samo adresa njegovog prvog elementa, ne ceo niz.

Formalni parametar za koji je definisana supstitucija vrednosti ponaša se u pravom smislu kao unutrašnja promenljiva funkcije, ima automatski životni vek i za njega se, pri pozivu funkcije, rezerviše memorijski prostor dodeljen tom pozivu funkcije (na stek segmentu) i u taj prostor se smešta vrednost odgovarajućeg stvarnog parametra, pa vrednost odgovarajućeg stvarnog parametra, čak i kad je promenljiva, ostaje nepromenjena po izlasku iz funkcije (sa jednakom vrednošću kao i pri ulasku u funkciju).

Na primer, funkcija mult može se deklarisati tako da ima dva formalna parametra x i y a da se rezultat izračunavanja funkcije – proizvod argumenata – vraća iz funkcije kao vrednost pridružena njenom imenu:

```
#include <stdio.h>
int a,b,c,d;
int mult(int, int);
main()
{
    a=2; b=3; c=3;
    d=a+mult(b,c);      /* d=11 */
    printf("d=%d", d);
}
int mult(int x, int y);
{
    int z;
    z=0;
    while (x! =0) {
        if (x%2) z=z+y;
        y=2* y; x=x/2;
    }
    return z;
}
```

10.4 Direktive preprocesora C-a

Prvi korak u kompilaciji C-programa, koncepcijski, jeste preprocesiranje, tj. *prekompilacija* kojom se obrađuju neke od konstrukcija jezika, tzv. **direktive preprocesora**. Najčešće korišćene od tih mogućnosti jesu uključenje sadržaja navedene datoteke (#include) i zamena imena proizvoljnom niskom karaktera (#define). Direktiva #define može da se koristi i kao makro sa argumentima, a još jedna korisna direktiva odnosi se na uslovnu kompilaciju.

Datoteka čiji se sadržaj uključuje #include direktivom najčešće predstavlja kolekciju #define direktiva i deklaracija (između ostalog). Linija izvornog programa oblika

#include ”ime_datoteke”
ili
#include <ime_datoteke>
zamenjuje se, pri prvom prolazu, od strane preprocesora, sadržajem datoteke sa navedenim imenom.

Direktiva oblika
#define ime tekst_zamene
proizvodi jednostavnu zamenu svakog pojavljivanja imena ime navedenim tekstom zamene. Ime se gradi isto kao i identifikator, a tekst_zamene je proizvoljan. Ime definisano ovom direktivom važi do kraja datoteke koja se kompilira. Ova direktiva može da ima i parametre, tako da je tekst zamene različit za različite vrednosti parametara. Na primer, direktiva
#define max(A,B) ((A) > (B) ? (A) : (B))
u fazi prekompilacije proizvodi zamenu svakog pojavljivanja formalnih parametara A, B u tekstu zamene stvarnim parametrima sa kojima se navodi ime max u programu. Na primer, programski red
x=max(p+q, r+s);
u fazi prekompilacije)preprocesiranja) zamenjuje se redom
x=((p+q) > (r+s) ? (p+q) : (r+s));

Direktiva zamene #define zove se i *makro*. Osnovna prednost korišćenja makroa u odnosu na poziv funkcije je što se ne troši vreme i prostor za poziv funkcije.

Ime može da bude i ”razdefinisano”, najčešće da bi moglo da se koristi za druge potrebe, na primer:
#define getchar ’A’
#undef getchar

Uz pomoć uslovnog iskaza koji se izvršava u vreme preprocesiranja (uslovne direktive) može da se kontroliše i samo preprocesiranje. Takva je preprocesorska direktiva #if – #elif – #else – #endif, koja, slično if-iskazu C-a, izračunava vrednost celobrojnog izraza #if-linije i u slučaju da je vrednost različita od 0, uključuje sve linije do #endif (ili #elif, ili #else). Često se koristi da bi se izbeglo višestruko definisanje istog objekta u jednoj datoteci C-programa (jedinici prevođenja).

Pretpostavimo da jedna datoteka C-programa sadrži direktivu #include kojom se uključuje neka datoteka zaglavlja zagl.h i direktivu #define kojom se definiše ime SIGN. Ako i datoteka zagl.h sadrži definiciju imena SIGN, doći će do višestrukog definisanja imena SIGN što nije dozvoljeno. Da bi se izbeglo to višestruko definisanje, moguće je ”umetnuti” ga u uslovnu preprocesorsku direktivu oblika

#ifndef(SIGN)
#define SIGN –
#endif

Uslovna direktiva može da se koristi i za izbor datoteke koja se uključuje u zavisnosti od definisanosti nekog (navedenog) imena.

10.5 Pokazivači i argumenti funkcija

Pokazivač je promenljiva koja sadrži adresu neke (druge) promenljive. Korišćenje pokazivača je od posebnog značaja u C-u, jer se njima često postiže kompaktniji i efikasniji programski zapis. S druge strane, njihovo nekontrolisano korišćenje može da dovede do teško čitljivih a ponekad i potpuno nerazumljivih programa.

Memorija računara predstavlja niz memorijskih ćelija kojima se manipuliše pojedinačno ili u grupama (bajtovima, registrima). Bajtovi su numerisani susednim brojevima tj. *adresirani* su (adresama od 0 do max-1). U C-u, jedan bajt može da bude interpretiran kao karakter, par bajtova obično se interpertira kao kratki celi broj (short integer), a četiri susedna bajta – dugi celi broj (long). Pokazivač je grupa susednih bajtova (obično dva ili četiri) koja može da sadrži adresu.

Unarni operator *referenciranja* – & – daje adresu objekta – operanda na koji se primenjuje. Tako, ako je c promenljiva tipa char a p – pokazivač koji pokazuje na karakter, iskaz

```
p = &c;
```

dodeljuje promenljivoj p adresu promenljive c, i kaže se da p "pokazuje na" c.

Operator referenciranja & može da se primeni samo na objekte sa adresom, tj. na promenljive i elemente nizova, i ne može da se primeni, na primer, na izraze ili konstante.

U izvesnom smislu inverzni operator operatora referenciranja, operator *dereferenciranja* ili *indirekcije* ∗, primenjuje se na pokazivač i daje sâm objekat na koji pokazivač pokazuje. Zato je i deklaracija pokazivačke promenljive označena tipom na koji pokazivač pokazuje, tj. pokazivač ip koji može da pokazuje na promenljivu celobrojnog tipa deklariše se sa:

```
int *ip;
```

a pokazivač p koji pokazuje na promenljivu znakovnog tipa deklariše se sa

```
char *p;
```

Dakle, ip je pokazivač (adresa) celobrojne promenljive; kada ip dobije konkretnu vrednost (adresu konkretne celobrojne promenljive), ∗ip će označavati sâmu tu celobrojnu promenljivu. Slično važi za pokazivač p.

Ilustrujmo algebru pokazivača sledećim primerom (promenljive x, y, z su celobrojne promenljive, a ip – pokazivač na promenljivu celobrojnog tipa:

```
int x=1, y=2, z[10];
int *ip;      /* ip je pokazivač na int */
ip = &x;      /*  ip sada pokazuje na x  */
y = *ip;      /*  y sada ima vrednost 1  */
*ip = 0;      /*  x sada ima vrednost 0  */
ip = &z[0];      /*  ip sada pokazuje na z[0] */
```

Celobrojna promenljiva na koju pokazuje prethodno deklarisana pokazivačka promenljiva ip može da se pojavi gde god se očekuje celobrojna promenljiva, na primer

∗ip = ∗ip + 10; /∗ uvećava ∗ip za 10 ∗ /,

y = ∗ip +1; /∗ vrednost promenljive na koju pokazuje ip, uvećana za 1, dodeljuje se promenljivoj y ∗/,

∗ip += 1; /∗ kao i ++∗ip; i (∗ip)++; povećava vrednost promenljive na koju pokazuje ip ∗/.

Pokazivačima se mogu razmenjivati vrednosti (adrese) i bez dereferenciranja (obraćanja objektima na koje pokazivači pokazuju), na primer:

iq = ip;

(pokazivač iq sada pokazuje na ono na šta pokazuje ip).

Već smo napomenuli da C prenosi argumente funkcija po vrednosti, tj. ne omogućuje direktnu promenu vrednosti argumenata od strane pozvane funkcije. Na primer, funkcija za razmenu vrednosti a, b (swap):

```
void swap(int x, int y)
{
    int temp;
    temp =x;
    x = y;
    y = temp;
}
```

i njen poziv

swap(a, b);

neće razmeniti vrednosti argumenata a i b, jer, zbog prenosa argumenata po vrednosti, funkcija swap razmenjuje samo sopstvene kopije promenljivih a i b (iz svog "stek okvira"), a ne i vrednosti tih promenljivih u funkciji iz koje se funkcija swap poziva.

Željeni efekat može se postići prenosom *pokazivača* na argumente umesto samih argumenata, tj. pozivom funkcije swap(&a, &b) definisane sledećom definicijom (argumenti funkcije su *pokazivači* na celobrojne promenljive):

```
void swap(int *px, int *py)      /* razmena *px i *py */
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

Grafički (slika 10.1),

Za prethodno definisanu funkciju deklaracija (prototip) bi bila oblika void swap(int ∗, int ∗), a poziv swap(&a, &b).

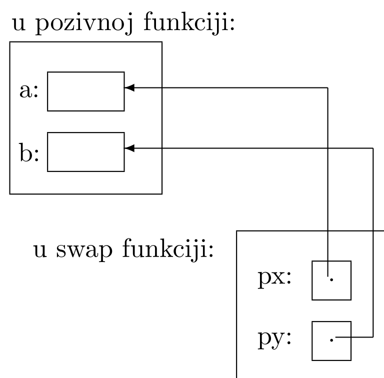

Figure 10.1: Pokazivači na argumente

10.6 Biblioteke funkcija

Neke od često korišćenih C-biblioteka funkcija su biblioteke sa standardnim zaglavljima <stdio.h>, <ctype.h>, <math.h>, <stdlib.h>, <string.h>, itd. U standardnom zaglavlju <stdio.h> deklarisane su funkcije ulaza / izlaza, npr. printf, scanf, sscanf, sprintf, getchar, putchar, itd. Neke od funkcija deklarisanih u standardnom zaglavlju <ctype.h> su funkcije ispitivanja klase karaktera - isalnum(c), isalpha(c), iscntrl(c), isdigit(c), islower(c), itd. U standardnom zaglavlju <math.h> nalaze se prototipovi elementarnih funkcija sin, cos, tan, exp, sqrt, zatim ceil, floor itd. U standardnom zaglavlju <stdlib.h> nalaze se prototipovi funkcija za konverziju tipa (npr. atof, atoi), zatim funkcija za alokaciju memorije (malloc, calloc, free) itd.

11

Nizovi

Nizovski tip podataka odlikuje se strukturnim vrednostima. Svaka vrednost nizovskog tipa – niz – jeste kolekcija **komponenti istog tipa**. Dakle, nizovski tip je strukturni tip. Nizovska struktura karakteriše se i sledećim svojstvima:

1. svaka pojedinačna komponenta, tzv. **element niza** je direktno dostupna i eksplicitno se označava;

2. broj komponenti definiše se kada se uvodi niz.

Komponenta se označava imenom niza i tzv. **indeksom** koji jednoznačno određuje željeni element. Indeks je **izračunljivi** objekat – izraz, i to razlikuje niz od drugih strukturnih tipova. Indeks u C-u je celobrojnog tipa i svaki niz obavezno počinje od indeksa 0.

Definicija niza uključuje **komponentni** tip i broj komponenti (maksimalnu vrednost indeksa).

U sintaksi C-a, na primer, definicija niza je oblika:

$\mathrm{T}_1\ a[\mathrm{dim}]$

gde je T_1 – komponentni tip, a je ime niza a dim je maksimalni broj elemenata niza a.

Komponenta niza a koja odgovara indeksnoj vrednosti i obeležava se sa $a[i]$. Na primer, $a[10]$, $a[i+j]$.

11.1 Nizovi karaktera - niske, stringovi

Najčešće korišćeni nizovi su nizovi karaktera tj. stringovi, niske. S obzirom da se nizovi sa elementima istog tipa, bez obzira na broj elemenata, u C-u mogu smatrati objektima istog tipa, postoji i razvijena biblioteka funkcija za rad sa stringovima proizvoljne dužine (deklaracije u zaglavlju <string.h>).

U ovoj tački prikazaćemo neke primere funkcija i njihovih mogućih implementacija, kao i programa za rad sa stringovima.

Primer 1 Funkcija konverzije stringa (koji se sastoji od karaktera – cifara) u ceo broj:

```
int atoi(char s[ ])
{
      int i, n;
      n=0;
      for (i=0; s[i]>='0' && s[i] <='9'; ++i)
            n=10*n + (s[i] -'0');
      return n;
}
```

Primer 2 – funkcija za obrtanje (inverziju) karaktera stringa

```
void obrni(char s[ ])
{
      int c, i, j;
      for(i = 0, j = strlen(s)–1; i < j; i++, j––) {
            c = s[i];
            s[i] = s[j];
            s[j] = c;
      }
}
```

Primer 3 – funkcija konverzije celog broja u string

```
void itoa(int n, char s[ ])
{
      int i, znak;
      if((znak=n) < 0) /* registrovanje znaka */
            n = -n; /* n postaje pozitivno */
      i = 0;
      do { /* generisanje cifara u obrnutom poretku */
            s[i++] = n% 10 + '0'; /* sledeca cifra */
      } while ((n /= 10) > 0); /* obrisati je */
      if (znak < 0)
            s[i++] = '–';
      s[i] = '\0';
      obrni(s);
}
```

Primer 4 – funkcija za izračunavanje dužine stringa

```
/* strlen: vraća dužinu stringa s */
int strlen(char s[])
{
      int i;
```

```
    for (i=0; s[i]!= '\0'; i++);
    return i;
}
```

Primer 5. Ako je potrebno izvršavati operacije nad "dugačkim" brojevima (koji imaju po n cifara, npr. za n=100), oni mogu izaći iz dijapazona celih (pa i dugačkih celih) brojeva u C-u. Jedan način da se sabiraju takvi brojevi jeste da se imitira "pismeno" sabiranje – zdesna ulevo, a da se i cifre zbira predstavljaju kao znakovi (karakteri). Promenljive cniz1 i cniz2 neka su nizovi karaktera koji predstavljaju cifre ulaznih brojeva; promenljiva cniz3 neka je niz karaktera koji predstavljaju cifre broja – zbira. Celobrojna promenljiva prenos neka pamti prenos (0 ili 1) sa mesta prethodne – manje težine, a celobrojna promenljiva zbirc neka predstavlja zbir cifara na mestu iste težine u ulaznim brojevima i prenosa sa mesta manje težine. Program na C-u sada može imati sledeći oblik:

```
#include <stdio.h>
char cniz1[100], cniz2[100], cniz3[100];
int prenos=0, n, zbirc, i;
main()
{
    scanf("%d", &n);
    scanf("%s", cniz1);
    scanf("%s", cniz2);
    for(i=n-1; i>= 0; i--)
    {
        zbirc = (cniz1[i]) - '0' + cniz2[i] - '0' + prenos;
        prenos = zbirc / 10;
        cniz3[n-i-1] = zbirc % 10 + '0';
    }
    /* ako je prenos sa mesta najvece tezine = 1, zbir ima jedno mesto vise od
sabiraka */
    if (prenos > 0)
    {
        cniz3[n] = prenos + '0';
        printf("%c", cniz3[n]);
    }
    /* izdavanje cifara zbira */
    for(i=n-1; i>=0; i--;) printf("%c", cniz3[i]);
    printf("\n");
}
```

11.2 Pokazivači i nizovi

U C-u su pokazivači i nizovi tako tesno povezani da zaslužuju istovremeni prikaz. Sve što može da se uradi korišćenjem indeksiranih elemenata niza, može i

korišćenjem pokazivača.

Na primer, deklaracija

int a[10];

definiše niz od 10 elemenata, tj. rezerviše 10 susednih objekata nazvanih a[0], a[1], . . . , a[9].

Oznaka a[i] označava i-ti element niza. Ako je pa pokazivač na ceo broj, deklarisan sa

int ∗pa;

tada iskaz dodele

pa = &a[0];

dodeljuje promenljivoj pa pokazivač na nulti element niza a, tj. pa sadrži adresu elementa a[0].

Grafički (slika 11.1):

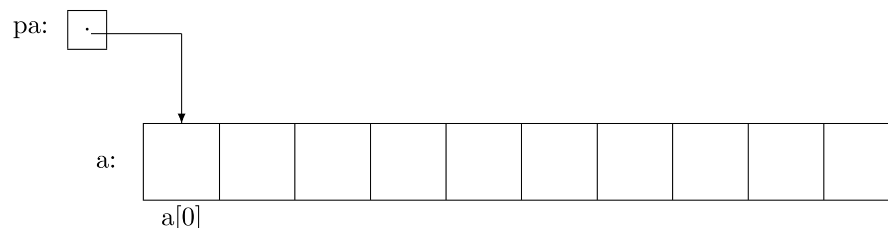

Figure 11.1: Pokazivač na niz

Sada dodela

x = ∗pa;

kopira sadržaj elementa a[0] u promenljivu x.

Ako pokazivač pa pokazuje na neki određeni element niza, onda, po definiciji, pa+1 pokazuje na sledeći element a pa-1 na prethodni element niza. Ako pa pokazuje na element a[0], onda je pa+i – *adresa* elementa a[i] a ∗(pa+i) je *sadržaj* elementa a[i]. Ova aritmetika nad pokazivačima (adresama) važi bez obzira na tip objekta – elementa niza.

Kako je ime niza – lokacija tj. adresa njegovog početnog elementa, dodela

pa = &a[0]; (pokazivač dobija adresu nultog elementa niza),

ima ekvivalentno značenje kao i dodela

pa = a;

Šta više, a+i (adresa i-tog elementa) je isto što i &a[i], pa je i ∗(a+i) jednak a[i], i to je upravo način na koji se pristupa sadržaju elementa a[i] u C-u. Slično, ako je pa pokazivač, onda je ∗(pa+i) isto što i pa[i].

Ipak, bitna razlika između pokazivača i imena niza je u tome što je pokazivač – promenljiva, pa ima smisla izraz pa=a ili pa++, dok ime niza to nije, pa nema smisla izraz oblika a=pa ili a++. Dakle, pokazivač pa jeste l-vrednost, dok ime niza a to nije.

11.3 Funkcije i pokazivači na karaktere

Kada se ime niza prenosi kao argument funkcije, prenosi se zapravo adresa početnog elementa. Ovaj argument je u pozvanoj funkciji lokalna promenljiva koja sadrži adresu, pa se nad tom promenljivom može sprovesti aritmertika nad adresama. Na primer, još jedna definicija funkcije strlen (koja izračunava dužinu niske):

```
/* strlen: vraća dužinu stringa s */
int strlen(char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return n;
}
```

Uvećanje pokazivača (adrese), s++, predstavlja ispravnu operaciju koja se izvršava nad kopijom pokazivača s u stek okviru poziva funkcije strlen, i ni na koji način se ne odražava na nisku karaktera (string) u funkciji koja poziva funkciju strlen.

Umesto formalnog parametra u definiciji funkcije, char *s; može da se koristi i ekvivalentni zapis char s[ ];

Poziv funkcije strlen može, kao stvarni argument da ima pokazivač na znakovnu promenljivu (tj. pokazivač na početni element stringa),

```
strlen(ptr);   /* char *ptr */,
```

ime niza (adresu početnog elementa niza),

```
strlen(array); /* char array [100] */,
```

ili konstantu tipa niske,

```
strlen("zdravo"); /* niskovna konstanta */
```

Dakle, C je pravilan i dosledan u svom tretmanu aritmetike adresa. On zapravo integriše aritmetiku pokazivača, nizova i adresa.

Najčešći oblik pojavljivanja niskovne konstante (literala, konstante tipa string), jeste argument funkcije, kao u prethodnom primeru: strlen("zdravo"); ili printf("zdravo\n"). Funkcija printf (kao i strlen) dobija kao argumant pokazivač na početak niza karaktera. Dakle, niskovnoj konstanti pristupa se preko pokazivača na njen prvi element.

Ako se pokazivaču dodeli niskovna konstanta, onda se pokazivaču, zapravo, dodeljuje adresa prvog elementa niza karaktera koji čine niskovnu konstantu. Na primer,

```
char *pporuka;
pporuka = "ovo je poruka";
```

Poslednja dodela ne predstavlja kopiranje nizova karaktera, već uključuje samo rad sa pokazivačima.

Još jedna razlika u radu sa nizovima i pokazivačima može se ilustrovati sledećim primerom:

```
char nporuka[ ] = ”ovo je poruka”;      /∗ niz ∗/
char ∗pporuka = ”ovo je poruka”;      /∗ pokazivač ∗/
```

nporuka je niz koji sadrži dodeljenu nisku karaktera. Neki od tih karaktera mogu i da se promene (npr. nporuka[1] = ’n’), ali će nporuka i dalje da se odnosi (i pokazuje) na isti memorijski prostor. S druge strane, pporuka je pokazivač koji pokazuje na niskovnu konstantu; pokazivač može da se ”premesti” (da pokazuje na nešto drugo), ali se sadržaj niskovne konstante ne može promeniti (a da rezultat te operacije bude definisan). Dakle, element niza nporuka[i] jeste l-vrednost dok karakter ∗p to nije. Grafički (slika 11.2),

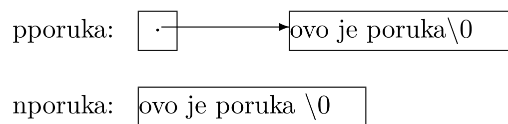

Figure 11.2: Niske karaktera i pokazivači

Razmotrimo odnos između pokazivača, nizova i funkcija na primeru varijanti dve funkcije (njihovi prototipovi, kao i niz drugih funkcija za rad sa stringovima, postoje i u standardnoj biblioteci <string.h>).

Funkcija strcpy(s,t) treba da kopira string t u string s. Dodela s=t ne obavlja zadatak jer se odnosi na pokazivače (pokazivaču s dodeljuje se vrednost pokazivača t, tj. pokazivač s pokazuje na prvi element stringa t), i njom se ne vrši željeno kopiranje stringova. Nad stringovima kao celinama (jedinstvenim objektima) se inače ne može obavljati ni jedna operacija, već samo nad pojedinačnim elementima. Dakle, i kopiranje mora da se izvrši u ciklusu po elementima stringova s, t. Prototip standardne funkcije strcpy u standardnom zaglavlju <string.h> ima oblik char ∗ strcpy(char ∗,const char ∗). Jedna implementacija može da ima sledeć oblik:

```
/∗ strcpy: kopira t u s; verzija sa indeksiranim nizovima ∗/
char ∗ strcpy(char ∗s, const char ∗t)      // parametri mogu biti i char s[ ], const
char t[ ]
{
    int i;
    i = 0;
    while ((s[i] = t[i]) != ’\0’)
        i++;
    return s;
}
```

Druga implementacija može da ima sledeći oblik:

```
/∗ strcpy: kopira t u s; verzija sa pokazivacima (1) ∗/
char ∗ strcpy(char ∗s, const char ∗t)
{
    while ((∗s = ∗t) != ’\0’) {
```

```
        s++;
        t++;
    }
    return s;
}
```

Funkcija strcpy (verzija sa pokazivačima) koristi parametre s, t onako kako joj odgovara – menja im vrednosti, ali se to ne odražava na odgovarajuće pokazivače u funkciji iz koje se strcpy poziva – ti pokazivači će, po povratku iz funkcije strcpy, i dalje pokazivati na početne elemente stringova s odnosno t. To je posledica činjenice da se argumenti prenose po vrednosti (ne po imenu).

"Iskusnija" verzija funkcije strcpy sa pokazivačima uključuje "pomeranje" pokazivača u while-test, i definicija funkcije dobija oblik:

```
/* strcpy: kopira t u s; verzija sa pokazivacima (2) */
char * strcpy(char *s, const char *t)
{
    while ((*s++ = *t++) != '\0');
    return ;
}
```

I test različitosti od '\0' (nule) može da se eliminiše jer while iskaz inače ispituje različitost izraza od nule. Zato funkcija može da dobije konačni oblik:

```
/* strcpy: kopira t u s; verzija sa pokazivacima (3) */
char * strcpy(char *s, const char *t)
{
    while (*s++ = *t++);
    return s;
}
```

Pored funkcije strcpy, u istom zaglavlju deklarisana je i funkcija strcnpy koja kopira najviše n karaktera drugog stringa u prvi:

```
char * strncpy(char * s1, const char * s2, size_t n);
```

Ako string s2 ima manje od n karaktera, u string s1 upisuje se na odgovarajućoj dužini nula-karakter ('\0').

Drugi primer je funkcija koja poredi dve niske karaktera s i t, i vraća negativnu vrednost, nulu ili pozitivnu vrednost, u zavisnosti od leksikografskog poretka niski s i t. Prototip ove standardne funkcije u standardnom zaglavlju <string.h> je oblika int strcmp(const char *, const char *).

Jedna implementacija funkcije strcmp:

```
/* strcmp: vraca <0 ako je s<t, 0 ako je s==t, >0 ako je s>t */
int strcmp(char *s, char *t)
{
    int i;
    for (i = 0; s[i] == t[i]; i++)
```

```
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

Druga, pokazivačka verzija funkcije strcmp ima sledeću definiciju:

```
/* strcmp: vraca <0 ako je s<t, 0 ako je s==t, >0 ako je s>t */
int strcmp(char *s, char *t)
{
    int i;
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

11.4 Višedimenzioni nizovi

Višedimenzioni nizovi u C-u koriste se mnogo manje nego nizovi pokazivača (v. tačku 11.2). Na primer, definicija

```
char dani[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
};
```

definiše tabelu (dvodimenzioni niz) čiji je prvi red – niz brojeva dana u mesecima u neprestupnim godinama, a drugi red – niz brojeva dana u mesecima u prestupnim godinama. Dvodimenzioni niz je niz čiji su elementi nizovi, pa se elementu dvodimenzionog niza pristupa navođenjem indeksiranog elementa (kolone) indeksiranog elementa (vrste) dvodimenzionog niza (npr. dani[0][2] je broj dana u februaru (2) neprestupne (0) godine (28), a dani[1][10] jeste broj dana u oktobru (10) prestupne (1) godine (31)).

Slično, definicije

```
char (* dani)[13]
```

odnosno

```
char * * dani
```

definišu pokazivač na niz celih brojeva, odnosno pokazivač na pokazivač na (prvi u nizu) ceo broj, pri čemu u prvom slučaju niz (tj. svaka vrsta matrice) ima po 13 elemenata, a u drugom slučju taj broj može da varira od vrste do vrste. Razlika u odnosu na prvu definiciju je u tome što se drugim dvema definicijama alocira memorijski prostor samo za pokazivač na niz odnosno na pokazivač a ne i za elemente tog niza.

Ako je dvodimenzioni niz argument funkcije, mora se navesti broj kolona, dok je broj vrsta nebitan – ime dvodimenzionog niza je pokazivač na niz vrsta od kojih

svaka sadrži onoliko elemenata koliki je navedeni broj kolona. Na primer, niz dani bi se kao argument funkcije naveo u obliku
f(char dani[2][13]){ ...}, ili
f(char dani[ ][13]){ ...}, ili
f(char (*dani)[13]){ ...}

11.5 Razvoj algoritama sa nizovima

11.5.1 Konverzija broja iz dekadnog u binarni oblik

Konstruisati algoritam koji konvertuje dekadni prirodni broj n u binarni broj predstavljen nizom b.

Kao što je vrednost dekadnog celog nenegativnog (prirodnog) broja n predstavljenog nizom dekadnih cifara $a_i, a_{i-1}, \ldots, a_1$ ($a_i, a_{i-1}, \ldots, a_1 \in \{0,1,2,3,4,5,6,7,8,9\}$) jednaka $a_i \times 10^{i-1} + a_{i-1} \times 10^{i-2} + \cdots + a_1 \times 10^0$, tako je i vrednost binarnog celog nenegativnog broja predstavljenog nizom binarnih cifara $b_j, b_{j-1}, \ldots, b_1$ ($b_j, b_{j-1}, \ldots, b_1 \in \{0,1\}$) jednaka $b_j \times 2^{j-1} + b_{j-1} \times 2^{j-2} + \cdots + b_1 \times 2^0$. Da bi se izvršila potrebna konverzija, potrebno je, za zadati niz dekadnih cifara $a_i, a_{i-1}, \ldots, a_1$ odrediti niz binarnih cifara $b_j, b_{j-1}, \ldots, b_1$ takvih da je $a_i \times 10^{i-1} + a_{i-1} \times 10^{i-2} + \cdots + a_1 \times 10^0 = b_j \times 2^{j-1} + b_{j-1} \times 2^{j-2} + \cdots + b_1 \times 2^0$. Postupak za određivanje binarnih cifara sličan je postupku za određivanje dekadnih cifara i biće ilustrovan prvo na jednom primeru.

Za dekadni broj 193, cifra jedinica (3) određuje se kao ostatak pri celobrojnom deljenju broja 193 osnovom 10, tj. 193:10=19(3), cifra desetica (9) kao ostatak pri celobrojnom deljenju količnika 19 osnovom 10, tj. 19:10=1(9), a cifra stotina kao ostatak pri celobrojnom deljenju količnika 1 osnovom 10, tj. 1:10=0(1). Slično se izračunavaju i binarne cifre (broja u binarnom sistemu), samo se sada deli osnovom 2.

$193 = [193/2] \times 2 + 1 = \mathbf{96} \times 2 + 1$ ($b_1 = 1$)
(Oznaka [193/2] označava celobrojni deo količnika 193/2).
$96 = [96/2] \times 2 + 0 = \mathbf{48} \times 2 + 0$ ($b_2 = 0$)
$48 = [48/2] \times 2 + 0 = \mathbf{24} \times 2 + 0$ ($b_3 = 0$)
$24 = [24/2] \times 2 + 0 = \mathbf{12} \times 2 + 0$ ($b_4 = 0$)
$12 = [12/2] \times 2 + 0 = \mathbf{6} \times 2 + 0$ ($b_5 = 0$)
$6 = [6/2] \times 2 + 0 = \mathbf{3} \times 2 + 0$ ($b_6 = 0$)
$3 = [3/2] \times 2 + 1 = \mathbf{1} \times 2 + 1$ ($b_7 = 1$)
$1 = [1/2] \times 2 + 1 = \mathbf{0} \times 2 + 1$ ($b_8 = 1$)

Dakle, $(193)_{10} = (11000001)_2$, jer je $1 \times 10^2 + 9 \times 10^1 + 3 \times 10^0 = 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$.

U opštem slučaju algoritam konverzije dekadnog prirodnog broja n u binarni broj predstavljen nizom binarnih cifara $b_k, b_{k-1}, \ldots b_1$ ima sledeći oblik:

```
{
    t = n;   /*  vrednost broja n pridružuje se tekućoj promenljivoj t  */
```

```
    k = 0;   /*  broj izračunatih binarnih cifara  */
    while (t > 0)
    {
        k = k + 1;
        b[k] = t%2;
        t = t/2;
}
```

11.5.2 Efikasnost algoritama

Efikasnost algoritma može da se meri vremenom njegovog izvršavanja, tj. brojem operacija pri njegovom izvršavanju (vremenska efikasnost) ili zauzećem meorije (prostorna efikasnost), kao funkcijama dimenzije problema koji algoritam rešava. Dimenzija problema je obično količina podataka koje treba obraditi. U algoritmima pretraživanja ili sortiranja (koji slede) to će biti broj elemenata koje treba pretražiti ili sortirati, n. Pri analizi vremenske složenosti algoritma zainteresovani smo, kako za prosečni slučaj (količinu vremena koje se utroši pri "tipičnim" ulaznim podacima), tako i za najgori slučaj (vreme potrebno za rešavanje problema pri najgoroj mogućoj ulaznoj konfiguraciji).

Prvi korak u dobijanju grube procene vremena izvršavanja programa tj. algoritma je identifikovanje unutrašnjih petlji. Koji se iskazi programa tj. koraci algoritma izvršavaju najčešće? Korisno je da programer identifikuje unutrašnje cikluse jer se gotovo sve vreme izvršavanja programa potroši u njima. Iskaze za koje nije neophodno da budu u ciklusu treba iz ciklusa i eliminisati. Drugo, potrebno je izvršiti izvesnu analizu da bi se procenilo koliko se puta unutrašnje petlje izvršavaju. Ta analiza ne mora biti precizna (da prebroji svaki korak u izvršavanju algoritma), već je dovoljno da odredi najveće elemente takvih proračuna.

Vreme izvršavanja nekog programa tj. algoritma je obično neka konstanta puta jedan od izraza: 1, $\ln n$, n, $n \ln n$, n^2, n^3, 2^n, 3^n i sl. (*glavni izraz*), plus neki "manji" izraz (nižeg reda veličine). Kaže se da je vreme izvršavanja algoritma (programa) *proporcionalno* glavnom izrazu. Pri velikom n, efekat glavnog izraza dominira. Za mala n, i manji izrazi mogu značajno da doprinesu ukupnom vremenu izvršavanja, pa je analiza složenija. Zato se pri formulama složenosti koje se izvode podrazumeva veliko n.

11.6 Pretraživanje nizova

11.6.1 Linearno pretraživanje

Dat je fiksirani niz $b[1..m]$ gde je $0 < m$. Zna se da je fiksirana vrednost x u nizu $b[1..m]$. Potrebno je da se razvije program koji određuje prvo pojavljivanje x-a u nizu b, tj. naći vrednost promenljive i koja je najmanji ceo broj t.d. $x = b[i]$.

Program ima sledeću strukturu:

$i = 1$; while $(x! = b[i])\ i + +$;

Program može da se uopšti na pronalaženje prvog u nizu elementa sa bilo kojim zadatim svojstvom, što je poznato kao:

Princip (linearnog pretraživanja): da bi se našla **najmanja** vrednost (eventualno ograničena odozdo) sa **datim svojstvom** U (bilo kojim), ispitivati vrednosti počevši od te donje granice, u **rastućem** poretku. Slično, kada se traži **najveća** vrednost, ispitivati vrednosti u opadajućem poretku.

U našem primeru linearnog pretraživanja, **dato svojstvo** U najmanje vrednosti bilo je jednakost elementa sa tim (najmanjim) indeksom i zadate vrednosti x, tj. $U(i) = (b[i] == x)$. To svojstvo moglo je biti, na primer, i $U(i) = (b[i] \geq 100)$, tj. naći element niza b sa najmanjim indeksom, koji je ≥ 100, i sl.

Zadatak pretraživanja niza može se uopštiti tako što se ukine pretpostavka da je tražena vrednost x sadržana u nizu b.

Transformacijom while u do-while iskaz, i zapisom u C-notaciji, funkcija za linearno pretraživanje niza celih brojeva sada dobija sledeći oblik:

```
/* linsearch: nalazi vrednost x u nizu b[1], ..., b[m] */
int linsearch(int x, int b[ ], int m)
{
    int i=0;
    do i++;   /*   (b[j] != x za j=1..i–1   */
    while(i<=m && b[i]! = x);
    if (i==m+1) return –1;
    else return i;
}   /*   (b[j] != x za j=1..i–1 && b[i] == x)   */
                        /*   || b[j] != x za j=1..m   */
```

Kompletnije rešenje prethodnog zadatka, pod pretpostavkom da vrednost x ne mora biti jednaka nekom elementu niza b, dobija se tako što se ta vrednost x dodeli novom elementu niza b, $b[m + 1]$, i tako rešenje svede na polazni slučaj. Sada će se vrednost x svakako naći među elementima niza b. Ako je nađena tek na $m + 1$-vom mestu, znači da nije bila u polaznom nizu b; ako se nađe ranije, znači da je tamo i bila.

```
/* linsearch: nalazi vrednost x u nizu b[1], ..., b[m]; */
/* niz b se prosiruje m+1.vim clanom jednakim x */
int linsearch(int x, int b[ ], int m)
{
    int i=0;
    b[m+1] = x;
    do i++;   /*   (b[j] != x za j=1..i–1   */
    while(b[i]! = x);   /*   (b[j] != x za j=1..i–1 && b[i] == x)   */
    return i;
}
```

Ako uslov $b[i]$ == x obeležimo sa $U(i)$, onda se algoritam linearnog pretraživanja može uopštiti na proizvoljni uslov $U(i)$, tako što se poređenje $b[i] == x$ svuda zameni uslovom $U(i)$ (a $b[i]! = x$ sa $\neg U(i)$).

Linearno pretraživanje ima vremensku složenost proporcionalnu sa m – brojem elemenata niza b.

11.6.2 Binarno pretraživanje

Ako je niz b uređen, imamo brži algoritam za pretraživanje niza. Pretpostavlja se da su sve vrednosti elemenata niza b različite, a algoritam treba da utvrdi da li se tražena vrednost x nalazi u nizu b ili ne. Tražena vrednost, x, poredi se sa vrednošću središnjeg elementa niza, $b[(m+1)/2]$. Ako je jednaka, nalazi se u nizu (na mestu $(m+1)$ / 2). Ako je $x < b[(m+1)/\ 2]$, može se nastaviti sa pretraživanjem samo leve polovine niza b, a ako je $x > b[(m+1)/\ 2]$, može se nastaviti sa pretraživanjem samo desne polovine niza b. Sličan postupak se nastavlja sve dok se ne nađe element koji je jednak x, ili dok interval pretraživanja ne postane prazan (leva granica veća od desne). To je algoritam **binarnog pretraživanja**. Ako sa l obeležimo levu granicu intervala koji se pretražuje, sa d – desnu, a sa s – sredinu, funkcija binarnog pretraživanja niza celih brojeva ima sledeći oblik:

```
/* binsearch: nalazi vrednost x u nizu b[1], . . . , b[m]; */
int binsearch(int x, int b[ ], int m)
{
    int l=1, d=m, s;
    while (l <= d) {
        s = (l + d) / 2;
        if (x < b[s]) d = s − 1;
        else if (x > b[s]) l = s + 1;
        else return s;
    }
    return −1;
}
```

Vremenska složenost algoritma binarnog pretraživanja je log$m$, i izvodi se na sledeći način:

Ako sa $S(m)$ obeležimo broj potrebnih operacija da se pretraži niz od m elemenata, onda je

$S(m) = 1 + S(m/2)$.

Ako za m uzmemo samo brojeve oblika 2^M, onda za takve brojeve važi

$S(2^M) = 1 + S(2^{M-1}) = 2 + S(2^{M-2}) = \cdots = M + S(2^0) = M + 1$.

Imamo da je $M = \log_2 m$, tj. $S(m) \approx \log_2 m$, što je funkcija koja mnogo sporije raste od m.

Kako je $\log_e m \approx \log_2 m$, to je i $S(m) \approx \ln m$.

Inače, ako je vremenska složenost logaritamska, onda i nije od značaja za koju je osnovu taj algoritam, jer se ocene razlikuju do na konstantu ($\log_a m = \log_b m \times \log_a b$).

Logaritamska funkcija mnogo sporije raste od linearne, pa su i algoritmi sa logaritamskom vremenskom složenošću mnogo efikasniji od onih sa linearnom složenošću. Kada se dimenzija problema, m, udvostruči, složenost se uveća samo za 1 ($\ln 2m = \ln 2 + \ln m \approx \ln m + 1$), a kada se m kvadrira, $\ln m$ se tek udvostruči (na primer, za $m = 10$, $\ln m \approx 3$, a za $m = 10^2 = 100$, $\ln m \approx 6$.

11.6.3 Interpolirano pretraživanje

Jedno poboljšanje binarnog pretraživanja je *interpolirano pretraživanje* koje odgovara pretraživanju rečnika: za reč na B ne traži se sredina rečnika već deo bliži početku. Zato se umesto s=(l+d)/2 (≈ s= l+(1/2)*(d-l)), može koristiti, npr. s=l+(x-b[l])*(d-l) / (b[d] - b[l]) .

Na primer, za niz od 18 elemenata koji uzimaju sledeće vrednosti iz skupa slova naše abecede:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P	R	I	M	E	R	P	R	E	T	R	A	Ž	I	V	A	NJ	A

sortirani niz će biti:

A A A E E I I M NJ P P R R R R T V Ž

Neka se traži element sa vrednošću V. Binarno pretraživanje počinje poređenjem slova V sa vrednošću elementa u sredini niza (podvučeni element),

A A A E E I I M <u>NJ</u> P P R R R R T V Ž

pa pošto je V "iza" slova u sredini (NJ), pretraživanje se nastavlja, primenom istog postupka, u desnoj polovini niza, tj.

P P R R <u>R</u> R T V Ž

R <u>T</u> V Ž

<u>V</u> Ž

i traženi element se pronalazi.

Interpolirano pretraživanje zahteva kodiranje slova (koja nisu numerički podaci) koje se može ostvariti kodiranjem slova njegovim rednim brojem u našoj abecedi (A – 1, B – 2, C – 3, Č – 4, Ć – 5, . . . , Ž – 30). Traženje slova V sada ne počinje od sredine niza (od slova NJ), već od elementa sa indeksom s=1+(28-1)*(18-1) / (30-1) = 16 tj. od slova T. Pretraživanje sada ima sledeći tok (podvučen element sa kojim se vrši poređenje):

ind.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
el.	A	A	A	E	E	I	I	M	NJ	P	P	R	R	R	R	<u>T</u>	V	Ž
kôd:	1	1	1	9	9	13	13	18	20	22	22	23	23	23	23	26	28	30
																	<u>V</u>	Ž

Interpolirano pretraživanje smanjuje broj ispitivanih elemenata na loglog n – sporo rastuća funkcija, skoro konstanta. Ovu metodu vredi primeniti za vrlo velike nizove, i kada je raspodela vrednosti u intervalu ravnomerna, kada je poređenje i pristup podatku skupo.

11.7 Sortiranje

Pod sortiranjem se podrazumeva razmeštanje elemenata nekog skupa u neopadajući ili nerastući poredak. Skup može biti predstavljen nizom, listom, datotekom, itd. Ako se sortira skup elementarnih podataka (brojeva, stringova), onda se sortiranje vrši upravo po vrednosti tih podataka. Moguće je sortiranje primeniti i na strukture kao elemente, pri čemu se sortiranje vrši prema nekom (zadatom) članu strukture, tj. po vrednosti tog člana koji se onda naziva *članom sortiranja* (v. glavu 12).

Razlozi za sortiranje su višestruki, npr.:

1. rešavanje zadataka grupisanja tj. sakupljanja elemenata sa jednakom vrednošću (ili jednakom vrednosšću zadatog polja);
2. nalaženje zajedničkih elemenata (preseka) dva ili više skupova (npr. datoteka); za sortirane skupove ovaj zadatak je jednostavan i obavlja se u jednom prolazu kroz svaki od skupova;
3. pretraživanje – nalaženje elementa sa zadatom vrednošću (zadatog polja), za skup sortiran po vrednostima tog polja obavlja se lako (npr. traženje po rečniku moguće je upravo zahvaljujući njegovoj sortiranosti).

Postoje dve vrste sortiranja – unutrašnje i spoljašnje sortiranje, i dve grupe metoda koje se na njih odnose, svaka metoda praćena nizom algoritama za njenu implementaciju.

Unutrašnje sortiranje podrazumeva sortiranje skupa u unutrašnjoj memoriji, dok se spoljašnje sortiranje odnosi na skup na spoljašnjem memorijskom mediju (npr. disku) i preptpostavlja da ceo skup ne može da stane u unutrašnju memoriju.

U postupku sortiranja potrebno je, pre svega, razviti korektan algoritam, a zatim i efikasan algoritam u pogledu vremena izvršavanja i zauzeća memorije.

Postoji veći broj algoritama unutrašnjeg sortiranja koji se međusobno razlikuju po primenljivosti na specifične strukture elemenata i skupa, ali i po efikasnosti pri izvršavanju.

11.7.1 Elementarne metode sortiranja

Elementarne metode sortiranja su jednostavne ali za velike skupove neefikasne. Ta neefikasnost je posledica činjenice da je njihova vremenska složenost proporcionalna sa n^2 gde je n – kardinalnost skupa.

Elementarne metode sortiranja stoga daju dobre rezultate za male skupove slogova (brojeva) (do 500). Kada se sortiranje izvodi jedanput ili ograničen broj puta, i elementarni algoritmi mogu biti zadovoljavajući.

Elementarne metode sortiranja, između ostalog, omogućuju izučavanje složenijih. Takve su, na primer, metode sortiranje izborom najmanjeg (najvećeg) elementa, sortiranje umetanjem, šelsort (modifikacija sortiranja umetanjem,) bubble sort, itd.

11.7.2 Sortiranje izborom najvećeg elementa

Sortiranje niza u neopadajućem ili nerastućem poretku podrazumeva nalaženje jedne permutacije elemenata niza u kojoj se elementi pojavljuju u neopadajućem tj. nerastućem poretku.

Metoda sortiranja izborom najvećeg elementa odnosi se na sortiranje niza podataka x sa n elemenata u nerastući poredak (slično izbor najmanjeg elementa obezbeđuje sortiranje u neopadajući poredak). Prvo se nalazi najveći element niza i on se "dovodi" na prvo mesto, zatim se nalazi najveći od preostalih $n-1$ elemenata i on se "dovodi" na drugo mesto, nalazi najveći od preostalih $n-2$ elemenata i dovodi na treće mesto, itd, zaključno sa nalaženjem većeg od poslednja dva elementa i njegovim "dovođenjem" na pretposlednje mesto. Na poslednjem mestu će ostati element koji nije veći ni od jednog u nizu (najmanji element). "Dovođenje" najvećeg elementa na prvo mesto najjednostavnije se realizuje razmenom mesta tog elementa sa prvim elementom. Da bi se znalo sa kog mesta "dolazi" najveći element (i na koje treba "poslati" prvi element), potrebno je pamtiti indeks najvećeg elementa. Slično važi i za "dovođenje" najvećeg elementa preostalog niza na drugo, treće, itd, mesto.

Dakle, globalna struktura algoritma ima sledeći oblik:

(a) odrediti najveći element x_{max} (algoritam za nalaženje maksimalnog elementa sledi);

(b) razmeniti x_{max} i x_1 i

(c) ponoviti korake (a) i (b) posmatrajući nizove $x_2, \ldots, x_n; x_3, \ldots, x_n; \ldots$, dok ne ostane samo x_n, tj.

```
for (i=1; i<n; i++)
{
    naći najveći element u nizu x[i], x[i+1], ..., x[n] i zapamtiti njegov indeks max;
    razmeniti x[i] i x[max];
}
```

Nalaženje maksimalnog elementa niza. Neka je dat niz $x[m..n]$. Zadatak je naći indeks j t.d. $x_j = \max(x_m, \ldots, x_n)$.

Nalaženje najvećeg elementa u nizu čiji su elementi na "mestima" (indeksima) $m, m+1, \ldots, n$ može se realizovati tako što se prvo prvi u nizu proglasi za najveći ($\max = m$), a zatim se najveći poredi sa svim sledećim elementima (sa indeksima od $m+1$ do n). Kadgod se naiđe na element veći od najvećeg, on se proglašava za najveći tj. njegov indeks, j, dodeljuje se indeksu najvećeg elementa, max ($\max = j$).

Funkcija za nalaženje najvećeg elementa u nizu $x[m..n]$ celih brojeva, u C-u ima sledeći oblik:

```
int maxelem(int x[ ], int m, int n)
   {
       int max, j;
       max = m;
       for (j=m+1; j <= n; j++)
           if (x[j] > x[max]) max=j;
       return max;
   }
```

Funkcija za sortiranje izborom najvećeg elementa. Kako smo razvili i funkciju za nalaženje najvećeg elementa i funkciju razmene (swap), možemo napisati i funkciju za sortiranje niza (celih brojeva) izborom najvećeg elementa u C-u (uz odgovarajuće pomeranje indeksa s obzirom da indeksi kreću od 0):

```
void maxsort(int x[ ], int n)
{
    void swap(int [ ], int, int);
    void maxelem(int [ ], int, int);
    int i;
    for(i=0; i<n–1; i++)
        swap(x, i, maxelem(x, i, n–1));
}
void swap(int x[ ], int i, int j)
{
    int priv;
    priv = x[i];
    x[i] = x[j];
    x[j] = priv;
}
```

Ovaj algoritam pokazuje dobra svojstva za male nizove, npr. do 1000 elemenata. Broj poređenja između elemenata niza je oko $n^2/2$ jer se spoljašnja petlja (po i) izvršava n puta a unutrašnja (po j) oko $n/2$ puta u proseku. Iskaz max= j se izvršava reda veličine $n\log n$ puta pa nije deo unutrašnje petlje. Stoga je vreme izvršavanja ovog algoritma proporcionalno sa n^2.

11.7.3 Sortiranje umetanjem

Ako je dat niz (x_n) sa elementima nekog, uređenog tipa T, koji treba urediti u neopadajući poredak, ova metoda sortiranja polazi od pretpostavke da imamo uređen početni deo niza, $x_1, \ldots, x_{i-1}$ (to svakako važi za $i = 2$, jer je podniz sa jednim elementom uređen); u svakom koraku, počevši od $i = 2$ i povećanjem i, i-ti element se stavlja na pravo mesto u odnosu na prvih (uređenih) $i - 1$.

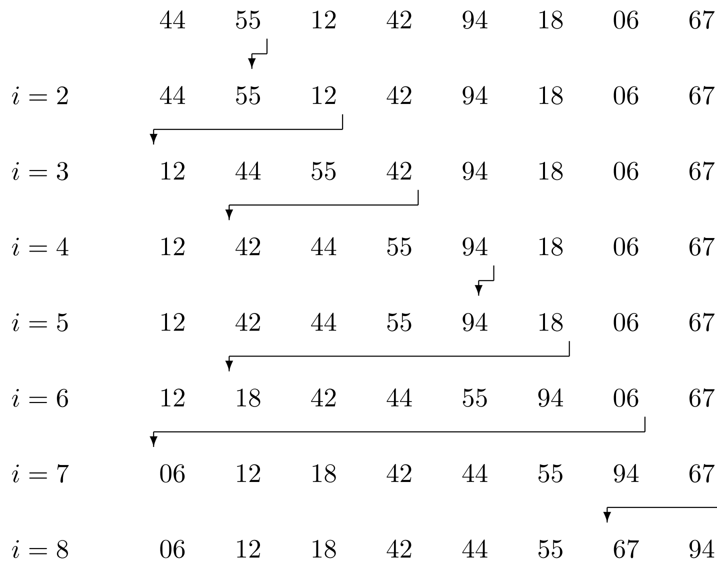

Figure 11.3: Primer sortiranja umetanjem

Na primer, slika 11.3 prikazuje postupak sortiranja datog niza od 8 celobrojnih elemenata:

Algoritam sortiranja umetanjem prolazi kroz sledeće korake:
for (i=2; i<=n; i++) {
    $v = x[i]$;
    "umetnuti v na odgovarajuće mesto u $x_1, x_2, \dots, x_i$
}

"Umetanje na odgovarajuće mesto" može se izvršiti pomeranjem elemenata koji prethode $(x[j], j = i-1, \dots, 1)$ za po jedno mesto udesno $(x[j+1] = x[j])$. Kriterijum kraja tog pomeranja je dvojak:

1. nađena je vrednost x_j t.d. $x_j < v$ (onda se v stavlja na $j+1$. mesto);
2. došlo se do levog kraja niza

Funkcija na C-u koja realizuje ovu metodu ima sledeći oblik (uz odgovarajuće pomeranje indeksa s obzirom da indeksi kreću od 0):

```
void umetsort(int x[ ], int n)
{
    int v, i, j ;
    for(i=1; i<n; i++) {
        v=x[i]; j=i–1;
```

```
            while (j>=0 && v<x[j]) {
                x[j+1]=x[j]; j--;
            }
            x[j+1]=v;
        }
    }
```

Broj poređenja, P_i, u i-tom prolazu je najviše i a najmanje 1, a u proseku je $i/2$. Broj premeštanja, M_i, je $P_i + 2$. Zato je ukupan broj poređenja i premeštanja

$P_{min} = n - 1$

$P_{max} = n + (n-1) + (n-2) + \cdots + 1 = n \cdot (n+1)/2 = \frac{1}{2}(n^2 + n)$,

i slično za M_i.

Najmanji broj poređenja i premeštanja je kada je polazni niz uređen; najveći je kada je polazni niz uređen u obrnutom poretku.

Moguća su poboljšanja algoritma koja koriste uređenost niza $x_1, x_2, \ldots x_{i-1}$ (npr. binarno pretraživanje kojim se dobije **sortiranje binarnim umetanjem**).

Za vežbu napisati odgovarajuću funkciju sortiranja binarnim umetanjem.

11.7.4 Šelsort

Šelsort je jednostavno proširenje sortiranja umetanjem koje dopušta direktnu razmenu udaljenih elemenata. Proširenje se sastoji u tome da se kroz algoritam umetanja prolazi više puta; u prvom prolazu, umesto koraka 1 uzima se neki korak h koji je manji od n (što omogućuje razmenu udaljenih elemenata) i tako se dobija h-sortiran niz, tj. niz u kome su elementi na rastojanju h sortirani, mada susedni elementi to ne moraju biti. U drugom prolazu kroz isti algoritam sprovodi se isti postupak ali za manji korak h. Sa prolazima se nastavlja sve do koraka $h = 1$, u kome se dobija potpuno sortirani niz.

Dakle, ako se svako pojavljivanje "1" u algoritmu sortiranja umetanjem zameni sa h, dobija se algoritam koji, uzimajući svaki h-ti element (počevši od bilo kog) proizvodi sortirani niz. Takav, "h-sortirani" niz je, u stvari, h sortiranih podnizova, koji se preklapaju. h-sortiranjem za veliko h premeštaju se udaljeni elementi, i time se olakšava h-sortiranje za manje vrednosti h. Koristeći ovu proceduru za bilo koji niz vrednosti od h, koji se završava sa 1, proizvodi se sortirani niz. To je šelsort.

Za najveći (prvi) korak h zgodno je uzeti (mada ne i optimalno u odnosu na vreme izvršavanja algoritma) najveći broj oblika $3 \times k + 1$ koji nije veći od n, kao sledeći korak h – celobrojnu trećinu prethodnog h, itd, sve dok se za h ne dobije vrednost 1. Moguće je za h uzeti i druge sekvence, na primer $n/2$, $n/2^2$, $n/2^3$, $\ldots$, 1.

Na primer, za petnaest elemenata u nizu (n=15), za h=13,4,1, redom, dobiju se sledeći nizovi:

vr. indeksa:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
početni niz:	P	R	I	M	E	R	S	O	R	T	I	R	A	NJ	A
$h = 13:$	NJ	A	I	M	E	R	S	O	R	T	I	R	A	P	R
$h = 4:$	A	A	I	M	E	P	I	O	NJ	R	R	R	R	T	S
$h = 1:$	A	A	E	I	I	M	NJ	O	P	R	R	R	R	S	T

U prvom prolazu, P sa pozicije 1 se poredi sa NJ sa pozicije 14, i razmenjuju se, zatim se R sa pozicije 2 poredi sa A sa pozicije 15 i razmenjuju se. U drugom prolazu, NJ, E, R, A sa pozicija 1,5,9,13, redom, razmeštaju se tako da daju redosled A, E, NJ, R na tim pozicijama, i slično za pozicije 2, 6, 10, 14, itd. Poslednji prolaz je baš sortiranje umetanjem, ali se nijedan element ne premešta daleko. Izbor za h je proizvoljan i ne postoji dokaz da su neki izbori za h optimalni. Niz koji se u praksi pokazao dobar je ...,1093, 364, 121, 40, 13, 4, 1.

Implementacija opisanog shellsort algoritma koji koristi korake h=3*k+1 (najveće takvo h <n), h/3, h/3^2, ..., 1, može da se zada sledećom funkcijom:

```
void shellsort(int x[ ], int n)
{
    int v,i,j,h;
    if (n/3*3+1<n) h=n/3*3+1;
    else h=n/3*3+1-3;
    for(h; h>0; h/=3) {
        for(i=h; i<n; i++) {
            v=x[i]; j=ih;
            while (j>=0 && v<x[j]) {
                x[j+h]=x[j]; j-=h;
            }
            x[j+h]=v;
        }
    }
}
```

Za implementaciju koja koristi korake h=n/2, n/2^2,..., 1, v. Kernighan, Ritchie, "The C programming language" – tačka 3.5.

Metodu shellsort je teško porediti sa drugim zato što je funkcionalni oblik vremena izvršavanja nepoznat i zavisi od sekvence za h. Za gornji algoritam su granice $n*(\log n)^2$ i $n^{1.25}$. Metoda nije posebno osetljiva na početni redosled elemenata, za razliku od algoritma sortiranja umetanjem koji ima linearno vreme izvršavanja ako je niz u početku sortiran, i kvadratno vreme ako je niz u obrnutom redosledu. Metoda se često primenjuje jer daje zadovoljavajuće vreme izvršavanja do, npr. 5000 elemenata, a algoritam nije složen.

11.7.5 Bubble sort

Ova metoda je elementarna: ponavlja se prolazak kroz niz elemenata i razmenjuju se susedni elementi, ako je potrebno, sve dok se ne završi prolaz koji ne zahteva

nijednu razmenu. Tada je niz sortiran. U svakom prolazu kroz niz broj elemenata koje treba porediti je za 1 manji jer na kraj niza ”isplivava” najveći (najmanji) element.

```
void bubblesort(int x[ ], int n)
{
      void swap(int x[ ], int i, int j);
      int j, ind;
      do {
            ind=0;
            for(j=1; j<n; j++)
                  if (x[j–1] > x[j]) {
                        swap(x, j–1, j);
                        ind = 1;
                  }
            n=n-1;
      } while (ind);
}
```

Algoritam bubble sort ima kvadratno vreme, složeniji je i manje efikasan od sortiranja umetanjem. Unutrašnja petlja ima više iskaza i od sortiranja umetanjem i od sortiranja izborom.

12

Strukture

Struktura u C-u je kolekcija jedne ili više promenljivih ne obavezno istih tipova, povezanih i grupisanih pod istim imenom radi lakšeg rada. Komponente strukture u C-u se nazivaju *članovima*.

Na primer, struktura je slog o osobi: sadrži članove ime, adresu, broj lične karte (ili matični broj), itd. Neki od članova mogu opet biti strukture (npr. ime ili adresa). Ili, struktura je tačka sa komponentama – x i y koordinatama. Struktura je i pravougaonik (stranica paralelnih koordinatnim osama), sa komponentama – parom tačaka t1, t2 (donje levo i gornje desno teme), itd:

```
struct tacka {
    int x;
    int y;
};
struct pravoug {
    struct tacka t1;
    struct tacka t2;
};
```

Deklaracija strukture definiše tip; on može da bude eksplicitno imenovan – kao u prethodnom primeru tip tacka, i to eksplicitno ime može se koristiti kasnije kao skraćenica za oznaku tog tipa strukture. Uz deklaraciju strukture (ili skraćenicu tipa koji definiše struktura) mogu se navesti promenljive odgovarajućeg tipa koji definiše struktura. Na primer, deklaracijom

```
struct {
    int x;
    int y;
} a, b, c;
```

definiše se (eksplicitno neimenovani) tip strukture i tri promenljive tog tipa – a, b, c. Iste promenljive mogu se deklarisati i korišćenjem prethodno uvedene skraćenice:

```
struct tacka a, b, c;
```

Struktura se može inicijalizovati navođenjem vrednosti komponenti – konstantnih izraza, npr.

```
struct tacka t = {100,100};
```

Obraćanje članu strukture vrši se navođenjem imena strukture i imena člana razdvojenih tačkom (.). Na primer,

```
printf("%d,%d", t.x, t.y);
```

ili, za promenljivu s tipa pravoug,

```
struct pravoug s;
```

oznaka s.t1.x označava x–koordinatu donjeg levog ugla pravougaonika s.

12.1 Strukture i funkcije

Operacije nad strukturom su kopiranje i dodela vrednosti, uzimanje adrese strukture operatorom & (operacije nad strukturom kao celinom), i obraćanje članovima strukture. Zato se strukture mogu pojaviti kao argumenti funkcije (kopiranje) odnosno kao vrednost koju funkcija vraća (dodela). Strukture se ne mogu porediti.

Strukture kao argumenti funkcije mogu se prenositi kao celine, prenosom komponenti ili prenosom pokazivača na strukturu. Vrednost funkcije takođe može biti tipa strukture ili pokazivača na strukturu. Sledeće funkcije ilustruju ove mogućnosti.

Na primer, funkcija izgtacka izgrađuje tačku od njenih koordinata x i y (argumenti su tipa int a vrednost koju funkcija vraća je tipa struct tacka):

```
/* izgtacka: izgrađuje tacku od komponenata x i y */
struct tacka izgtacka(int x, int y)
{
    struct tacka priv;
    priv.x = x;
    priv.y = y;
    return priv;
}
```

Funkcije za sabiranje dve tačke ima i argumente i vrednost tipa strukture tacka:

```
/* sabtacaka: sabira dve tacke */
struct tacka sabtacaka(struct tacka t1, struct tacka t2)
{
    t1.x += t2.x;
    t1.y += t2.y;
    return t1;
}
```

Funkcija tacuprav testira da li je zadata tačka u unutrašnjosti zadatog pravougaonika (argumenti su tipa struktura tacka odnosno pravoug, a funkcija vraća ceo broj – rezultat testa):

```
/* tacuprav: vraca 1 ako je ta cka t u pravougaoniku p, 0 inace */
```

```
int tacuprav(struct tacka t, struct pravoug p)
{
    return t.x > p.t1.x && t.x < p.t2.x
        && t.y > p.t1.y && t.y < p.t2.y;
}
```

U slučaju da je struktura koja se predaje kao argument funkciji – velika, umesto kopiranja cele strukture (u slučaju da se prenosi sama satruktura kao argument) eknomičnije je preneti pokazivač na strukturu kao argument. Na primer

```
struct tacka t, *pt;
pt = &t;
```

deklariše promenljivu pt kao pokazivač na strukturu tipa struct tacka. Kako je pt dobila vrednost – adresu strukture t tipa tacka, to je ∗pt – sama ta struktura t, a (∗pt).x i (∗pt).y su članovi te strukture. Zagrade su obavezne jer bez njih značenje oznake ∗pt.x je isto što i ∗(pt.x), tj. objekat na koji pokazuje pt.x; kako pt.x nije pokazivač (već int), oznaka nema smisleno značenje.

Skraćenica za oznaku člana strukure na koju pokazuje pokazivač p je

```
p->član
```

pa se može pisati

```
printf("tacka t je (%d,%d)\n", pt->x, pt->y);
```

sa istim značenjem kao i

```
printf("tacka t je (%d,%d)\n", (*pt).x, (*pt).y);
```

Značajno je da je prioritet operatora nad strukturama (. i −>), kao i zagrađivanja argumenata kod funkcija i indeksiranje nizova [ ], najviši, a asocijativnost – kao i kod tih operatora - sleva nadesno. Na primer,

```
struct {
    int len;
    char *str;
} *p;
```

```
++p ->len
```

uvećava len a ne p,

∗p −>str uzima ono na šta pokazuje str, ∗p −>str++ uvećava str posle uzimanja onoga na šta str pokazuje (kao i ∗s++), (∗p −>str)++ uvećava ono na šta str pokazuje, a ∗p++ −>str uvećava p posle uzimanja objekta na koji pokazuje str.

12.2 Nizovi struktura

Ako treba da obrađujemo veći broj struktura istog tipa, možemo da uvedemo onoliko nizova koliko članova ima struktura, ili da uvedemo niz struktura. Na primer, u programu koji broji pojavljivanja ključnih reči C-a u ulaznom tekstu, s obzirom da imamo niz ključnih reči i niz brojača (uz svaku reč po jedan), prirodno je udružiti ključnu reč i brojač u strukturu i uvesti niz struktura:

```
struct kljuc {
    char *rec;
```

```
        int brojac;
    } tabelak[NKLJUC];
```

Ova deklaracija deklariše tip strukture kljuc, i definiše niz tabelak struktura tog tipa i rezerviše prostor za njega. Svaki element niza je struktura. Ako se taj niz inicijalizuje na početku programa ključnim rečima C-a sa 0 pojavljivanja

```
    struct kljuc {
        char *rec;
        int brojac;
    } tabelak[ ] = {
        /* ...*/,
        "break", 0,
        "case", 0,
        "char", 0,
        /* ... */
        "void", 0,
        "while", 0
    };
```

onda glavna funkcija čita tekst sa ulaza reč po reč – pozivom odgovarajuće funkcije, i traži učitanu reč u nizu tabelak. To traženje može da se realizuje nekom varijantom linearnog ili, s obzirom da je niz ključnih reči sortiran – nekom varijantom binarnog pretraživanja (v. tačku 11.6). Odgovarajuće funkcije, prilagođene ovom problemu, imaju sledeće definicije:

```
/* linsearch: nalazi rec u nizu tab[0], ..., tab[n-1] */
int linsearch(char *rec, struct kljuc tab[ ], int n)
{
    int i;
    i = –1;
    do {i++;}    /*  (tab[i].rec != rec za j=0..i–1    */
        while (i<n && strcmp(tab[i].rec, rec));
    if (i<n) return i; /* (tab[j].rec != rec za j=0..i–1 && tab[i].rec == rec) */
                        /*    || tab[j].rec != rec za j=0..n–1   */
    else return –1;
}

/* binsearch: nalazi rec u nizu tab[0], ..., tab[n-1] */
int binsearch(char *rec, struct kljuc tab[ ], int n)
{
    int por;
    int l, d, s;
    l = 0;
    d = n – 1;
    while (l <= d) {
        s = (l + d) / 2;
        if ((por = strcmp(rec, tab[s].rec)) < 0)
```

```
            d = s – 1;
        else if (por > 0)
            l = s + 1;
        else
            return s;
    }
    return –1;
}
```

Funkcija pretraživanja, npr. binsearch, koristi se u našem primeru pretraživanja ključnih reči, pozivom sledećeg oblika :

```
    if ((n=binsearch(rec, tabelak, NKLJUC))>=0)
        tabelak[n].brojac++;
```

Promenljiva "rec" ovde označava izdvojenu reč iz teksta koja može (a ne mora) biti ključna reč C-a, pa se zato može naći (ili ne) među rečima niza struktura tabelak (npr. kao ključna reč n-te strukture, za n>=0, a ako funkcija vrati –1, reč nije nađena u nizu tabelak). Ako se nađe, brojač odgovarajuće (n-te) ključne reči se povećava za 1.

Strukture i nizove moguće je kombinovati i na druge načine. Na primer, moguće je definisati strukturu koja kao jedan svoj član ima niz. Jedan primer za ovakvo strukturiranje podataka je struktura polinoma koja se može opisati sledećom definicijom tipa (polinom stepena n sa koeficijentima u nizu a):

```
typedef struct polinom
{
    int n;
    double a[10];
} polinom;
```

12.3 Pokazivači na strukture

Nizovi i pokazivači u C-u mogu da se koriste u sličnom kontekstu. Tako se, kao na svaku drugu promenljivu, može definisati pokazivač i na strukturu, pri čemu se i elementima niza struktura može pristupati preko pokazivača na strukture umesto preko indeksa niza struktura. Dakle, i prethodni zadatak prebrojavanja ključnih reči, umesto korišćenjem nizova, može da se reši i primenom pokazivača na strukture.

Spoljašnja deklaracija tabelak ostaje ista, ali se funkcije main i binsearch nešto modifikuju. Pre svega, funkcija binsearch vraća pokazivač na strukturu tipa kljuc (čiji član rec ima vrednost jednaku traženoj ključnoj reči), a ne int (kao u prethodnom rešenju – indeks elementa niza struktura sa traženom rečju) ili NULL (ako takva struktura ne postoji). Zatim, elementima niza struktura tabelak pristupa se preko pokazivača, što dalje zahteva da inicijalne vrednosti za l i d budu pokazivači

na početak i kraj (tj. iza kraja) niza tabelak, a pokazivač na strukturu "u sredini" (s) sada ne može da se računa kao (l+d)/2 (jer sabiranje pokazivača nije dozvoljena operacija), već kao l + (d–l) / 2 (jer je oduzimanje pokazivača dozvoljena operacija, kao i dodavanje celog broja (d–l)/2 na pokazivač l.

Ako p pokazuje na jednu strukturu u nizu struktura tabelak, onda aritmetika nad pokazivačima obezbeđuje da p++ pokazuje na sledeću strukturu u tom nizu.

Funkcija binsearch sada ima definiciju oblika:

```
/* binsearch: naci rec u nizu tab[0], ..., tab[n-1] */
struct kljuc *binsearch(char *rec, struct kljuc *tab, int n)
{
    int por;
    struct kljuc *l = &tab[0];
    struct kljuc *d = &tab[n];
    struct kljuc *s;
    while (l <= d) {
        s = l + (d − l) / 2;
    if ((por = strcmp(rec, s − >rec)) < 0)
        d = s − 1;
    else if (por > 0)
        l = s +1;
    else
        return s;
    }
    return NULL;
}
```

Sada se funkcija pretraživanja, binsearch, koristi u našem primeru pretraživanja ključnih reči, pozivom sledećeg oblika:

```
if(p=binsearch(rec, tabelak, NKLJUC)) ! = NULL)
    p–>brojac++;
```

12.4 Unija

Unija je promenljiva koja može da dobije vrednosti različitih tipova, u raznim vremenskim trenucima. Unija, dakle, omogućuje smeštanje različitih vrsta (tipova) podataka u jedan isti memorijski prostor. To je zapravo struktura čiji se svi članovi smeštaju od iste (nulte) pozicije tog memorijskog prostora.

Sintaksa unije u C-u zasniva se na sintaksi strukture. Tako, ako želimo da jedna ista promenljiva, *u*, može da primi celobrojnu, realnu vrednost, ili vrednost tipa pokazivača na karakter, onda se promenljiva *u* može definisati kao unija sledećom definicijom promenljive (i tipa unije):

```
union u_obelezje {
    int ivred;
```

```
        float fvred;
        char *svred;
    } u;
```

Ovakva definicija može da bude korisna u upravljanju tabelama simbola jednog prevodioca.

Promenljiva *u* je dovoljno "velika" (tj. odgovarajući memorijski prostor je dovoljno veliki) da primi najveći od tri navedena tipa, bez obzira na njihovu konkretnu veličinu koja zavisi od implementacije. Promenljiva *u* može se navesti u izrazu, na mestu na kome se očekuje vrednost tipa int, float ili char *, pod uslovom da je upotreba promenljive *u* (tip koji se u izrazu očekuje) saglasna sa tipom trenutne vrednosti promenljive *u*. Ako je uniji dodeljena vrednost jednog tipa a iz nje se čita vrednost drugog tipa, rezultat će zavisiti od implementacije.

Članovima unije pristupa se kao i članovima strukture, navođenjem imena unije (odnosno pokazivača na uniju) i imena člana:

```
ime_unije.clan
pokazivac_na_uniju − >clan
```

Unije se mogu koristiti u strukturama i nizovima, isto kao što se i strukture i nizovi mogu koristiti u unijama - kao njihovi članovi. Na primer, u tabeli simbola simtab koja je definisana nizom struktura čiji je jedan član prethodno opisana unija,

```
struct {
    char *ime;
    int u_tip;
    union {
        int ivred;
        float fvred;
        char *svred;
    } u;
} simtab[NSIM];
```

članu ivred obraćamo se sa

```
simtab[i].u.ivred
```

a prvom karakteru stringa svred obraćamo se jednim od sledeća dva izraza:

```
*simtab[i].u.svred
sintab[i].u.svred[0]
```

Nad unijom su dopuštene iste operacije kao i nad strukturom: dodela vrednosti i kopiranje (unije kao celine), uzimanje adrese i pristup članovima.

Unija može da se inicijalizuje samo vrednošću tipa svog prvog člana (na primer, prethodno definisana unija *u* može da se inicijalizuje samo celobrojnom vrednošću).

12.5 Bit-polje

Kada je memorijski prostor ograničen i kritičan, može da se ukaže potreba da se nekoliko objekata smesti u jedan registar ili drugu memorijsku celinu definisanu implementacijom – na primer u slučaju jednobitnih "zastavica" koje signaliziraju neki događaj ili stanje. Na primer, kompilator može da pridruži svakom identifikatoru programa takve informacije – jeste / nije rezervisana reč, jeste / nije vrste "external", jeste / nije static, itd. Niz takvih jednobitnih "zastavica" može da se "spakuje" u jedan char ili int.

Jedna implementacija ovakvog koncepta je i tzv. *bit-polje*, ili samo *polje*. Bit-polje predstavlja skup susednih bitova unutar jedinstvene memorijske celine definisane implementacijom. Sintaksa bit-polja je slična sintaksi strukture i može da se ilustruje sledećim primerom:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern : 1;
    unsigned int is_static : 1;
} flags;
```

Prethodna definicija definiše promenljivu flags (zastavice) koja sadži tri jednobitna polja. Polja moraju da budu tipa int – eksplicitno kao unsigned ili signed. Pojedinim poljima pristupa se na isti način kao i članovima strukture, tj. flags.is_keyword, flags.is_extern, i sl. Polja se ponašaju kao mali celi brojevi i mogu da učestvuju u aritmetičkim izrazima kao i drugi celi brojevi. Tako, moguća je dodela flags.is_extern = flags.is_static = 1;

Ali, polja nemaju adresu i operator referenciranja, &, ne može da se primeni na njih. Skoro sve u vezi sa bit-poljima zavisi od implementacije.

13

Datoteke – ulaz i izlaz

Datoteka je niz znakova (bajtova) kome je pridruženo ime, i koji se može interpretirati na razne načine – npr. kao tekst, kao program na programskom jeziku, kao program preveden na mašinski jezik, kao slika, crtež, zvuk, itd. Na sadržaj datoteke tj. način na koji ga treba interpretirati, ukazuje ekstenzija uz ime datoteke (npr. c, exe, txt, pas, obj, exe, bmp, itd). (Pojam spoljašnje datoteke se bitno razlikuje od pojma datoteke kao strukture podataka u nekim programskim jezicima).

U programskom jeziku C datoteka je pre svega vezana za ulaz i izlaz podataka. Mogućnosti ulaza i izlaza nisu sastavni deo samog jezika C, ali su neophodne za komunikaciju programa sa okruženjem i mogu biti znatno kompleksnije od onih koje smo neformalno koristili do sada (getchar, putchar, printf, scanf). Ovaj aspekt jezika podržan je funkcijama standardne biblioteke, čija su svojstva precizirana u preko deset standardnih zaglavlja – neka od njih su već korišćena – <stdio.h>, <string.h>, <ctype.h>, itd.

13.1 Standardni ulaz i izlaz

Standardna biblioteka C-a uključuje jednostavni model tekstualnog ulaza i izlaza. Ulazni tekst se sastoji od niza linija, svaka linija se završava karakterom za novi red (ili odgovarajućim karakterima koji se konvertuju u karakter za novi red – npr. parom karaktera CR (ASCII 13), LF (ASCII 10) pod operativnim sistemima Windows).

Najjednostavniji mehanizam za čitanje jednog karaktera sa standardnog ulaza (obično tastature), kako je već rečeno, predstavlja funkcija

int getchar(void)

koja vraća sledeći ulazni karakter, ili EOF kada se dođe do kraja. Simbolička konstanta EOF je definisana u zaglavlju <stdio.h>, i obično ima vrednost -1.

U mnogim okruženjima tastatura se može zameniti datotekom, korišćenjem *konvencije preusmerenja*, <: ako program prog koristi funkciju getchar, onda se on može izvršiti komandom

prog <indat
pri čemu se karakteri čitaju iz datoteke indat umesto sa tastature.

Najjednostavniji mehanizam za izdavanje jednog karaktera na standardni izlaz (obično ekran), kako je već ranije rečeno, predstavlja funkcija
int putchar(int)
koja, pozvana sa putchar(c) izdaje karakter c na standardni izlaz i vraća karakter c ili EOF ako dođe do greške. Opet se, umesto standardnog izlaza, može koristiti i datoteka (outdat), preusmerenjem oblika
prog >outdat

Za preusmerenje izveštaja o greškama sa standardnog izlaza na datoteku, npr. error, koristi se operator preusmerenja 2>, na primer,
prog >outdat 2 >error
Svaka izvorna datoteka koja koristi funkcije ulazno/izlazne biblioteke, mora da uključi zaglavlje <stdio.h>.

13.1.1 Formatirani izlaz

Kao što je već pomenuto u delu "Pregled programskog jezika C", funkcija formatiranog izlaza na standardni izlaz ima deklaraciju oblika

int printf(char *format, arg_1, arg_2, ...);

Karakteristični za string format, pod čijom kontrolom se argumenti konvertuju i izdaju na standardni izlaz, jesu objekti za specifikaciju konverzije koji govore kojim tipom interpretirati i u koji oblik konvertovati svaki od argumenata. Ovi objekti počinju znakom % i završavaju se karakterom konverzije – nekim od karaktera "d", "i" (za izdavanje celog broja), "c" za izdavanje jednog karaktera, "s" za izdavanje stringa (niske karaktera), "f" za izdavanje realnog broja u neeksponencijalnom obliku, "e" ili "E" za izdavanje realnog broja u eksponencijalnom obliku, sa malim odnosno velikim slovom e, "g" ili "G" za izdavanje realnog broja u neeksponencijalnom ili eksponencijalnom (sa malim odnosno velikim slovom e) obliku (koji je kraći), "p" za memorijsku adresu u heksadekadnom obliku, "o" za neoznačenu oktalnu vrednost, "u" za neoznačenu celobrojnu vrednost, "x" ili "X" za neoznačenu heksadekadnu vrednost (sa malim odnosno velikim ciframa a – f).

Između znaka % i karaktera konverzije, objekat za specifikaciju konverzije može da uključi i druge karaktere specifičnog značenja:

- znak minus za levo poravnanje
- broj koji određuje minimalnu širinu polja
- tačku, koja razdvaja širinu polja od broja koji označava preciznost
- broj koji označava preciznost – maksimalni broj karaktera koji će se izdati, ili broj cifara iza decimalne tačke broja u pokretnom zarezu, ili najmanji broj cifara celog broja

- karakter h ili l za kratki odnosno dugi ceo broj.

Slično funkciji printf (6.9), funkcija
int sprintf(char *string, char *format, arg_1, arg_2, ...)
konvertuje i formatira argumente prema stringu format, ali odgovarajuće karaktere ne izdaje na standardni izlaz već ih upisuje u string.

13.1.2 Formatirani ulaz

Kao što je već pomenuto u delu "Pregled programskog jezika C", funkcija scanf jeste analogon funkcije printf za formatirani unos:
int scanf(char *format, arg_1, arg_2, ...)
Ona uzima nisku karaktera sa standardnog ulaza, konvertuje je u vrednosti svojih argumenata i dodeljuje ih promenljivim na koje pokazuju argumenti arg_1, arg_2, ... Dakle, svi argumenti funkcije scanf osim prvog (formata) jesu pokazivači.

Slično funkciji sprintf, funkcija
int sscanf(char *string, char *format, arg_1, arg_2, ...)
konvertuje nisku karaktera u vrednosti svojih argumenata, ali nisku karaktera ne uzima sa standardnog ulaza već iz niske string.

String format može da sadrži:

- beline ili tabulatore koji se ignorišu
- obične karaktere (isključujući %) koji se očekuju na ulazu
- objekte za specifikaciju konverzije koji se sastoje od znaka % i karaktera konverzije (sa sličnim značenjem kao kod printf), i, između njih, opcionog karaktera * kojim se "preskače" ulazno polje, bez dodele vrednosti promenljivoj, opcionog broja koji određuje maksimalnu dužinu polja, i opcione oznake kratkog odnosno dugog celog broja.

13.2 Pristup datotekama

Sva komunikacija programa sa okruženjem do sada je podrazumevala standardni ulaz odnosno standardni izlaz, čiju definiciju program automatski dobija od operativnog sistema.

Postoji potreba – i mogućnost – da program pristupi imenovanoj datoteci koja nije povezana sa programom, tj. da se povežu spoljašnje ime (datoteke) i programski iskaz kojim se čitaju podaci iz te datoteke. To se postiže *otvaranjem* datoteke bibliotečkom funkcijom fopen, koja uzima spoljašnje ime datoteke (npr program.c) i vraća pokazivač, tzv. *pokazivač datoteke*, koji pokazuje na strukturu sa informacijama o toj datoteci. Definicija tipa te strukture zove se FILE i nalazi se u standardnom zaglavlju <stdio.h>. Deklaracija funkcije fopen ima oblik

FILE *fopen(char *ime, char *nacin);

gde ime označava string koji sadrži spoljašnje ime datoteke, nacin je nameravani način korišćenja datoteke – "r" za čitanje, "w" za upis", "a" za dodavanje.

Da bi datoteka mogla da se koristi, funkcija fopen mora da se pozove sa odgovarajućim stvarnim argumentima, a njena vrednost da se dodeli pokazivaču na strukturu tipa FILE koji će se nadalje koristiti za čitanje iz tj. upis u datoteku:

FILE *fp;

FILE *fopen(char *ime, char *nacin);

fp = fopen(ime, nacin);

Otvaranje (postojeće) datoteke za upis (odnosno dodavanje) briše (odnosno čuva) prethodni sadržaj te datoteke, dok otvaranje nepostojeće datoteke za upis ili dodavanje kreira tu datoteku. Pokušaj otvaranja nepostojeće datoteke za čitanje proizvodi grešku (fopen vraća NULL).

Postoji više načina za čitanje iz ili upis u datoteku otvorenu za čitanje odnosno upis. Najjednostavniji je par funkcija

int getc(FILE *fp)
int putc(int c, FILE *fp)

kojima se, slično funkcijama getchar, putchar za standardni ulaz odnosno izlaz, čita sledeći karakter datoteke na čiju strukturu pokazuje pokazivač fp, odnosno upisuje karakter c.

Pri startovanju programa (izvršavanju), operativni sistem automatski otvara tri standardne datoteke (definisane u zaglavlju <stdio.h>): stdin (standardni ulaz, obično tastatura), stdout, stderr (standardni izlaz i standardna datoteka za izveštavanje o greškama – obično ekran). Stdin, stdout i stderr su zapravo pokazivači odgovarajućih datoteka, dakle objekti tipa FILE *, ali su to konstantni objekti tog tipa (konstante) i ne mogu im se dodeljivati (menjati) vrednosti.

Sada se funkcije getchar, putchar mogu definisati kao odgovarajuće funkcije getc, putc nad odgovarajućom standardnom datotekom (stdin odnosno stdout):

#define getchar() getc(stdin)
#define putchar(c) putc((c), stdout)

Za formatirani ulaz/izlaz iz/u datoteku, mogu se koristiti funkcije analogne funkcijama scanf, printf, ali za datoteku čiji je pokazivač datoteke fp:

int fscanf(FILE *fp, char *format, arg_1, arg_2, ...)
int fprintf(FILE *fp, char *format, arg_1, arg_2, ...)

Najzad, funkcija
int fclose(FILE *fp)
raskida vezu između pokazivača datoteke i spoljašnjeg imena, koja je bila uspostavljena pozivom funkcije fopen. Pošto postoji ograničenje na broj istovremeno otvorenih datoteka, dobro je zatvoriti svaku datoteku čim prestane njeno korišćnje.

Na primer, funkcija koja kopira sadržaj datoteke sa pokazivačem ulazp u datoteku sa pokazivačem izlazp ima sledeću definiciju:

```
/*filecopy: kopira datoteku ulazp u datoteku izlazp */
void filecopy(FILE *ulazp, FILE *izlazp)
{
    int c;
    while ((c=getc(ulazp)) != EOF)
        putc(c, izlazp);
}
```

a program koji poziva ovu funkciju:

```
#include <stdio.h>
/*kopiranje: kopira datoteku a.c u datoteku b.c */
main()
{
    FILE *ulazp, *izlazp;
    void filecopy(FILE *, FILE *);
    if ((ulazp=fopen(""a.c", "r"))==NULL) {
        printf("ne moze da se otvori datoteka a.c\n");
        return 1; }
    else {
        if ((izlazp=fopen("b.c", "w"))==NULL) {
            printf("ne moze da se otvori datoteka b.c\n");
            return 1; }
        else {
            filecopy(ulazp, izlazp);
            fclose(izlazp);
            izlazp=fopen("b.c", "r");
            filecopy(izlazp, stdout);
            fclose(izlazp); }
        fclose(ulazp);
    }
}
```

13.3 Argumenti u komandnoj liniji

Primer iz prethodne tačke odnosio se na kopiranje datoteke a.c u datoteku b.c. Da bismo poopštili rešenje (na kopiranje datoteke x u datoteku y), možemo da iskoristimo svojstvo C-a da se u komandnoj liniji kojom se izvršava .exe program navedu argumenti programa, u našem slučaju – ime datoteke koja se kopira i ime datoteke u koju se kopira.

Funkcija main do sada je definisana bez argumenata – main(), a komandna linija kojom se poziva izvršavanje programa je sadržala samo jedan "argument" - naziv samog izvršnog programa.

U opštem slučaju, funkcija main može da ima argumente, a njihove vrednosti u tesnoj su vezi sa brojem i vrednostima argumenata komandne linije kojom se

poziva izvršavanje odgovarajućeg programa. Dakle, u opštem slučaju funkcija main se definiše sa dva argumenta: argc (ARGument Count) – broj argumenata u komandnoj liniji kojom se poziva izvršavanje programa, i argv (ARGument Vector) – niz pokazivača na stringove (niske karaktera) koji sadrže argumente komandne linije, po jedan za svaki string. Ako se program poziva samo imenom (u komandnoj liniji nema dodatnih argumenata), onda je broj argumenata komandne linije, tj. vrednost argumenta argc funkcije main jednak 1, dok je argv[0] pokazivač na ime exe-datoteke, a argv[1] je NULL.

Ako se program poziva navođenjem dodatnih argumenata u komandnoj liniji (pored samog imena izvršnog programa), funkcija main definiše se sa argumentima argc, argv, (main(int argc, char *argv[ ])), pri čemu argc dobija vrednost jednaku broju argumenata (računajući i sámo ime datoteke izvršnog programa – to je argv[0]), argv[1] jednak je prvom opcionom argumentu (koji se navodi iza imena izvršne datoteke), argv[2] jednak je drugom opcionom argumentu, itd.

Sada program iz prethodne tačke može da ima dva opciona argumenta – ime ulazne i ime izlazne datoteke, i sledeći oblik:

```
#include <stdio.h>
/*kopiranje: kopira ulaznu datoteku u izlaznu datoteku */
main(int argc, char *argv[ ])
{
    FILE *ulazp, *izlazp;
    void filecopy(FILE *, FILE *);
    if ((ulazp=fopen(*++argv, "r"))==NULL) {
        printf("ne moze da se otvori datoteka %s \n", *argv);
        return 1; }
    else {
        if ((izlazp=fopen(*++argv, "w"))==NULL) {
            printf("ne moze da se otvori datoteka %s\n", *argv);
            return 1; }
        else {
            filecopy(ulazp, izlazp);
            fclose(izlazp);
            izlazp=fopen(*argv, "r");
            filecopy(izlazp, stdout);
            fclose(izlazp); }
        fclose(ulazp);
    }
}
```

Ako se program nalazi u datoteci datkopi, onda se njegovo izvršavanje kojim se datoteka a.c kopira u datoteku b.c poziva komandnom linijom oblika

```
datkopi a.c b.c
```