Glava 1

Programski jezici i prevodioci

Početak razvoja programskih jezika je bio u bliskoj vezi sa razvojem računara tj. sa razvojem hardvera 1940-tih godina. Programiranje u današnjem smislu nastalo je sa pojavom računara fon Nojmanovog tipa čiji se rad kontroliše programima koji su smešteni u memoriji, zajedno sa podacima nad kojim operišu. Na prvim računarima tog tipa moglo je da se programira samo na mašinski zavisnim programskim jezicima, tj. direktno na mašinskom jeziku ili na asembleru. Ovi programski jezici nazivaju se i *niži* programski jezici zbog svoje direktne povezanosti sa hradverom.

Prvi programski jezici zahtevali su od programera da bude upoznat sa najfinijim detaljima računara za koji se piše program. Problemi ovakvog načina programiranja su višestruki. Naime, ukoliko je želeo da programira na novom računaru, programer je morao da izuči sve detalje njegove arhitekture (na primer, skup instrukcija procesora, broj registara, organizaciju memorije). Programi napisani za jedan računar mogli su da se izvršavaju isključivo na istim takvim računarima, nije ih bilo moguće *preneti* na drugačije računare već je za njih bilo neophodno pisati nove programe. Iako su prvi računari bili skupoceni, mnoge kompanije su još više novca trošile na razvoj softvera, zbog kompleksnosti programiranja na niskom nivou. Već u to vreme postojala je ideja o razvoju apstraktnijih programskih jezika koji bi automatski bili prevođeni na mašinski jezik.

Polovinom 1950-ih godina nastali su *viši programski jezici*. Viši programski jezici sakrivaju detalje konkretnih računara od programera i drastično menjaju programiranje. U narednom periodu, dalji razvoj programskih jezika vodio je ka sve većem udaljavanju od hardvera i prilagođavanju programerima, sa ciljem da se programiranje učini lakšim, udobnijim i efikasnijim.

Da bi viši programski jezici mogli da se koriste postoje specijalizovani programi, tzv. *jezički procesori* ili *programski prevodioci*. Programski prevodioci omogućavaju lakšu komunikaciju čoveka i računara. Oni automatski prevode program napisan na višem programskom jeziku, dakle jeziku koji je blizak programeru, na mašinski jezik, jezik koji računar može da izvrši. Na taj način, programer ne mora da bude upućen u različite vrste arhitektura računara, a prenosivost napisanog programa se prebacuje sa programera na programski prevodilac. Razvoj programskih jezika usko je vezan sa razvojem programskih prevodilaca.

Ne postoji *najbolji* programski jezik već se za različite potrebe biraju različiti jezici, a svakodnevno se radi na unapređivanju postojećih i pravljenju novih programskih jezika. Programski jezici nastaju, razvijaju se i nestaju već više od sedamdeset godina. Neki izvori navode da je u realnoj upotrebi bilo do sada oko 250 jezika, a neki izvori koji pretenduju da popišu sve programske jezike koji su ikad postojali navode više od 9000 konkretnih jezika. Veliki broj programskih jezika rezultat je napora da se ubrza i olakša programiranje, odnosno da se nađu najbolji načini za rešavanje različitih praktičnih problema. Programer, u toku svog školovanja, pa i u toku svoje celokupne karijere, ne može da u potpunosti savlada sve bitne i aktuelne programske jezike. Međutim, ukoliko se programski jezici izučavaju uočavanjem bitnih svojstva i karakteristika postojećih jezika, to omogućava da se reletivno lako razumeju i po potrebi brzo savladaju novi programski jezici.

Ne postoji ni *najzastupljeniji* ili *najpopulariniji* programski jezik jer su polja i načini primene za neke jezike skoro neuporedivi. Indeks TIOBE (eng. *TIOBE Programming Community index*) još od davne 1985. godine meri popularnost programskih jezika koristeći razne faktore, kao što su broj programera u svetu koji vladaju nekim jezikom, broj univerzitetskih kurseva, broj kompanija koje koriste neki jezik i slično. Za rangiranje se koristi više popularnih pretraživača veba. Jezik C je dugo godina bio uvek na prvom ili drugom mestu, od 2000. godine jezik Java je takođe dugo bio na prvom ili drugom mestu, a C++ u prvih pet. U januaru 2021. godine poredak prvih osam jezika bio je ovakav: C, Java, Pajton, C++, C#, Visual Basic, JavaScript, PHP. U septembru 2024. godine poredak prvih osam jezika bio je ovakav: Python, C+, Java, C, C#, JavaScript, Visual Basic, Go. Po zastupljenosti koda u određenom programskom jeziku na portalu `github`, poredak je vrlo sličan. Među korisnicima foruma `stackoverflow` koji okuplja hiljade informatičara širom sveta, spisak najčešće korišćenih jezika u godini 2020, izgleda ovako: JavaScript, HTML/CSS, SQL, Python, Java, Bash/Shell, C#,

PHP, TypeScript, C++, C.

Razvojni ciklus jednostavnih programa savremenih viših programskih jezika teče na sledeći način[1]. Nakon faze razumevanja problema koji se rešava, prva faza u razvoju programa je njegovo *pisanje* kojim se u računar unosi *izvorni program* ili *izvorni kôd* (engl. *source code*) pomoću nekog editora teksta ili razvojnog okruženja. Naredna faza je njegovo *prevođenje*, kada se na osnovu izvornog programa na višem programskom jeziku dobija prevedeni kôd na asemblerskom odnosno mašinskom jeziku. Ovako dobijen kôd se naziva *objektni kôd* (engl. *object code*). U fazi *povezivanja* više objektnih programa povezuje se sa objektnim kodom iz standardne biblioteke u jedinstvenu celinu koja se naziva izvršivi program (engl. *executable program*). Povezivanje vrši specijalizovan program *povezivač* ili *uređivač veza*, koji se često naziva i *linker* (engl. *linker*). Faza prevođenja i faza povezivanja često nisu jasno razdvojene, odnosno prevodilac automatski poziva linker nakon faze prevođenja. Nakon povezivanja, kreiran je program u *izvršivom obliku* i on može da se *izvršava*. Nabrojane faze se obično ponavljaju, vrši se dopuna programa, ispravljanje grešaka, itd.

1.1 Klasifikacije programskih jezika

Najopštija podela viših programskih jezika je podela po načinu rešavanja problema. Po načinu rešavanja problema, programski jezici se dele na *proceduralne* i *deklarativne*. Većina programskih jezika danas je *proceduralna* što znači da je zadatak programera da precizno opiše način (proceduru) kojim se dolazi do rešenja problema.

Primer 1.1 (Proceduralno programiranje)**.** *Napisati program koji za unete različite cene artikala određuje koji je jeftiniji.*

Proceduralno rešenje ovog problema podrazumeva pisanje algoritma kojim se izračunava jeftiniji artikal. Alogoritam se može definisati na sledeći način.

```
Unesi cenu prvog artikala
Unesi cenu drugog artikala
Da li je cena prvog artika manja od cene drugog artikla?
Ako jeste, odštampaj "Prvi artikal je jeftiniji"
Inače, odštampaj "Drugi artikal je jeftiniji"
```

Ovaj algoritam se može prevesti u naredni Python kôd:

```
cena1 = int(input('Unesi cenu prvog artikla: '))
cena2 = int(input('Unesi cenu drugog artikla: '))

if cena1 < cena2:
    print('Prvi artikal je jeftiniji')
else:
    print('Drugi artikal je jeftiniji')
```

Pokretanjem prethodnog programa za ulazne vrednosti 1000 i 1500 dobija se naredni izlaz

```
Unesi cenu prvog artikla:
1000
Unesi cenu drugog artikla:
1500
Prvi artikal je jeftiniji
```

Značajni proceduralni programski jezici su, na primer, C, Pascal, Python i Java.

Nasuprot proceduralnim jezicima, *deklarativni* programski jezici od programera zahtevaju da precizno opiše problem, dok se mehanizam programskog jezika onda bavi pronalaženjem rešenja problema. Ovo u mnogome olakšava proces programiranja, međutim, zbog nemogućnosti automatskog pronalaženja efikasnih algoritama koji bi rešili široku klasu problema, domen primene deklarativnih jezika je često ograničen. Deklarativni jezici nisu među dominantnim jezicima opšte namene, ali se uspešno koriste u mnogim specifičnim domenima.

Primer 1.2 (Deklarativno programiranje)**.** *Napisati program koji pronalazi sva rešenja naredne kriptoaritmetike:*

[1] Ukoliko se razmatra razvoj netrivijalnog softvera, on obuhvata značajno veći broj faza koje ovde nisu pomenute.

```
ONE + ONE = TWO
```

U kriptoaritmetikama, različitim slovima odgovaraju različite cifre i početna cifra broja ne može biti nula.

Deklarativno rešenje ovog problema podrazumeva da se precizno opiše problem. Najpre je potrebno da se zadaju moguće vrednosti promenljivih O, N, E, T i W. To su vrednosti iz intervala $[0, 9]$*, pri čemu O i T ne mogu biti 0. Dalje je potrebno precizno napisati ograničenje koje treba da važi. To se može zapisati na sledeći način:*

```
  O*100 + N*10 + E
+ O*100 + N*10 + E
= T*100 + W*10 + O
```

U programskim jezicima koji imaju podršku za programiranje ograničenja *prethodni uslovi se mogu direktno preslikati u kôd. Na primer, u programskom jeziku Prolog, to se može zapisati na sledeći način*

```
crypto :-
    % Definisi promenljive O, N, E, T, W kao cifre od 0 do 9
    Digits = [O, N, E, T, W],
    Digits ins 0..9,

    % T i O ne smeju biti 0 (jer su brojevi TWO i ONE trocifreni)
    T #\= 0,
    O #\= 0,

    % Sve cifre moraju biti razlicite
    all_different(Digits),

    % Jednakost kroptoaritmetike: ONE + ONE = TWO
    O * 100 + N * 10 + E + O * 100 + N * 10 + E #= T * 100 + W * 10 + O,

    % Nadji resenje za svako slovo.
    label(Digits),

    % Odstampaj resenje.
    format('O = ~d, N = ~d, E = ~d, T = ~d, W = ~d~n', [O, N, E, T, W]),
    format('ONE = ~d~d~d, TWO = ~d~d~d~n', [O, N, E, T, W, O]).
```

Pokretanjem programa dobija se odgovor

```
O = 2, N = 0, E = 6, T = 4, W = 1
ONE = 206, TWO = 412
```

Značajniji deklarativni programski jezici su jezici Prolog i SQL.

1.1.1 Programske paradigme

Programske paradigme predstavljaju različite stilove programiranja koji često služe i za klasifikaciju programskih jezika. Paradigme se razlikuju po konceptima i apstrakcijama koje se koriste da bi se predstavili elementi programa (na primer, promenljive, funkcije, objekti, ograničenja) i koracima od kojih se sastoje izračunavanja (dodele, sračunavanja vrednosti izraza, tokovi podataka, itd.). Broj programskih paradigmi nije tako veliki kao broj programskih jezika. Izučavanje programskih paradigmi značajno olakšava razumevanje programskih jezika kao i očekivanja i mogućnosti koje jezik pruža. Poznavanje određene paradigme nam omogućava da brže i lakše savladamo svaki programski jezik koji toj paradigmi pripada.

Osnovne programske paradigme

U okviru programskih paradigmi, jasno su razgraničena četiri stila programiranja: *imperativno*, *funkcionalno*, *objektno-orjentisano* i *logičko* programiranje.

Imperativni jezici. Najkorišćeniji programski jezici danas spadaju u grupu *imperativnih* programskih jezika. U ovim jezicima stanje programa karakterišu *promenljive* kojima se predstavljaju podaci i *naredbe* kojima se vrše određene transformacije promenljivih.

Imperativni jezici razmatraju izračunavanje kao niz iskaza (naredbi) koje menjaju *stanje* programa određeno tekućim vrednostima promenljivih. Vrednosti promenljivih se menjaju naredbom dodele, a kontrola toka programa se vrši korišćenjem sekvence (nizanje naredbi), selekcije (izbor koja će naredba biti izvršena u zavisnosti od uspunjenosti nekog uslova) i iteracije (ponavljanje izvršavanja naredbi). Imperativni jezici su obično izrazito proceduralni.

Primer 1.3 (Programski jezik C)**.** *Program koji određuje koji je od dva artikla jeftiniji se na programskom jeziku C može napisati na sledeći način.*

```
#include<stdio.h>
int main() {
  int cena1, cena2;

  printf("Unesi cenu prvog artikla\n");
  scanf("%d", &cena1);
  printf("Unesi cenu drugog artikla\n");
  scanf("%d", &cena2);

  if(cena1 < cena2)
      print("Prvi artikal je jeftiniji\n");
  else
      printf("Drugi artikal je jeftiniji\n");

  return 0;
}
```

Imperativni jezici, uz objektno-orijentisane jezike, se najčešće koriste u industrijskom, sistemskom i aplikativnom programiranju. U nastavku su dati primeri najznačajnijih imperativnih jezika.

Fortran je prvi viši programski jezik koji je nastao u periodu 1953-1957 godine i koji je kasnije stekao širok krug korisnika. Ime Fortran nastalo je kao skraćenica od *FORmula TRANslating System* i dugo godina je pisano velikim slovima, kao akronim[2]. Projektovanje i razvoj Fortrana vodio je Džon Bakus (engl. *John Backus*) u okviru kompanije IBM. Prvi interpretator za Fortran bio je razvijen 1953. godine. Programiranje je postalo brže, ali novi programi su se izvršavali 10-20 puta sporije nego programi napisani na asembleru. Početna verzija kompilatora za Fortran I objavljena je nekoliko godina kasnije – 1956. godine i imala je oko 25000 linija asemblerskog koda. Kompilirani programi izvršavali su se skoro jednako brzo kao programi ručno pisani na asembleru. Pisanje programa ubrzano je i po 40 puta. Znatno je olakšano i održavanje programa zbog bolje čitljivosti i omogućena je prenosivost između različitih računara (za koje su postojali razvijeni Fortran kompilatori). Već 1958. više od polovine svih programa pisano je na Fortran-u. Ovaj jezik se, uz velike izmene u odnosu na prvobitne verzije, i danas koristi i namenjen je, pre svega, za numerička i naučna izračunavanja.

Cobol je nastao 1959. godine zajedničkom inicijativom nekoliko vodećih kompanija i univerziteta da se napravi jezik pogodan za izradu poslovnih, finansijskih i administrativnih aplikacija. Ime Cobol potiče od *COmmon Business Oriented Language*. Mnoge aplikacije pisane u Cobol-u su i danas u upotrebi, ali se Cobol više ne koristi za razvoj novih aplikacija.

Algol je jedan od najuticajnijih programskih jezika koji je nastao i razvijan od kraja pedesetih do početka sedamdesetih godina prošlog veka. On uvodi razne bitne karakteristike modernih programskih jezika a ime mu potiče od „ALGOrithmic Language“. U njegov razvoj bili su uključeni mnogi znameniti informatičari iz Evrope i Amerike.

Pascal je jedan od naslednika jezika Algol, koji je 1970. godine dizajnirao švajcarski informatičar Niklaus Virt (nem. *Niklaus Wirth*) kao mali i efikasan jezik koji ohrabruje korišćenje strukturiranog programiranja i drugih dobrih praksi programiranja. Jezik Pascal bio je veoma popularan tokom osamdesetih

[2]Ovo važi i za druge programske jezike, čija su imena najpre pisana velikim slovima jer su nastala kao akronimi. Na primer, to važi za jezike Cobol, Algol, Basic, Lisp i Prolog.

i devedesetih godina prošlog veka, ali je ipak smatrano da je njegovo glavno polje nastava programiranja (a ne i industrijske primene). Unapređenja i modifikacije jezika Pascal dovele su do jezika Modula, Oberon i Modula-2 (koje je, osamdesetih godina prošlog veka, razvio takođe Virt), kao i do objektno-orijentisanog jezika Object Pascal.

Basic je inicijalno razvijen 1964. godine za početnike u programiranju, za koje su Fortran i Algol bili previše kompleksni. Ime mu potiče od *Beginner's All-Purpose Symbolic Instruction Code*. Jezik je bio ekstremno jednostavan – prva verzija imala je samo četrnaest naredbi. Popularnost jezika porasla je sa pojavom mikro-računara i personalnih računara. U međuvremenu je razvijeno mnogo njegovih modifikacija i proširenja, od kojih je trenutno najpopularniji Visual Basic.

C je programski jezik opšte namene koji je 1972. godine razvio Denis Riči[3] u Belovim telefonskim laboratorijama (engl. *Bell Telephone Laboratories*) u SAD. Ime C dolazi od činjenice da je jezik nastao kao naslednik jezika B. C je jezik koji je bio namenjen prevashodno pisanju sistemskog softvera i to u okviru operativnog sistema Unix. Međutim, vremenom je počeo da se koristi i za pisanje aplikativnog softvera na velikom broju drugih platformi. C je danas prisutan na širokom spektru platformi — od mikrokontrolera do superračunara. Jezik C značajno je uticao i na razvoj drugih programskih jezika.

Funkcionalni jezici. Funkcionalno programiranje je u ekspanziji. Koncepti i ideje koje su nastale u okviru funkcionalnih jezika sve više prodiru u svakodnevno programiranje i u jezike koji u osnovi pripadaju drugim programskim paradigmama. Funkcionalni jezici razmatraju programiranje kao proces izračunavanja matematičkih funkcija. Koreni funkcionalnog programiranja leže u λ-računu razvijenom 1930-tih kako bi se izučavao pojam izračunljivosti i algoritma. Mnogi funkcionalni programski jezici mogu se smatrati nadogradnjama λ-računa.

Primeri značajnijih funkcionalnih programskih jezika su Lisp, Scheme, ML, Haskell, Erlang, Elixir i Elm. Funkcionalni jezici su najčešće jezici opšte namene. Veoma su koncizni i neki programeri ih svrstavaju u deklarativnu paradigmu.

Lisp je razvio Džon Makarti (engl. *John McCarthy*) 1958. godine na univerzitetu MIT i prvi je funkcionalni programski jezik. Ime Lisp nastalo je od *LISt Processing*, jer jezik podržava listu kao osnovnu strukturu podataka. Lisp je dugo smatran jezikom veštačke inteligencije. Od Lisp-a su se dalje razvili svi savremeni funkcionalni programski jezici. Neke varijante Lispa se i danas koriste.

Haskell je zvanično predstavljen 1990. godine. Osnovna ideja je bila da se standardizuju funkcionalni programski jezici koji su do tada postojali jer ih je bio veliki broj, a nijedan od njih nije bio široko prihvaćen. Haskell se danas često koristi u akademskim krugovima, ali ima i široku primenu u industriji za rešavanje složenih problema, posebno u oblastima gde su ispravnost i pouzdanost koda ključne.

Erlang je razvijen krajem 1980-ih u Ericssonu, švedskoj telekomunikacionoj kompaniji, sa ciljem da olakša izgradnju distribuiranih, skalabilnih i visoko dostupnih sistema koji su tada bili potrebni u telekomunikacijama. Erlang je zvanično pušten u javnost 1998. godine i od tada se koristi u raznim aplikacijama, kao što su telekomunikacioni sistemi, bankarstvo, i masivne online igre. Sa pojavom interneta, ideje koje su razvijane u okviru Erlanga dobijaju na značaju u novom kontekstu i razvijaju se novi programski jezici koji kao osnovu koriste Erlang.

Elixir je nastao 2011. godine sa ciljem da kombinuje robusnost i konkurentnost Erlanga sa modernijim i pristupačnijim sintaksnim konstrukcijama. Kao i Erlang, Elixir je odličan izbor za distribuirane sisteme koji zahtevaju visoku dostupnost, paralelno izvršavanje procesa i otpornost na greške, ali nudi modernije alate i sintaksu prilagođenu savremenom razvoju softvera. Elixir je naročito popularan u razvoju veb aplikacija. I Erlang i Elixir predstavljaju spoj funkcionalnog i konkurentnog programiranja.

Elm je nastao 2012. godine i napravljen je za razvoj korisničkih interfejsa na vebu. Koristi se često u kombinaciji sa jezikom Elixir, u kojem se programira logika sistema, dok se u Elmu programira korisnički interfejs.

Clojure je nastao 2007. godine sa ciljem da obezbedi jednostavan i moćan alat za konkurentno i paralelno programiranje. Clojure se zasniva na jeziku Lisp ali ima osobinu interoperabilonsti sa Javom, tj. izvršava se na javinoj virtulenoj mašini. To omogućava lak pristup postojećim Java bibliotekama i alatima, što ga čini moćnim za korišćenje u okruženjima gde je Java već prisutna. Takođe, stekao je popularnost među programerima koji cene minimalizam i jednostavnost Lisp sintakse.

Primer 1.4 (Programski jezik Haskell)**.** *Napisati program koji računa sumu kvadrata svih neparnih prirodnih brojeva čiji je kvadrat manji od 10000.*

[3]Dennis Ritchie (1941–2011), američki informatičar, dobitnik Tjuringove nagrade 1983. godine.

```
sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

Logički jezici. Logička paradigma je deklartivna paradigma koja se oslanja na matematičku logiku (konkretno, na metod rezolucije). Osnovni predstavnik ove paradigme je programski jezik Prolog, koji se često i koristi kao sinonim za logičku paradigmu. Prolog nije jezik opšte namene već se koristi u kontekstu tradicionalne veštačke inteligencije i predstavljanja znanja, za rešavanje logičkih problema, rezonovanje u sistemima koji su zasnovani na pravilima i u integraciji podataka i znanja.

Primer 1.5 (Zagonetka: programski jezik Prolog)**.** *Postoje tri kuće u nizu, svaka je obojena različitom bojom: crvena, plava i zelena. Svaki vlasnik kuće pije različito piće: vodu, čaj i kafu. Svaki vlasnik ima drugačiju životinju za kućnog ljubimca: mačku, psa i pticu.*

Tragovi:

Osoba u crvenoj kući ima mačku. Osoba u zelenoj kući pije kafu. Osoba u plavoj kući pije čaj. Osoba u zelenoj kući nema pticu.

Koristeći tragove, rešiti zagonetku: Ko poseduje psa?

```
resenje :-
  % Tri kuće: Kuca1, Kuca2, Kuca3
  Kuce = [Kuca1, Kuca2, Kuca3],

  % Svaka kuća je predstavljena listom [Boja, Pice, Ljubimac]
  % Inicijalizujemo promenljive za boje, pića i ljubimce
  Kuca1 = [Boja1, Pice1, Ljubimac1],
  Kuca2 = [Boja2, Pice2, Ljubimac2],
  Kuca3 = [Boja3, Pice3, Ljubimac3],

  % Postoje tri moguće boje: crvena, zelena i plava
  Boje = [crvena, zelena, plava],
  % Postoje tri moguća pića: voda, čaj, kafa
  Pica = [voda, caj, kafa],
  % Postoje tri moguća ljubimca: mačka, pas, ptica
  Ljubimci = [macka, pas, ptica],

  % Svaka kuća ima različitu boju, piće i ljubimca
  permutation(Boje, [Boja1, Boja2, Boja3]),
  permutation(Pica, [Pice1, Pice2, Pice3]),
  permutation(Ljubimci, [Ljubimac1, Ljubimac2, Ljubimac3]),

  % Trag 1: Osoba u crvenoj kući ima mačku.
  member([crvena, _, macka], Kuce),

  % Trag 2: Osoba u zelenoj kući pije kafu.
  member([zelena, kafa, _], Kuce),

  % Trag 3: Osoba u plavoj kući pije čaj.
  member([plava, caj, _], Kuce),

  % Trag 4: Osoba u zelenoj kući nema pticu.
  not(member([zelena, _, ptica], Kuce)),

  % Određivanje ko ima psa i ispis rezultata
  member([Boja, Pice, pas], Kuce), % Pronađi kuću sa psom
  format('Osoba koja ima psa živi u kući koja je ~w i pije pice ~w.~n', [Boja, Pice]).
```

Rešenje zagonetke je

```
Osoba koja ima psa živi u kući koja je zelena i pije pice kafa.
```

Objektno-orijentisani jezici. *Objekti* su specijalizovane strukture podataka koje uz polja podataka sadrže i metode kojima se manipuliše tim podacima. Podaci se mogu obrađivati isključivo primenom metoda što smanjuje zavisnosti između različitih komponenata programskog koda i čini ovu paradigmu pogodnu za razvoj velikih aplikacija uz mogućnost saradnje većeg broja programera. Najčešće korišćene tehnike programiranja u objektno orijentisanom programiranju uključuju sakrivanje informacija, enkapsulaciju, apstraktne tipove podataka, modularnost, nasleđivanje i polimorfizam. Značajniji objektno-orijentisani jezici su C++, Java i C#.

Primer 1.6 (Programski jezik Java)**.** *Osnovne karakteristike svake osobe su njeno ime i broj godina. Svaka osoba ume da se predstvi, odnosno da saopšti svoje ime i broj godina. Napisati program u kojem se kreiraju i predstavljaju dve osobe.*

```
class Osoba {
    private String ime;
    private int godine;

    // Konstruktor
    public Osoba(String ime, int godine) {
        this.ime = ime;
        this.godine = godine;
    }

    // Metod za pozdrav
    public void predstaviSe() {
        System.out.println("Zdravo, ja sam " + ime + " i imam " + godine + " godina.");
    }

    public static void main(String[] args) {
        // Kreiranje objekata
        Osoba osoba1 = new Osoba("Ana", 30);
        Osoba osoba2 = new Osoba("Marko", 25);

        // Pozivanje metoda za predstavljanje
        osoba1.predstaviSe();
        osoba2.predstaviSe();
    }
}
```

C++ je kreirao Bjern Stroustrup (danski informatičar) 1986. godine kao direktni naslednik jezika C. C++ se, u trenutku nastanka, mogao smatrati njegovim objektno-orijentisanim proširenjem. C++ je i dalje jedan od najpopularnijih jezika i koristi se za razvoj zahtevnih aplikacija, s jedne strane zbog svojih objektno-orijentisanih svojstava, a s druge zbog bliske veze sa mašinom, u duhu jezika C. U izvesnom smislu, potomkom i unapređenjem jezika C++ može se smatrati jezik C#, koji je razvijen 2000. godine.

Java je objektno-orijentisani programski jezik razvijen 1995. godine sa motivom da bude što manje zavisnosti za fazu izvršavanja i da se kompilirani Java kôd (takozvani bajtkod) može izvršavati na bilo kojoj platformi koja podržava Javu (tj. koja raspolaže Java virtuelnom mašinom) bez ponovnog kompiliranja. Sintaksa jezika Java slična je jezicima C i C++ ali ima manje operacija niskog nivoa. Java omogućava modifikacije koda u fazi izvršavanja. Java je trenutno jedan od najpopularnijih programskih jezika.

C# je nastao oko 2000. godine u kompaniji Micorosoft kao deo njihove .NET inicijative. Po svojim karakteristikama je donekle sličan programskom jeziku Java. Zamišljen kao moderni objektno-orijentisani programski jezik opšte namene koji se karakteriše relativnom jednostavnošću programiranja. Jezik se stalno obogaćuje i unapređuje. I dalje je veoma popularan izbor za programiranje aplikacija za Windows i veb aplikacije.

Objective C i Swift su programski jezici koji se koriste za razvoj aplikacija na platformama kompanije *Apple*. Objective-C je stariji jezik koji je razvijen krajem 1980-ih, kao proširenje programskog jezika C, dodajući objektno orijentisane karakteristike. Iako je Objective-C bio glavni jezik za razvoj u kompaniji *Apple* dugi niz godina, njegova relativno složena i zastarela sintaksa dovela je do potrebe za modernijom, bržom i sigurnijom alternativom koja je predstavljena 2014. godine kao jezik Swift. Swift je danas primarni jezik za razvoj aplikacija na *Apple* platformama, a zbog svojih modernih karakteristika brzo je postao popularan među programerima.

Savremene programske paradigme

Savremeni programski jezici su multiparadigmatski, odnosno u sebi sadrže više različitih stilova programiranja. Savremene programske paradigme uključuju *skript*, *komponentno*, *generičko*, *konkurentno* i *vizuelno* programiranje, kao i paradigme *upitnih jezika* i *programiranja ograničenja*.

Skript programiranje je oblik programiranja koji se koristi za pisanje kratkih, jednostavnih programa ili „skripti“ koje automatizuju zadatke ili upravljaju funkcijama u većim softverskim sistemima. Skript programiranje je vrlo korisno za zadatke u kojima je važna brzina razvoja, fleksibilnost i jednostavnost, čineći ga izuzetno popularnim u raznim oblastima IT-a. U okviru samog skript programiranja, izdvajaju se skript jezici koji se koriste u domenu veb programiranja, skript jezici opšte namene, skript jezici za procesiranje teksta, komandni jezici i jezici sa specifičnim domenom (na primer, za matematiku i statistiku).

Perl je nastao sredinom osamdesetih godina prošlog veka. Jedna od njegovih ključnih karakteristika su moćne funkcije za obradu teksta, a koristio se u radu sa tekstualnim podacima, sa bazama podataka, u mrežnom programiranju i slično, ali i kao skript jezik za Linux sisteme.

Python je multiparadigmatski jezik jednostavne i izražajne sintakse čija prva verzija je objavljena 1991. godine. Ima elemente objektno-orijentisanih, imperativnih, funkcionalnih jezika itd. Raspolaže ugrađenim strukturama podataka visokog nivoa. Pošto se programi interpretiraju, osnovni ciklus razvoja programa (pisanje-testiranje-debagovanje) odvija se izuzetno brzo. Python je najpopularniji jezik u oblastima kao što su istraživanje podataka i mašinsko učenje, ali se koristi i za veb-aplikacije, za mobilne aplikacije, kao i za ugrađene sisteme. U nastavi je Python na mnogim mestima kao prvi programski jezik zamenio Pascal, Javu i C.

PHP je nastao 1994. godine kao internet jezik koji može biti ugrađen u HTML kôd. Ime PHP potiče od *Hypertext Preprocessor*. PHP kôd se izvršava na serveru i dinamički generiše HTML sadržaj koji se šalje i prikazuje klijentu. Više od polovine svih veb sajtova kao jezik na strani servera koriste PHP u nekom obliku.

JavaScript je nastao 1995. godine i obično se koristi za programe koji se izvršavaju na strani klijenta (na primer, u okviru pregledača veba). JavaScript omogućava da se sadržaj veb strana menja interaktivno, na primer, u zavisnosti od akcija korisnika. JavaScript danas koristi većina veb strana. JavaScript i PHP se koriste skladno – prvi na strani klijenta, a drugi na strani servera.

ASP.NET je nastao 2002. godine, kao skript jezik za veb aplikacije i dinamičko kreiranje veb sadržaja kompanije Microsoft. Ima dosta sličnosti sa jezikom PHP, ali može da se izvršava samo na Windows serverima.

Lua je jezik opšte namene koji je nastao 1993. godine u Brazilu. Jezik Lua je poznat po jednostavnoj sintaksi, efikasnosti i širokoj primeni, naročito u razvoju video igara. Lua podržava proceduralno, objektno orijentisano i funkcionalno programiranje, kao i programiranje vođeno podacima.

Ruby je jezik opšte namene koji se pojavio 1995. godine, u Japanu. Otovorenog je koda, sa fokusom na jednostavnost i produktivnost. Osnovna ideja dizajna jezika je da se adresiraju ljudske potrebe (a ne potrebe računara), odnosno da programiranje učini programere zadovoljnim, produktivnim i srećnim.

Bash je nastao 1987. godine u okviru GNU projekta. To je skriptni jezik za Unix i Unix-olike operativne sisteme, kao što su Linux i macOS. Bash omogućava korisnicima da izvršavaju komande direktno u komandnoj liniji. Koristi se za pisanje skripti koje mogu automatizovati razne zadatke, kao što su instalacija softvera, upravljanje datotekama i izvođenje sistemskih operacija. Bash se često korsiti za administraciju sistema i upravljanje serverima. Različite distribucije Linux-a dolaze sa Bash-om kao podrazumevanim jezikom komandne linije.

R i S su skrpit jezici specifične namene razvijeni za statističku analizu, vizualizaciju podataka i naučno istraživanje. Jezik S je razvijen 1976. godine u Bell Labs-u i komercijalni je jezik koji je poslužio

kao inspiracija za kreiranje jezika R. R je objavljen kao otvoreni softver 1995. godine. R je veoma popularan u akademskim i istraživačkim krugovima, kao i u industrijama koje se bave analizom podataka.

Primer 1.7. *Učenje novog programskog jezika obično započinje razumevanjem programa koji štampa poruku* Hello, world!*. U nastavku su dati primeri ovog programa u različitim skript jezicima. Možemo da zaključimo kako su svi ovi primeri veoma slični.*

Programski jezik Perl

```
print "Hello, World!\n";
```

Programski jezik Python

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

Programski jezik PHP

```
<?php
echo "Hello, World!";
```

Programski jezik JavaScript

```
console.log("Hello, World!");
```

Programski jezik Lua

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

Programski jezik Ruby

```
puts "Hello, World!"
```

Programski jezik Bash

```
echo "Hello, World!"
```

Konkurentno programiranje je pristup pisanju koda koji omogućava izvršavanje više zadataka u istom vremenskom intervalu (bilo da je to istovremeno, onda kada postoji odgovarajuća harverska podrška na raspolaganju, ili samo vremenski ispreplitano, onda kada je za izvršavanje na raspolaganju samo jedan procesor). Konkurentno programiranje pomaže u optimizaciji korišćenja resursa i povećanju performansi aplikacija. Ovaj pristup je ključan u razvoju modernih softverskih sistema, posebno u aplikacijama koje obrađuju velike količine podataka, zahtevaju visoku dostupnost ili performanse.

Iako je konkurentnost razvijana još od jezika Algol, velika popularnost i potreba za konkurentnim programiranjem nastaje početkom 21. veka dostupnošću višeprocesorskih mašina i napretkom razvoja računarskih mreža. Podrška za konkurentno programiranje postoji dostupna gotovo u svim jezicima opšte namene, ali postoje i jezici koji su se razvijeni baš sa ciljeme da se olakša i unapredi ovaj stil programiranja.

Go (poznat i kao *Golang*) je programski jezik koji su razvili inženjeri kompanije Google 2007. godine, sa ciljem da stvore jezik koji kombinuje performanse i efikasnost jezika niskog nivoa poput C-a sa jednostavnošću i brzinom razvoja koje nude moderni jezici. Prva stabilna verzija objavljena je 2012. godine, a od tada je Go postao veoma popularan, naročito za razvoj servera, mrežnog softvera i za distribuiranu obradu podataka.

Rust je programski jezik koji se ističe obezbeđivanjem sigurnosti u radu sa memorijom, performansama i podrškom za razvoj konkurentnih aplikacija. Razvila ga je kompanija *Mozilla* 2010. godine, a danas ga razvija i održava *Rust Foundation*. Rust je osmišljen kao bezbedna i brza alternativa jezicima poput C i C++, sa posebnim fokusom na onemogućavanje grešaka u radu sa memorijom. Rust je brzo postao popularan među programerima koji rade na visokoperformantnim aplikacijama ili softveru gde su stabilnost i sigurnost kritični. Koristi se za razne projekte, uključujući veb servere, igre, alate komandne linije i komponente operativnih sistema kao što je Linux kernel.

Kotlin je moderan programski jezik koji se izvršava na Java virtuelnoj mašini i koristi se primarno za razvoj Android aplikacija. Razvila ga je kompanija *JetBrains* 2011. godine, dok je prva stabilna verzija objavljena 2016. godine. *Google* je 2017. godine zvanično podržao Kotlin kao osnovni jezik za razvoj Android aplikacija, što je značajno doprinelo njegovoj popularnosti. Kotlin je postao popularan izbor za Android razvoj zbog svoje čitljivosti, smanjenja količine koda i ugrađene sigurnosti.

Primer 1.8 (Konkurentno sabiranje elemenata niza u jeziku Go)**.** *Izračunavanje zbira elemenata u nizu može se podeliti tako da se izvršava konkurentno za različite delove niza, i zatim se dobijeni zbirovi podnizova saberu. Primer programa koji izračunavanje zbira elemenata u nizu izvršava podelom niza na dva dela dat je u nastavku.*

```
package main

import (
    "fmt"
    "sync"
)

// Funkcija za sabiranje elemenata dela niza
func sumPart(arr []int, result *int, wg *sync.WaitGroup) {
    defer wg.Done()
    partialSum := 0
    for _, v := range arr {
        partialSum += v
    }
    *result = partialSum
}

func main() {
    arr := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    var wg sync.WaitGroup

    // Promenljive za čuvanje rezultata sabiranja
    var sum1, sum2 int

    // Duzina jednog podniza
    mid := len(arr) / 2

    // Delimo niz na dva dela i pokrećemo sabiranje u dve gorutine
    wg.Add(2)
    go sumPart(arr[:mid], &sum1, &wg) // sabira prvu polovinu niza
    go sumPart(arr[mid:], &sum2, &wg) // sabira drugu polovinu niza

    // Čekamo da obe gorutine završe
    wg.Wait()

    // Konačna suma
    totalSum := sum1 + sum2
    fmt.Printf("Suma elemenata niza je: %d\n", totalSum)
}
```

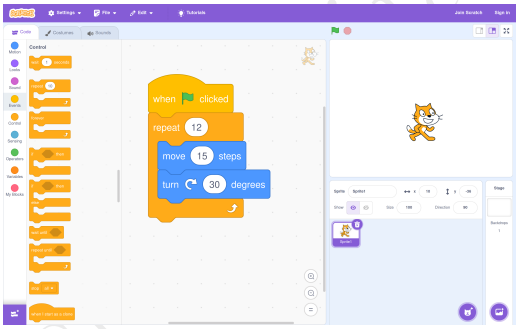
Komponentno programiranje je pristup razvoju softvera koji se fokusira na izgradnju aplikacija korišćenjem nezavisnih, ponovo upotrebljivih komponenti. Ove komponente mogu biti različitih vrsta, uključujući

biblioteke, module ili čak mikroservise, a sve zajedno omogućavaju brži i efikasniji razvoj aplikacija. Komponentno programiranje najčešće karakteriše pristup *prevuci i postavi* (eng. *drag and drop*), tj. pristup da se kôd ne kuca u potpunosti u tekstualnom editoru, već da se najveći deo koda automatski generiše prevlačenjem i povezivanjem odgovarajućih kompnenti. Komponentno programiranje je slično objektno-orjentisanom programiranju, ali su komponente veće programske celine u odnosu na klase. Komponentno programiranje se može ostvariti u različitim programskim jezicima opšte namene (npr. C, C++, Java, C#, Swift), kroz razvojna okruženja i biblioteke.

Primer 1.9. *Grafički korisnički interfejsi se najčešće prave komponentnim stilom programiranja. Jedna od početnih komponenti može da bude prozor na koji se onda dodaju različite druge komponente, na primer dugmići, tekstualna polja, tekst i slično.*

Vizuelno programiranje koristi grafičko okruženje za kreiranje programa, umesto pisanja koda u tekstualnom obliku. U vizuelnom programiranju, programeri koriste vizuelne elemente kao što su blokovi, ikone, linije i dijagrami za pravljenje programa. Ovaj pristup je često intuitivniji i pristupačniji, posebno za početnike, jer omogućava manipulaciju objekata i logike programa na grafički način. Primeri vizuelnog programiranja za početnike su okruženja *Scratch* i *Blockly*, dok se alati *Node-RED*, *LabVIEW* i *Unreal Engine Blueprints* koriste u industrijskom okruženju.

Primer 1.10. *Razvojno okruženje za* Scratch *je zajedno sa različitim tutorijalima besplatno dostupno na internetu* `https://scratch.mit.edu/projects/editor/`. *Naredna slika prikazuje razvojno okruženje i program koji omogućava pokretanje mačke (u gornjem desnom uglu) u krugu za 360 stepeni u smeru kazaljke na satu.*



Generičko programiranje je paradigma koja omogućava pisanje koda koji može da se upotrebljava sa različitim tipovima podataka. Osnovna ideja je da se algoritmi i strukture podataka mogu definisati na uopšten način, tako da rade sa bilo kojim tipom podataka, dok god taj tip zadovoljava određene zahteve. Ovakav pristup programiranju obezbeđuje fleksibilnost i ponovnu upotrebljivost napisanog koda. Primeri programskih jezika koji podržavaju različite vidove generičkog programiranja su C++, Java i C#.

Primer 1.11 (Generička funkcija za pronalaženje maksimumam u jeziku C++)**.** *Umesto konkretnog tipa, koristi se tipska promenljiva T, koja omogućava apstrahovanje tipa iz funkcije i upotrebu funkcije na svim tipovima koj imaju definisan operator poređenja* `>`. *Na primer, ovako definisana funkcija može se na isti način koristiti za cele brojeve, realne brojeve, i za karaktere.*

```
#include <iostream>
using namespace std;

// Generička funkcija koja računa veću od dve promenljive
template <typename T>
T maksimum(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << maksimum(110, 210) << endl;     // Celi brojevi
    cout << maksimum(5.7, 5.3) << endl;     // Realni brojevi
    cout << maksimum('g', 'e') << endl;     // Karakteri
    return 0;
```

```
}
```

Upitni jezici pripadaju deklarativnoj paradigmi. To su specijalizovani programski jezici koji se koriste za interakciju sa bazama podataka. Oni omogućavaju korisnicima da efikasno pretražuju, manipulišu i upravljaju podacima u različitim vrstama baza podataka. Najpoznatiji upitni jezik je SQL, koji se koristi za rad sa *relacionim* bazama podataka. Osim SQL-a, postoje i drugi upitni jezici, kao što su SPARQL za RDF podatke i XQuery za XML podatke.

SQL je akronim od eng. *Structured Query Language*). Prva verzija jezika se pojavila 1970-ih godina. Nakon toga, SQL je postao osnovni alat za rad sa relacionim bazama podataka, omogućavajući definisanje, manipulaciju i kontrolu podataka na jednostavan i efikasan način. SQL se koristi za poslovnu analitiku, razvoj najrazličitijih aplikacija i rad sa velikim količinama podataka.

SPARQL (eng. *SPARQL Protocol and RDF Query Language*) je jezik za postavljanje upita nad RDF (eng. *Resource Description Framework*) podacima, koji su osnova za rad sa semantičkim vebom i povezanim podacima (eng.*linked data*). SPARQL je razvio W3C (eng. *World Wide Web Consortium*).

XQuery (eng. *XML Query Language*) je upitni jezik dizajniran za pretraživanje i manipulaciju XML podacima. Omogućava korisnicima da efikasno pretražuju i izvode informacije iz XML dokumenata, kao i da transformišu te podatke. XQuery je standardizovan od strane W3C.

Primer 1.12. *Naredni SQL upit iz baze podataka koja sadrzi tabelu* `Osobe` *izvlači sve informacije o svim osobama starijim od 24 godine.*

```
SELECT * FROM Osobe WHERE Godine > 24;
```

Programiranje ograničenja je deklarativna paradigma u okviru koje je posao programera da detaljno opiše uslove u sistemu za koji se traži rešenje. Koristi se u domenu optimizacija i rešavanja kombinatornih problema. Kao veoma praktična i potrebna tehnika programiranja, veliki broj programskih jezika opšte namene ima biblioteke koje pružaju podršku za programiranje u ovom stilu (jezici C, C++, Java, Python, C# i mnogi drugi).

Primer 1.13. *Za primer 1.2 koji ilustruje programiranje ograničenja u programskom jeziku Prolog, u nastavku je dat odgovarajući program u programskom jeziku Python. Možemo da primetimo da se ova dva programa ne razlikuju suštinski.*

```
from constraint import Problem, AllDifferentConstraint

# Definisi funkciju ogranicenja kriptoaritmetike
def equation_constraint(o, n, e, t, w):
    # ONE + ONE should equal TWO
    return (o * 100 + n * 10 + e) * 2 == t * 100 + w * 10 + o

# Definisi funkciju ogranicenja da je vrednost razlicita od nule
def non_zero_constraint(value):
    return value != 0

# Napravi instacu problema
problem = Problem()

# Definisi promenljive O, N, E, T, W kao cifre od 0 do 9
problem.addVariables("ONETW", range(10))

# Dodaj ogranicenje da O i T ne mogu biti 0 (jer su brojevi TWO i ONE trocifreni)
problem.addConstraint(non_zero_constraint, ["OT"])

# Sve cifre moraju biti razlicite
problem.addConstraint(AllDifferentConstraint(), "ONEWT")

# Jednakost kroptoaritmetike: ONE + ONE = TWO
problem.addConstraint(equation_constraint, "ONEWT")
```

```
# Izracunaj i odstampaj resenja
solutions = problem.getSolutions()
if solutions:
    for solution in solutions:
        print(f"Solution found: O={solution['O']}, N={solution['N']}, E={solution['E']}, "
              f"T={solution['T']}, W={solution['W']}")
else:
    print("Resenje nije nadjeno.")
```

Jezici za obeležavanje podataka/teksta

Jezici za obeležavanje podataka ili teksta, kao što su to, na primer, HTML/CSS, XML i LaTeX, iako veoma bitni u kontekstu programiranja, nisu programski jezici i kao takvi ne navodimo ih u klasifikaciji programskih jezika. Ovim jezicima definiše se struktura teksta ili podataka, ne pišu se programi koji definišu izvršavanje, što je ključno da bi se neki jezik smatrao programskim jezikom. Ovi jezici služe da se podaci ili tekst sistematično označe i da onda, na osnovu tih oznaka, specijalizovani programi mogu da obrađuju te podatke ili da prikažu tekst na odgovarajući način.

Primer 1.14. *HTML definiše strukturu veb stranice i određuje naslove, podnaslove paragrafe i slično. Veb pregledač na osnovu tih informacija (i eventulano dodatnih stilskih informacija kroz CSS) određuje vizuelni prikaz veb stranice. Na primer, naredni tekst je obeležen:*

```
<h1>Ljubičica</h1>
<p>Ljubičice su uglavnom višegodišnje zeljaste biljke. One imaju cetove
zanimljivih  arhitektura i boja, usled čega se mnoge vrste uzgajaju
kao ukrasne.</p>
```

Oznaka `<h1>` *obeležava početak teksta naslova, dok oznaka* `</h1>` *obeležava kraj teksta koji pripada tom naslovu. Slično, oznaka* `<p>` *obeležava početak teksta paragrafa, dok oznaka* `</p>` *obeležava kraj teksta koji pripada tom paragrafu. Različiti veb pregledači mogu da prikažu ovaj tekst na različite načine.*

Pitanja i zadaci za vežbu

Pitanje 1.1. *Koja su mane mašinski zavisnih programskih jezika?*

Pitanje 1.2. *Koje su prednosti viših programskih jezika?*

Pitanje 1.3. *Zašto postoji veliki broj programskih jezika?*

Pitanje 1.4. *Čemu služe programskih prevodioci?*

Pitanje 1.5. *Šta meri TIOBE indeks?*

Pitanje 1.6. *Koja je osnovna razlika između proceduralnog i deklarativnog programiranja? Navesti primere jezika koji pripadaju proceduralnoj i jezika koji pripadaju deklarativnoj paradigmi.*

Pitanje 1.7. *Koje su četiri osnovne programske paradigme? Navesti osnovne karakteristike svake od njih.*

Pitanje 1.8. *Koja četiri značajna programska jezika su prva istorijski nastala? Kojim paragimama pripadaju ti jezici?*

Pitanje 1.9. *Koje su osnovne karakteristike imperativne paradigme? Navesti bar četiri jezika koji pripadaju imperativnoj paradigmi.*

Pitanje 1.10. *Koje su osnovne karakteristike funkcionalne paradigme? Navesti bar četiri jezika koji pripadaju funkcionalnoj paradigmi.*

Pitanje 1.11. *Koje su osnovne karakteristike objektno-orjentisane paradigme? Navesti bar četiri jezika koji pripadaju objektno-orjentisanoj paradigmi.*

Pitanje 1.12. *Koje su osnovne karakteristike logičke paradigme? Koji jezik se koristi kao sinonim za logičko programiranje?*

Pitanje 1.13. *Koje su osnovne karakteristike i ciljevi skript programiranja? Koji poddomeni skript programiranja postoje? Koji jezici podržavaju ovu paradigmu?*

Pitanje 1.14. *Koje su osnovne karakteristike i ciljevi konkurentnog programiranja? Kada je nastalo konkurentno programiranje i kako je dobilo na značaju? Koji jezici podržavaju ovu paradigmu? Koji funkcionalni programski jezici podržavaju konkurentno programiranje?*

Pitanje 1.15. *Koje su osnovne karakteristike i domeni upotrebe komponentnog programiranja? Kojoj osnovnoj paradigmi je komponentno programiranje najsličnije i zašto?*

Pitanje 1.16. *Koje su osnovne karakteristike i ciljevi vizuelnog programiranja? Koja okruženja podržavaju ovu paradigmu?*

Pitanje 1.17. *Koje su osnovne karakteristike i ciljevi generičkog programiranja? Koji jezici podržavaju ovu paradigmu?*

Pitanje 1.18. *Koje su osnovne karakteristike i domeni upotrebe paradigme upitnih jezika? Koji jezici podržavaju ovu paradigmu?*

Pitanje 1.19. *Koje su osnovne karakteristike i domeni upotrebe paradigme programiranja ograničenja? Koji jezici podržavaju ovu paradigmu?*

Pitanje 1.20. *Zašto jezici za obeležavanje teksta ne pripadaju programskim paradigmama? Koji su najbitniji jezici za obeležavanje teksta?*

Pitanje 1.21. *Navesti primer logičkog jezika, primer funkcionalnog programskoj jezika i primer objektno-orijentisanog programskog jezika.*

Pitanje 1.22. *Kojoj grupi jezika pripada Lisp? Kojoj grupi jezika pripada Prolog? Kojoj grupi jezika pripada C++? Kojoj grupi jezika pripada Hasklell? Kojoj grupi jezika pripada C? Kojoj grupi jezika pripada Go?*

Pitanje 1.23. *Za veliki broj savremenih programskih jezika kažu da su muliparadigmatski (engl. multiparadigm). Šta to znači?*

Pitanje 1.24. *Na kojoj logičkoj metodi je zasnovan jezik Prolog?*

Pitanje 1.25. *Na kojem formalizmu izračunavanja je zasnovan jezik Lisp?*

Pitanje 1.26. *U objektno-orijentisanoj paradigmi, šta čini jednu klasu?*

1.2 Leksika, sintaksa, semantika i pragmatika programskih jezika

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

1.2.1 Leksika

Osnovni leksički elementi prirodnih jezika su reči, pri čemu se razlikuje nekoliko različitih vrsta reči (imenice, glagoli, pridevi, ...) i reči imaju različite oblike (padeži, vremena, ...). Zadatak leksičke analize prirodnog jezika je da identifikuje reči u rečenici i svrsta ih u odgovarajuće kategorije. Slično važi i za programske jezike. Programi se računaru zadaju predstavljeni nizom karaktera. Pojedinačni karakteri se grupišu u nedeljive celine koje predstavljaju osnovne leksičke elemente, koji bi bili analogni rečima govornog jezika.

Primer 1.15. *Leksika jezika* UR *mašina je veoma jednostavna i sadrži samo četiri rezervisane reči (Z, S, J, T) i brojevne konstante.*

Primer 1.16. *Razmotrimo naredni fragment koda u jeziku C++:*

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

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

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

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

1.2.2 Sintaksa

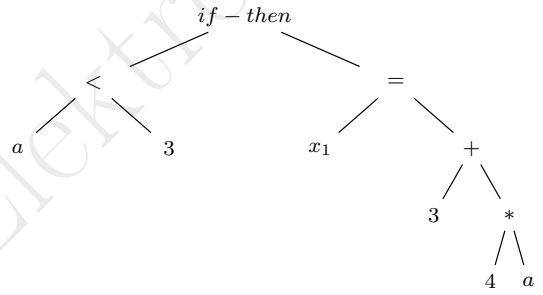
Sintaksa prirodnih jezika definiše načine na koji pojedinačne reči mogu da kreiraju ispravne rečenice jezika. Slično je i sa programskim jezicima, gde se umesto ispravnih rečenica razmatraju ispravni programi.

Primer 1.17. *Sintaksa jezika* UR *mašina definiše ispravne programe kao nizove instrukcija oblika:* $Z(\text{broj})$, $S(\text{broj})$, $J(\text{broj}, \text{broj}, \text{broj})$ *i* $T(\text{broj}, \text{broj})$. *Na primer, jedan sintaksno ispravan program je*

```
1. Z(0)
2. S(0)
```

Sintaksa definiše formalne relacije između elemenata jezika, time pružajući strukturne opise ispravnih niski jezika. Sintaksa se bavi samo formom i strukturom jezika bez bilo kakvih razmatranja u vezi sa njihovim značenjem. Sintaksička struktura rečenica ili programa se često predstavlja u obliku stabla.

Primer 1.18. *Fragment koda iz primera 1.16 je u jeziku C++ sintaksički ispravan i njegova sintaksička struktura se može predstaviti na sledeći način:*



Dakle, taj fragment koda predstavlja `if-then` *naredbu kojoj je uslov dat izrazom poređenja vrednosti promenljive* `a` *i konstante* `3` *(levo podstablo na slici), a telo (desno podstablo na slici) predstavlja naredba dodele promenljivoj* `x1` *vrednosti izraza koji predstavlja zbir konstante* `3` *i proizvoda konstante* `4` *i vrednosti promenljive* `a`.

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

Semantika

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

Primer 1.19. *Za sintaksno ispravan* UR *program iz primera 1.17 može se pridružiti sledeće značenje: „Postavi vrednost nultog registra na nulu i zatim ga uvećaj za jedan.“*

Primer 1.20. *Za kôd iz primera 1.16 jezika C++ može se pridružiti sledeće značenje: „Ako je tekuća vrednost promenljive* `a` *manja od 3, tada promenljiva* `x1` *treba da dobije vrednost zbira broja 3 i četvorostruke tekuće vrednosti promenljive* `a`*“.*

Postoje sintaksički ispravne rečenice prirodnog jezika kojima nije moguće dodeliti ispravno značenje, tj. za njih nije definisana semantika, na primer: „Bezbojne zelene ideje besno spavaju“ ili „Pera je oženjeni neženja“. Slično je i sa programskim jezicima. Sintaksno ispravni programi ne moraju da imaju ispravno značenje.

Primer 1.21. *Za naredni C++ kôd ne može se definisati ispravno značenje i zbog toga se program ne može prevesti u izvršivi kôd.*

```
int x = 0;
int x = 5;
```

Iako su oba reda sintaksno ispravna, kôd je semantički neispravan jer se dva puta deklariše promenjiva sa istim imenom.

Neki aspekti semantičke korektnosti programa se mogu proveriti tokom prevođenja programa (na primer, da su sve promenljive koje se koriste u izrazima definisane i da su odgovarajućeg tipa), dok se neki aspekti mogu proveriti tek u fazi izvršavanja programa (na primer, da ne dolazi do deljenja nulom). Na osnovu toga, razlikuje se *statička* i *dinamička* semantika.

Primer 1.22. *Za naredni C++ kôd može se definisati statičko značenje pa u skladu sa time program se može prevesti u izvršivi kôd. Međutim, za njega se ne može definisati dinamičko značenje, pa u fazi izvršavanja dolazi do greške.*

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

Dok većina savremenih jezika ima precizno i formalno definisanu leksiku i sintaksu, formalna definicija semantike postoji samo za neke programske jezike. U ostalim slučajevima, semantika programskog jezika se opisuje neformalno, opisima zadatim korišćenjem prirodnog jezika. Čest je slučaj da neki aspekti semantike ostaju nedefinisani standardom jezika i prepušta se implementacijama programskih prevodilaca da samostalno odrede potpunu semantiku.

Primer 1.23. *Programski jezik C ne definiše kojim se redom vrši izračunavanje vrednosti izraza, što u nekim slučajevima može da dovede do različitih rezultata istog programa prilikom prevođenja i izvršavanja na različitim sistemima. Na primer, nije definisano da li se za izračunavanje vrednosti izraza* `f() + g()` *najpre poziva funkcija* `f` *ili funkcija* `g`*.*

1.2.3 Pragmatika programskih jezika

Pragmatika jezika govori o izražajnosti jezika i o odnosu različitih načina za iskazivanje istih stvari. Pragmatika prirodnih jezika se odnosi na psihološke i sociološke aspekte kao što su korisnost, opseg primena i efekti na korisnike. Isti prirodni jezik se koristi drugačije, na primer, u pisanju tehničkih uputstava, a drugačije u pisanju pesama.

Pragmatika programskih jezika uključuje pitanja kao što su lakoća programiranja, efikasnost u primenama i metodologija programiranja. Pragmatika je ključni predmet interesovanja onih koji dizajniraju i implementiraju programske jezike, ali i svih koji ih koriste.

U kontekstu programera, pragmatika programskog jezika odnosi se i na praktičnu upotrebu jezika u stvarnim situacijama programiranja, tj. kako se jezik koristi u praksi za postizanje specifičnih ciljeva, kao što su efikasnost, čitljivost i održivost koda. Pragmatika se bavi i pitanjem kako i zašto programeri koriste određene elemente jezika u različitim kontekstima. U nastavku su dati neki primeri pragmatičnih izbora u programskom jeziku C. Biraju se različiti elementi jezika u zavisnosti od specifičnih potreba i problema koje rešavaju.

Primer 1.24. *U programskom jeziku C++ postoje različite vrste petlji. Pragmatika programskog jezika razmatra kako programer odlučuje da li će za neki program koristiti* `for` *ili* `while` *petlju. Na primer,* `for` *petlja se upotrebljava kada za ponavljanje postoji očekivani broj izvršavanja, dok se upotreba* `while` *petlje očekuje onda kada postoji specifičan uslov prekida ponavaljanja. Na primer, za program koji štampa prvih n prirodnih brojeva upotrebljava se* `for` *petlja*

```
for(i = 0; i < n; i++) {
    cout << i << " ";
}
```

dok se za program koji sabira brojeve koji se unose sa standardnog ulaza sve dok se ne unese broj 0 upotrebljava `while` *petlja*

```
while(broj != 0) {
cin >> broj;
    suma += broj;
}
```

Ove odluke utču na bolju čitljivost koda.

Primer 1.25. *Programeri biraju da li će koristiti niz naredbi* `if-else` *ili naredbu* `switch/case`*. Kada postoji mnogo različitih mogućih vrednosti neke promenljive, naredba* `switch/case` *se koristi radi bolje organizacije i čitljivosti koda. S druge strane, ako imamo neki binarni uslov, prirodno je koristiti naredbu* `if-else`*. Na primer, za provera da li je broj paran koristi se naredba* `if-else`*:*

```
if (broj % 2 == 0) {
    cout << "Broj je paran!" << endl;
} else {
    cout << "Broj je neparan!" << endl;
}
```

S druge strane, za štampanje ostatka pri deljenju sa brojem 5, preglednije je koristiti naredbu `switch/case`

```
switch (broj % 5) {
    case 1: cout << "Jedan" << endl; break;
    case 2: cout << "Dva" << endl; break;
    case 3: cout << "Tri" << endl; break;
    case 4: cout << "Cetiri" << endl; break;
    default: cout << "Broj je deljiv sa 5" << endl;
}
```

Primer 1.26. *Programeri koriste funkcije da bi smanjili dupliranje koda i olakšali održavanje. Funkcije omogućavaju lakšu organizaciju koda, ponovnu upotrebu i modularnost.*

Primer 1.27. *Programeri nekada koriste rekurziju (funkcija koja poziva samu sebe) umesto iteracije. Rekurzija je često prirodniji izbor za probleme koji se mogu podeliti na manje podprobleme, kao što su pretraga stabala ili rešavanje problema poput rekurzivno definisanih matematičkih funkcija. Iteracija se bira kada se traži efikasnije i manje memorijski zahtevno rešenje, jer rekurzija može izazvati prekomernu upotrebu memorije. U nastavku je dat kôd rekurzivnog i iterativnog računanja faktorijela prirodnog broja.*

```
int faktorijelRekurzivno(int n) {
    if (n == 0) {
        return 1;  // Bazni slučaj: faktorijel od 0 je 1
    } else {
        return n * faktorijelRekurzivno(n - 1);  // Rekurzivni poziv
    }
}
```

```
int faktorijelIterativno(int n) {
    int rezultat = 1;
    for (int i = 1; i <= n; i++) {
        rezultat *= i;  // Množi rezultat sa svakim sledećim brojem
    }
    return rezultat;
}
```

Pragmatika programskih jezika može se razmatrati i u širem kontekstu u kojem obuhvata najpre izbor jezika i tehnologije koji će se koristiti za rešavanje nekog konkretnog problema, a zatim i razumevanje koncepata dizajna programskih jezika. Pragmatika jezika se, između ostalog, bavi i sledećim pitanjima dizajna programskih jezika: izbor i karakteristike tipova podataka koje jezik obezbeđuje kao i načinima da se kreiraju kompleksni tipovi podataka, načini na koji se može ostvariti kontrola toka izvršavanja u programima, upotreba i karakteristike potprograma, modularnost i načini omogućavanja podele koda i modularnog programiranja, pristup i upravljanje memorijom i slično.

Pitanja i zadaci za vežbu

Pitanje 1.27. *Šta definiše leksika, a šta sintaksa programskih jezika?*

Pitanje 1.28. *Navesti primer leksički ispravnog, ali sintaksički neispravnog dela programa.*

Pitanje 1.29. *Šta definiše semantika programskih jezika?*

Pitanje 1.30. *Navesti primer sintaksički ispravnog, ali semantički neispravnog dela programa.*

Pitanje 1.31. *Koja je razlika između statičke i dinamičke semantike?*

Pitanje 1.32. *Čime se bavi pragmatika programskih jezika? Navesti primere.*

1.3 Jezički procesori

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

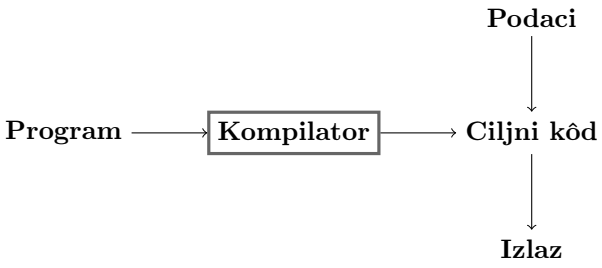
U zavisnosti od toga da li se ceo program analizira i transformiše u mašinski kôd pre nego što može da se izvrši, ili se analiza i izvršavanje programa obavljaju naizmenično deo po deo programa (na primer, naredba po naredba), jezički procesori se dele u dve grupe: *kompilatore* i *interpretatore*. Pored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

Kompilatori. Kompilatori (ili kompajleri) su programski prevodioci kod kojih su faza prevođenja i faza izvršavanja programa potpuno razdvojene. Nakon analize *izvornog koda* programa višeg programskog jezika, kompilatori generišu *izvršivi (mašinski) kôd* i dodatno ga optimizuju, a zatim čuvaju u vidu *izvršivih (binarnih) datoteka*. Jednom sačuvani mašinski kôd moguće je izvršavati neograničen broj puta, bez potrebe za ponovnim prevođenjem (slika 1.1). Krajnjim korisnicima nije neophodno dostavljati izvorni kôd programa na višem programskom jeziku, već je dovoljno distribuirati izvršivi mašinski kôd[4]. Jedan od problema u radu sa kompilatorima je da se prevođenjem gubi svaka veza između izvornog i izvršivog koda programa. Svaka (i najmanja) izmena u izvornom kodu programa zahteva ponovno prevođenje programa ili njegovih delova pri čemu samo prevođenje zahteva značajne, pre svega vremenske resurse. S druge strane, kompilirani programi su obično veoma efikasni, značajno efikasniji nego kada se program izvršava ne neki od narednih načina.

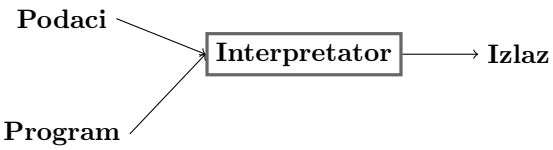
Interpretatori. Interpretatori su programski prevodioci kod kojih su faza prevođenja i faza izvršavanja programa isprepletane (slika 1.2). Interpretatori analiziraju deo po deo (najčešće naredbu po naredbu) izvornog koda programa i odmah nakon analize vrše i njegovo izvršavanje. Rezultat prevođenja se ne smešta u izvršive datoteke, već je prilikom svakog izvršavanja neophodno iznova vršiti analizu izvornog koda. Zbog ovoga, programi koji se interpretiraju se obično izvršavaju znatno sporije nego u slučaju kompilacije. S druge strane, razvojni ciklus programa je često kraći ukoliko se koriste interpretatori. Naime, prilikom malih izmena programa nije potrebno iznova vršiti analizu celokupnog koda.

Kombinacija kompilacije i interpretacije. Danas se često primenjuje i tehnika kombinovanja kompilatora i interpretatora. Naime, kôd sa višeg programskog jezika se prvo kompilira u neki precizno definisan

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



Slika 1.1: Kompilacija i izvršavanje



Slika 1.2: Interpretacija i izvršavanje

međujezik niskog nivoa (ovo je obično jezik neke apstraktne virtuelne mašine), a zatim se vrši interpretacija ili kompilacija u vreme izvršavanja ovog međujezika i njegovo izvršavanje na konkretnom računaru.

Neke platforme i programski jezici zasnovani su na ideji da se programi kompiliraju u specifičan „polupre-vedeni kôd“ (često se naziva bajtkod, engl. *bytecode*) a zatim se taj kôd prilikom izvršavanja interpretira ili kompilira na neki specifičan način. Bajtkod se može shvatiti kao asemblerski tj. mašinski jezik neke *virtuelne mašine* koji je mnogo nižeg nivoa nego originalni program na višem programskom jeziku, ali koji za razliku od realnog asemblera ne pokriva detalje konkretne arhitekture na kojoj će se program izvršavati. Veoma popularna je i takozvana JIT kompilacija (engl. *just in time compilation*), koja podrazumeva da se bajtkod izvršava tako što se često izvršavane naredbe programa prevode u mašinske instrukcije za ciljnu mašinu, koje se onda čuvaju i direktno izvršavaju (umesto da se interpretiraju). Moguća je i takozvana AOT kompilacija (engl. *ahead of time compilation*) koja podrazumeva da se pre svog izvršavanja ceo bajtkod prevede na mašinski jezik ciljnog računara. I JIT i AOT kompilacijom se dobija mnogo brže izvršavanje programa nego kod klasičnog interpretiranja bajtkoda, a zadržavaju se prednosti koje prevođenje na bajtkod donosi. To je pre svega jednostavna prenosivost programa prevedenih na bajtkod na različite platforme — za svaku novu platformu potrebno je izgraditi samo interpreter, JIT ili AOT kompilator koji obrađuju bajtkod relativno blizak mašinskom jeziku, a to je mnogo jednostavnije nego razviti kompilator za polazni izvorni jezik visokog nivoa (prevođenje do nivoa bajtkoda može da bude izrazito složen proces, koji uključuje kako različite analize, tako i različite napredne optimizacije izvornog programa).

Među najznačajnijim jezicima koji danas koriste bajtkod su Java i C#. Java programi mogu da se kompiliraju na takozvani JAVA bajtkod, koji se onda može interpretirati ili JIT kompilirati na bilo kakvom računaru (koji ima raspoloživu takozvanu Java virtualnu mašinu — JVM). U toku je i aktivni razvoj AOT kompilatora za Javu. Postoje prevodioci i za druge programske jezike koji koriste ovaj pristup i generišu Java bajtkod (na primer, programski jezici Scala i Kotlin). C# je deo Microsoft-ove .NET platforme i on se, kao i Visual Basic i F#, prevodi na bajtkod platforme .NET. Iako dominantno vezana za operativni sistem Windows, platforma .NET je podržana i na drugim operativnim sistemima (na primer, Mono je implementacija otvorenog koda platforme .NET koja može da se koristi i na Linux sistemima).

Naglasimo da programski jezik sâm ne određuje da li će programi na njemu biti prevođeni na jedan, drugi ili neki treći način. Zaista, za mnoge jezike postoji raspoloživo više raznorodnih sistema za prevođenje i izvršavanje. Intepretator se često koristi u fazi razvoja programa da bi omogućio interakciju korisnika sa programom, dok se u fazi eksploatacije kompletno razvijenog i istestiranog programa koristi kompilator koji proizvodi program sa efikasnim izvršavanjem.

Kako bi se izvršila standardizacija i olakšala izgradnja jezičkih prevodioca potrebno je precizno definisati šta su ispravne sintaksičke konstrukcije programskog jezika. Opisi na govornom, prirodnom jeziku, iako mogući, obično nisu dovoljno precizni i potrebno je korišćenje preciznijih formalizama. Ovi formalizmi se nazivaju *metajezici* dok se jezik koji se opisuje korišćenjem metajezika naziva *objektni jezik*. Metajezici obično rezervišu neke simbole kao specijalne, obično za svoj zapis koriste samo ASCII karaktere i imaju svoju posebnu sintaksu pogodnu za jednostavnu obradu na računaru. U okviru ove glave biće ukratko opisano i nekoliko metajezika.

1.3.1 Struktura kompilatora

Prevođenje (kompilacija) programskog jezika je transformisanje teksta programa na jednom računarskom jeziku u tekst programa na drugom jeziku. Obično je polazni jezik programski jezik visokog nivoa, a ciljni jezik je asembler ili mašinski jezik, ili jezik neke „virtualne mašine“. *Kompilatori* prevode čitav program na jeziku višeg nivoa u program na mašinskom ili nekom drugom ciljnom jeziku. Ukoliko je ciljni jezik mašinski jezik, onda, očigledno, kompilacija mora zavisiti od mašine na kojoj će se program izvršavati. Kako bi bilo moguće prevođenje programa u odgovarajuće programe na mašinskom jeziku nekog konkretnog računara, neophodno je precizno definisati sintaksu i semantiku programskih jezika.

Današnji kompilatori obično imaju tri ključne komponente:

Prednji sloj (eng. *front-end*) čita program zapisan na višem programskom jeziku, obrađuje ga i pohranjuje u obliku interne reprezentacije tj. međukoda. Sastoji se od sledećih komponenti (kojima u nekim slučajevima prethodi pretprocesor):

- Leksički analizator;
- Sintaksički analizator;
- Semantički analizator;
- Generator međukoda.

Srednji sloj (eng. *middle-end*) optimizije međukod i priprema ga za prevođenje na ciljni jezik. Čini ga jedna komponenta:

- Optimizator međukoda.

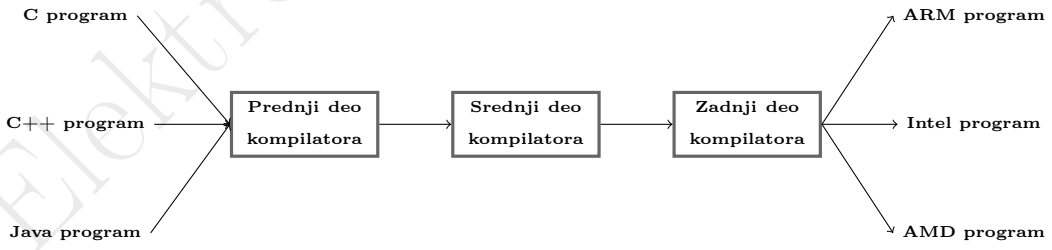
Iako se srednji sloj bavi samo jednim poslom, tj. optimizacijom međukoda, ovaj sloj je najkompleksniji i najvažniji sloj modernih kompilatora jer od njega najviše zavisi efikasnost izvršivog koda.

Zadnji sloj (eng. *back-end*) prevodi internu reprezentaciju (međukod) u ciljni jezik (često, ali ne nužno mašinski jezik). Čini ga naredna komponenta:

- Generator i optimizator ciljnog koda.

U preocesu finalnog prevođenja u ciljni jezik, generator i optimizator često rade naizmenično i isprepletano.

Razlaganje kompilatora na navedene komponente omogućava kombinovanje različitih prednjih slojeva (koji omogućavaju prevođenje različitih programskih jezika na međukod) i različitih zadnjih slojeva (koji su prilagođeni različitim ciljnim arhitekturama) sa jednim srednjim slojem, koji je ujedno i najkompleksniji sloj kompilatora. To je prikazano na slici 1.3.



Slika 1.3: Različiti slojevi kompilatora

U nastavku su ukratko opisane sve navedne faze kompilacije.

1.3.2 Leksička analiza

Leksička analiza je proces izdvajanja *tokena*, osnovnih jezičkih elemenata, iz niza ulaznih karaktera (na primer, karaktera koji čine program). U analogiji sa prirodnim jezikom, leksička analiza bi odgovarala podeli rečenice na reči i određivanju vrste svake reči. Deo kompilacije koji se bavi leksičkom analizom naziva se *leksički analizator* ili *lekser*.

Primer 1.28. *Razmotrimo naredni kôd napisan u programskom jeziku C++:*

```
if (starost >= 18)
    dopuna = 0;
else
    dopuna = 18 - starost;
```

Navedeni kôd je, zapravo, samo niz karaktera (`\n` *označava znak za novi red, a* `\t` *označava tabulator):*

```
if (starost >= 18)\n\tdopnua=0;\nelse\n\tdopuna = 18 - starost;
```

Zadatak leksera je da razloži ovaj niz na tokene kao što su identifikator, celobrojna vrednost, matematički operator, itd.

Token je sintaksička klasa, kategorija, kao što su u prirodnom jeziku kategorije, na primer, imenice, glagoli ili prilozi. Leksema je konkretan primerak, konkretna instanca jednog tokena.

Primer 1.29. *U prirodnim jezicima, jedna imenica je* pesma*, a jedan glagol je* pevati*. Slično, u programskim jezicima, za token* `IDENTIFIER`*, primeri leksema bi mogli da budu* `starost` *i* `dopuna` *iz primera 1.28, a za token* `OPERATOR` *primer lekseme mogao bi da bude* `-`*.*

Pored izdvajanja tokena, odnosno *tokenizacije*, leksički analizator može imati i druge zadatke, kao što je, na primer, eliminacija komentara (ako nema pretprocesora). Tokom leksičke analize u specijalnu tabelu koja se naziva *tabela simbola*, upisuju se prepoznati identifikatori i pridružuju im se određene relevantne informacije (na primer, vrsta i kolona u kodu gde je taj identifikator pronađen). Ova tabela dopunjuje se tokom narednih faza kompilacije (na primer, informacijama o tipovima).

Leksička analiza može da otkrije neke (jednostavne) vrste grešaka u kodu.

Primer 1.30. *U programskom jeziku C, prilikom kompilacije narednog koda, biće prijavljenje greške koje su date u komentarima. Ove greške prijavljuje leksički analizator.*

```
int a = 09;  /* error: invalid digit "9" in octal constant */
printf("Hi); /* error: missing terminating " character */
```

Leksički analizatori obično prosleđuju spisak izdvojenih leksema (i tokena kojima pripadaju) drugom programu koji nastavlja analizu teksta programa.

Regularni izrazi kao metajezik za opis leksike. Token, kao skup svih mogućih leksema, opisuje se formalno pogodnim obrascima koji mogu da uključuju cifre, slova, specijalne simbole i slično. Za te obrasce za opisivanje tokena (tj. za opis leksike programskog jezika) kao metajezik se obično koristi formalizam *regularnih izraza (engl. regular expressions)*.

Osnovne konstrukcije koje se koriste prilikom zapisa regularnih izraza su:

karakterske klase: navode se između `[` i `]` i označavaju jedan od navedenih karaktera.

alternacija: navodi se sa `|` i označava alternativne mogućnosti.

opciono pojavljivanje: navodi se sa `?`.

ponavljanje: navodi se sa `*` i označava da se nešto javlja nula ili više puta.

pozitivno ponavljanje: navodi se sa `+` i označava da se nešto javlja jedan ili više puta.

Primer 1.31. *Primeri osnovnih konstrukcija:*

- *primer karakterske klase:* `[0-9]` *označava cifru.*
- *primer alternacije:* `a|b` *označava slovo* `a` *ili slovo* `b`*;*
- *primer opcionog pojavljivanja:* `a?` *označava da slovo* `a` *može, a ne mora da se javi;*
- *prime ponavljanja:* `a*` *označava niz od nula ili više slova* `a`*;*
- *primer pozitivnog ponavljanja:* `[0-9]+` *označava neprazan niz cifara.*

Primer 1.32. *Razmotrimo identifikatore u programskom jeziku C++. Govornim jezikom, identifikatore je moguće opisati kao*

„neprazne niske koje se sastoje od slova, cifara i podvlaka, pri čemu ne mogu da počnu cifrom“.

Ovo znači da je prvi karakter identifikatora slovo ([a-zA-Z]) ili podvlaka (_), za čim sledi nula ili više (*) slova ([a-zA-Z]), cifara ([0-9]) ili podvlaka (_). Na osnovu ovoga, moguće je napisati regularni izraz kojim se opisuju identifikatori:

```
([a-zA-Z]|_)([a-zA-Z]|[0-9]|_)*
```

Ovaj izraz je moguće zapisati jednostavnije kao:

```
[a-zA-Z_][a-zA-Z_0-9]*
```

Algoritam za izdvajanje leksema iz ulaznog teksta zasniva se na takozvanim konačnim automatima. Postoje programi koji na osnovu opisa tokena u vidu regularnih izraza generišu leksere na izabranom programskom jeziku, na primer, na jeziku C. Primer takvog programa je `lex`.

Formalizam regularnih izraza je obično dovoljan da se opišu leksički elementi programskog jezika (na primer, skup svih identifikatora, skup brojevnih literala, i slično). Međutim nije moguće konstruisati regularne izraze kojim bi se opisale neke konstrukcije koje se javljaju u programskim jezicima. Na primer, nije moguće napisati regularni izraz kojim bi se opisali svi ispravni aritmetički izrazi, tj. skup $\{a,\ a+a,\ a*a,\ a+a*a,\ a*(a+a),\ \ldots\}$.

1.3.3 Sintaksička analiza

Sintaksička analiza, poznata i kao *parsiranje*, je proces organizovanja leksema izdvojenih u fazi leksičke analize u ispravnu jezičku konstrukciju. U programskim jezicima ispravne jezičke konstrukcije mogu da uključuju dodele, petlje, uslovne naredbe itd. U analogiji sa prirodnim jezikom, sintaksička analiza odgovara proveri da li su reči u rečenici složene u skladu sa gramatičkim pravilima jezika, kao i određivanju gramatičke strukture rečenice (određivanje subjekta, predikata, itd).

Primer 1.33. *Rečenica* Jelena ide u školu. *je sintaskički ispravna, dok rečenica* Jelena u ide školu. *nije ispravna. Primetimo da obe rečenice sadrže identične reči. Slično, kôd*

```
dopuna = 18 - starost;
```

jeste ispravan, dok naredni kôd

```
18 - starost = dopuna;
```

nije ispravan. Primetimo da i u ovom slučaju, oba primera sadrže identične lekseme.

Rezultat sintaksičke analize za ispravnu ulaznu jezičku konstrukciju je *sintaksičko stablo* (ili *stablo parsiranja*). Programi koji vrše parsiranje zovu se *sintaksički analizatori* ili *parseri.*

Sintaksička analiza može da otkrije raznovrsne greške u kodu.

Primer 1.34. *U programskom jeziku C++, prilikom kompilacije narednog koda, biće prijavljenje greške koje su date u komentarima, pri čemu će umesto* ln1 *i* ln2 *biti istaknuti odgovarajući brojevi linija u kojima se ovaj kôd nalazi. Ove greške prijavljuje sintaksički analizator.*

```
18 - starost = dopuna; /* error: lvalue required as left operand of assignment
                                ln1 |     18 - starost = dopuna;
                                    |     ~~~^~~~~~~~~~
                       */

if starost >= 18  /* error: expected ‘(’ before ‘starost’
                                ln2 |     if starost >= 18
                                    |        ^~~~~~~
                                    |        (
                       */
```

Načini opisa sintakse programskih jezika. Za opisivanje sintakse jezika obično se koriste kontekstno-slobodne gramatike (jer izražajna snaga regularnih izraza nije dovoljno velika za to) i drugi odgovarajući metajezici: *BNF (Bakus-Naurova forma)*, *EBNF (proširena Bakus-Naurova forma)* i *sintaksički dijagrami*. BNF je pogodna notacija za zapis kontekstno-slobodnih gramatika, EBNF proširuje BNF operacijama regularnih izraza čime se dobija pogodniji zapis, dok sintaksički dijagrami predstavljaju slikovni metajezik za predstavljanje sintakse. Dok je BNF veoma jednostavan metajezik, precizna definicija EBNF zahteva više truda i ona je data kroz ISO 14977 standard.

Kontekstno-slobodne gramatike. Sintaksa jezika se obično opisuje gramatikama. U slučaju prirodnih jezika, gramatički opisi se obično zadaju neformalno, koristeći govorni jezik kao metajezik u okviru kojega se opisuju ispravne konstrukcije, dok se u slučaju programskih jezika, koriste znatno precizniji i formalniji opisi. Za opis sintaksičkih konstrukcija programskih jezika koriste se uglavnom kontekstno-slobodne gramatike (engl. *context free grammars*).

Kontekstno-slobodne gramatike su izražajniji formalizam od regularnih izraza. Sve što je moguće opisati regularnim izrazima, moguće je opisati i kontekstno-slobodnim gramatikama, tako da je kontekstno-slobodne gramatike moguće koristiti i za opis leksičkih konstrukcija jezika (ali regularni izrazi obično daju koncizniji opis).

Kontekstno-slobodne gramatike su određene skupom pravila. Sa leve strane pravila nalaze se takozvani pomoćni simboli (neterminali), dok se sa desne strane nalaze niske u kojima mogu da se javljaju bilo pomoćni simboli bilo takozvani završni simboli (terminali). Svakom pomoćnom simbolu pridružena je neka sintaksička kategorija. Jedan od pomoćnih simbola se smatra istaknutim, naziva se početnim simbolom (ili aksiomom). Niska je opisana gramatikom ako ju je moguće dobiti krenuvši od početnog simbola, zamenjujući u svakom koraku pomoćne simbole desnim stranama pravila.

Primer 1.35. *Neka je gramatika zadata sledećim pravilima:*

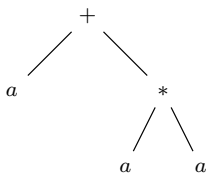
$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid a \end{aligned}$$

Neterminal E odgovara izrazima, neterminal T sabircima (termima), a neterminal F činiocima (faktorima).

I ova gramatika opisuje ispravne aritmetičke izraze. Na primer, niska $a + a * a$ se može izvesti na sledeći način:

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T * F \Rightarrow \\ a + F * F \Rightarrow a + a * F \Rightarrow a + a * a.$$

Ovom izvođenju odgovara naredno stablo izvođenja.



Kontekstno-slobodne gramatike čine samo jednu specijalnu vrstu formalnih gramatika. U kontekstno-slobodnim gramatikama, sa leve strane pravila uvek se nalazi tačno jedan neterminalni simbol a sa desne strane pravila može se, u opštem slučaju, nalaziti proizvoljan niz terminalnih i neterminalnih simbola.

BNF. BNF je metajezik pogodan za zapis pravila kontekstno-slobodnih gramatika. U BNF notaciji, sintaksa objektnog jezika se opisuje pomoću konačnog skupa *metalingvističkih formula (MLF)* koje direktno odgovaraju pravilima kontekstno-slobodnih gramatika.

Svaka metalingvistička formula se sastoji iz leve i desne strane razdvojene specijalnim, takozvanim univerzalnim metasimbolom (simbolom metajezika koji se koristi u svim MLF) `::=` koji se čita „po definiciji je“, tj. MLF je oblika `A ::= a`, gde je `A` metalingvistička promenljiva, a `a` metalingvistički izraz. Metalingvističke promenljive su fraze prirodnog jezika u uglastim zagradama (`<`, `>`), i one predstavljaju pojmove, tj. sintaksičke kategorije objektnog jezika. Ove promenljive odgovaraju pomoćnim (neterminalnim) simbolima formalnih gramatika. U

programskom jeziku, sintaksičke kategorije su, na primer, `<program>`, `<ceo broj>` i `<identifikator>`. Metalingvističke promenljive ne pripadaju objektnom jeziku.

S druge strane, metalingvističke konstante su simboli objektnog jezika. To su, na primer, `0`, `1`, `2`, `+`, `-`, ali i rezervisane reči programskog jezika, na primer, `if`, `then`, `begin`, `for`, itd. Dakle, uglaste zagrade razlikuju neterminalne simbole tj. imena sintaksičkih kategorija od terminalnih simbola objektnog jezika koji se navode tačno onako kakvi su u objektnom jeziku.

Metalingvistički izrazi se grade od metalingvističkih promenljivih i metalingvističkih konstanti primenom operacija konkatenacije i alternacije (`|`).

Primer 1.36. *Jezik celih brojeva u dekadnom brojnom sistemu može se opisati sledećim skupom MLF:*

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

Primer 1.37. *Gramatika aritmetičkih izraza može u BNF da se zapiše kao:*

```
<izraz>  ::= <izraz> + <term> | <term>
<term>   ::= <term> * <faktor> | <faktor>
<faktor> ::= ( <izraz> ) | <ceo broj>
```

EBNF. EBNF (od „Extended Backus-Naur Form“ tj. „Proširena Bakus-Naurova forma“) je formalizam (opisan standardom ISO 14977) koji je iste izražajnosti kao BNF, ali uvodi skraćene zapise koji olakšavaju zapis gramatike i čine je čitljivijom. EBNF proširuje BNF nekim elementima regularnih izraza:

- Završni simboli objektnog jezika se navode pod navodnicima kako bi se svaki karakter, uključujući i one koji se koriste kao metasimboli u okviru EBNF, mogli koristiti kao simboli objektnog jezika.
- Pravougaone zagrade `[` i `]` ukazuju na opciono pojavljivanje.
- Vitičaste zagrade `{` i `}` ukazuju na ponavljanje.
- Svako pravilo se završava eksplicitnom oznakom kraja pravila.
- Obične male zagrade se koriste za grupisanje.

Izražajnost i metajezika BNF i metajezika EBNF jednaka je izražajnosti kontekstno-slobodnih gramatika, tj. sva tri ova formalizma mogu da opišu isti skup jezika. EBNF nudi koncizniji zapis u odnosu na BNF.

Primer 1.38. *Korišćenjem EBNF identifikatori programskog jezika C se mogu definisati sa:*

```
<identifikator> ::= <slovo ili _> { <slovo ili _> | <cifra> }
<slovo ili _>   ::= "a"|...|"z"|"A"|...|"Z"|"_"
<cifra>         ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Primer 1.39. *Korišćenjem EBNF, jezik celih brojeva u dekadnom brojnom sistemu može se opisati sledećim skupom MLF:*

```
<ceo broj> ::= ["+"|"-"]<cifra>{<cifra>}
<cifra>    ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Primer 1.40. *Gramatika aritmetičkih izraza može u EBNF da se zapiše kao:*

```
<izraz>  ::= <term> {"+" <term>}
<term>   ::= <faktor> {"*" <faktor>}
<faktor> ::= "(" <izraz> ")" | <ceo broj>
```

Primer 1.41. *Naredba grananja* `if` *sa opcionim pojavljivanjem* `else` *grane može se opisati sa:*

```
<if_naredba>  ::= if "(" <bulovski_izraz> ")"
                     <niz_naredbi>
                  [ else
                     <niz_naredbi> ]
<niz_naredbi> ::= "{" <naredba> ";" { <naredba> ";" } "}"
```

Sintaksička analiza na osnovu zadate sintakse jezika zasniva se na takozvanim potisnim automatima. Postoje programi koji na osnovu opisa sintakse jezika (na primer, u vidu EBNF-a) generišu parsere na izabranom programskom jeziku, na primer, na jeziku C. Primer takvog programa je program `Yacc` (skraćeno od *Yet another compiler compiler*).

1.3.4 Semantička analiza

Semantička analiza je proces u kojem se proveravaju semantički uslovi i primenjuju pravila koja nije pogodno opisati sintaksičkim pravilima ni primenjivati u fazi sintaksičke analize. Semantička analiza vrši se nad sintaksičkim stablom izgrađenim tokom faze sintaksičke analize i neznatno ga modifikuje. Semantička analiza obično uključuje proveru tipova, implicitne konverzije, kao i provere koje se odnose na vidljivost promenljivih, to jest na njihov domen (jer kontekstno-slobodne gramatike nisu dovoljno izražajne da opišu pravila domena).

Semantička analiza može da otkrije raznovrsne greške u kodu. Takođe, kao rezultat semantičke analize mogu biti prijavljena i upozorenja. Upozorenja u kodu treba ozbiljno razmotriti i eliminisati.

Primer 1.42. *U programskom jeziku C++, prilikom kompilacije narednog koda, biće prijavljenja greška i upozorenje koji su dati u komentarima, pri čemu će umesto* ln1 *i* ln2 *biti istaknuti odgovarajući brojevi linija u kojima se ovaj kôd nalazi. Ova greška, odnosno upozorenje, rezultat su semantičke analize koda.*

```
int starost, starost;  /* error: redeclaration of ‘int starost’
                             ln1 |     int starost, starost;
                                 |                    ^~~~~~~
                                  note: ‘int starost’ previously declared here
                             ln1 |     int starost, starost;
                                 |         ^~~~~~~
                          */

if (starost = 18)      /* warning: suggest parentheses around assignment used
                             as truth value [-Wparentheses]
                             ln2 |  if (starost = 18)
                                 |      ~~~~~~~~^~~~
                          */
```

Načini opisa semantike programskih jezika Dok većina savremenih jezika ima precizno i formalno definisanu sintaksu, formalna definicija semantike postoji samo za neke programske jezike. U ostalim slučajevima, semantika programskog jezika se opisuje neformalno, opisima zadatim korišćenjem prirodnog jezika. Čest je slučaj da neki aspekti semantike ostaju nedefinisani standardom jezika i prepušta se implementacijama kompilatora da samostalno odrede potpunu semantiku.

Primer 1.43. *Semantika koda iz primera 1.28 se neformalno može opisati na sledeći način:*

> *Ako je vrednost promenljive* starost *veća ili jednaka 18, onda vrednost promenljive* dopuna *treba da postane jednaka nuli. U suprotnom, izračunaj vrednost promenljive* dopuna *oduzimanjem od broja 18 vrednosti promenljive* starost.

Ovo je primer semantike konkretnog programa. Ova semantika je opisana prirodnim jezikom na osnovu opisa semantike samog jezika, koja definiše značenje za naredbu if*, za naredbu dodele i za operator* -.

Formalno, semantika programskih jezika se zadaje na neki od naredna tri načina:

denotaciona sematika — programima se pridružuju matematički objekti (na primer funkcije koje preslikavaju ulaz u izlaz).

operaciona semantika — zadaju se korak-po-korak objašnjenja kako će se naredbe programa izvršavati na stvarnoj ili apstraktnoj mašini.

aksiomatska semantika — opisuje se efekat programa na tvrđenja (logičke formule) koja opisuju stanje programa. Najpoznatiji primer aksiomatske semantike je *Horova logika*.

Primer 1.44. *Semantika UR mašina definiše značenje programa operaciono, dejstvom svake instrukcije na stanje registara apstraktne UR mašine kao što je navedeno u tabeli 1.1.*

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

Tabela 1.1: Tabela URM instrukcija

1.3.5 Generisanje međukoda

Generisanje međukoda je proces u okviru kojeg se izlaz iz faze semantičke analize (u vidu označenog sintaksičkog stabla) prevodi u linearnu reprezentaciju, nezavisnu od konkretnih mašina. Na primer, složeni izrazi mogu se svoditi u međukodu samo na pojedinačne operacije sa po dva argumenta, koje lako mogu da se prevedu na asembler. To svojstvo je očuvano i tokom optimizacije međukoda. Time je omogućeno da se (mnogi) izrazi u međukodu mogu čuvati kao nizovi jednostavnih četvorki koje čine operator, dva argumenta i rezultat.

Primer 1.45. *Za izraz* `obim = a + b + c` *može da se generiše naredni međukod*

```
tmp1 = a
tmp2 = b
tmp3 = tmp1 + tmp2
tmp4 = c
tmp5 = tmp3 + tmp4
obim = tmp5
```

Ovako generisan kôd sadrži nepotrebne pomoćne promenljive i neophodno ga je optimizovati.

Algoritam generisanja međukoda je definisan za sve sintaksno i semantički ispravne programe. Kako je ulaz u fazu generisanja međukoda program za koji je utvrđeno da je sintaksno i semantički ispravan, nakon ove faze nije moguće da se pojave grešake i/ili upozorenja koji bi se saopštili korisniku. To važi sve do faze linkovanja, kada ponovo može da dođe do greške.

1.3.6 Optimizacija međukoda

Optimizacija međukoda je proces u okviru kojeg se na generisani međukod primenjuju raznovrsne optimizacije u cilju dobijanja efikasnijeg i kvalitetnijeg ciljnog koda, a bez promene njegovog vidljivog ponašanja tj. uz čuvanje *semantike programa*. Te optimizacije mogu da uključuju eliminisanje koda koji se ne koristi, propagaciju konstanti, promenu poretka naredbi, transformaciju petlji i slično.

Optimizacije mogu da budu lokalne i da se odnose na delove programa koji imaju jednostavnu, sekvencijalnu formu. Komplikovanije su optimizacije koje se odnose na razgranate delove programa i petlje, a najkomplikovanije optimizacije koje uključuju zavisnosti između različitih funkcija.

Primer 1.46. *Optimizacija može da naredne komande na međujeziku*

```
x := x + 0  // može se eliminisati: mrtav kod (kod bez efekta)
x := x * 1  // može se eliminisati: mrtav kod (kod bez efekta)
x := x * 0  // može se uprostiti: vrednost izraza je 0
z := x + u  // moguća je primena propagacije vrednosti x
y := 2 * x  // može se eliminisati: mrtav kod (kod bez efekta)
y := 2 * z  // može se ubrzati: množenje brojem 2 je ekvivalentno
```

```
            // šiftovanju za jedno mesto u levo, pri čemu je
            // šiftovanje značajno efikasnija operacija od množenja
```

zameni narednim komandama (na osnovu analize koja je data kao komentar uz svaku naredbu):

```
x := 0
z := 0 + u;
y := z << 1
```

Ovako izmenjen kôd je bolji jer sadrži manji broj efikasnijih naredbi.

Primer 1.47. *Kôd iz primera 1.45 može se optimizovati na način opisan u komentarima.*

```
tmp1 = a
tmp2 = b
tmp3 = tmp1 + tmp2 // prenos vrednosti: umesto tmp1 može se koristiti a
                   // umesto tmp2 može se koristiti b
                   // na taj način tmp1 i tmp2 postaju mrtav kod koji se može eliminisati
tmp4 = c
tmp5 = tmp3 + tmp4 // prenos vrednosti: umesto tmp4 može se koristiti c
                   // na taj način tmp4 postaje mrtav kod koji se može eliminisati
obim = tmp5        // prenos vrednosti: umesto tmp5 može se koristiti gornji izraz
                   // na taj način tmp5 postaje mrtav kod koji se može eliminisati
```

Optimzovani kôd je

```
tmp3 = a + b
obim = tmp3 + c
```

Optimizacija međukoda može da se izvršava sa različitim ciljevima. Tri osnovna cilja su smanjenje (i) vremena izvšavanja, (ii) veličine izvršivog programa i (iii) energije koju program troši prilikom izvršavanja. U primerima 1.46 i 1.47 ostvarena su sva tri cilja. Međutim, često su ovi ciljevi međusobno suprotstavljeni i to tako što ostvarenje jednog cilja negativno utiče na bar jedan od druga dva cilja. Na primer, smanjenje vremena izvršavanja često povećava veličinu izvršivog programa.

Proces optimizacije mora da uzme u obzir željene karakteristike programa. Najčešće se optimizacije vrše sa ciljem poboljšanja vremenskih performansi programa, pri čemu se vodi računa da povećanje veličine izvršivog programa bude razumno. U kontekstu uređaja sa ugrađenim računarom koji imaju ograničene resurse, može da bude važnija veličina izvršivog programa pa se u tom slučaju sprovodi drugačiji skup optimizacija. U kontekstu uređaja koji imaju ograničene izvore energije (na primer, rade na baterije: mobilni telefoni, pametni satovi i slično) veoma je bitna energetska efikasnost programa. Takođe, pored željenih karaktestika izvršivog programa koji kompilator treba da generiše, proces optimizacije mora da uzme u obzir i ukupno vreme kompilacije jer se nekada ne isplati primenjivati najkompleksnije optimizacije.

U oblasti prevođenja programskih jezika, faza optimizacije međukoda je najčešći predmet inovacija i istraživanja. Prethodne faze se već dugo sprovode na suštinski iste načine.

1.3.7 Generisanje i optimizacija ciljnog koda

Generisanje i optimizacija ciljnog koda je proces prevođenja međukoda na ciljni jezik, često jezik prilagođen nekoj konkretnoj računarskoj arhitekturi, kao što su konkretni mašinski jezici. Taj proces uključuje baratanje resursima niskog nivoa (na primer, da li će neka promenljiva da bude čuvana u registrima ili u memoriji). U okviru optimizacije koda, nizovi instrukcija na ciljnom jeziku (na primer, mašinskih instrukcija) mogu biti zamenjeni jednostavnijim fragmentima koda ili njihov poredak može biti promenjen radi kvalitetnijeg korišćenja procesora i registara.

1.3.8 Ilustracija sprovođenja faza kompilacije

Sve navedene faze kompilacije mogu se ilustrovati narednim pojednostavljenim primerom.

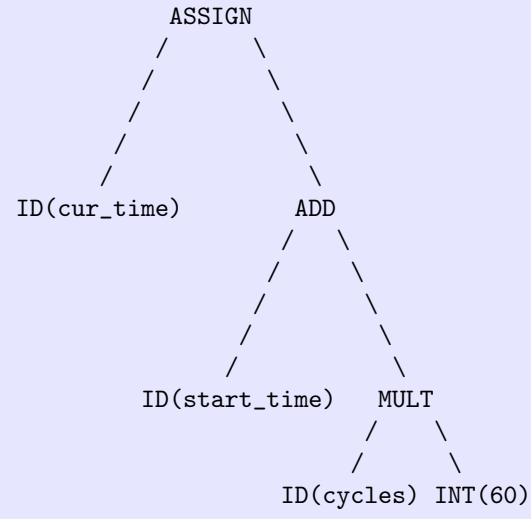
Primer 1.48. *Razmotrimo faze proveđenja za naredni izvorni kod koji obuhvata neredbu dodele i složeni aritmetički izraz za promenljive realnog tipa:*

```
cur_time = start_time + cycles * 60
```

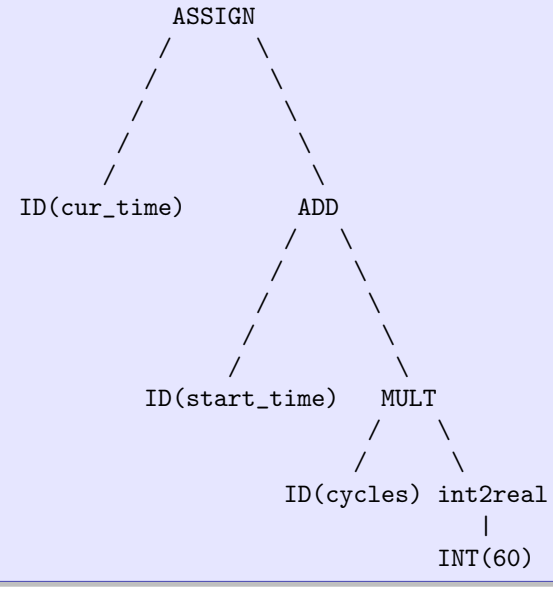
Leksička analiza: *izdvajanje reči i njihovih kategorija (tri identifikatora, dodela, sabiranje, množenje i celobrojna konstanta)*

```
ID(cur_time) ASSIGN ID(start_time) ADD ID(cycles) MULT INT(60)
```

Sintaksička analiza: *izgradnja sintaksičkog stabla*



Semantička analiza: *dodavanje čvora konverizije u sinstaksno stablo, usled nepoklapanja tipova*



Generisanje međukoda: *uvođenje privremenih promenljivih kako bi svaka naredba dodele imala najviše dva argumenta*

```
tmp1 = int2real(60)
tmp2 = cycles * tmp1
tmp3 = start_time + tmp2
cur_time = tmp3
```

Optimizacija međukoda:

Korak 1: zamena konverzije celobrojne promenljive 60 sa realnom promenljivom 60.0

```
tmp1 = 60.0
tmp2 = cycles * tmp1
tmp3 = start_time + tmp2
cur_time = tmp3
```

Korak 2: prenos vrednosti promenljive tmp1 *u izraz u kojem* tmp1 *učestvuje, prenos vrednosti promenljive* tmp3 *u izraz u kojem* tmp3 *učestvuje*

```
tmp1 = 60.0
tmp2 = cycles * 60.0
tmp3 = id2 + tmp2
id1 = id2 + tmp2
```

Korak 3: eliminacija mrtvog koda (nepotrebne promenljive tmp1 *i* tmp3*)*

```
tmp2 = cycles * 60.0
cur_time = start_time + tmp2
```

Generisanje koda na ciljnom jeziku: *odabir registara i instrukcija ciljne arhitekture*[5]

```
MOVF R2, cycles      # Prebaci u registar R2 vrednost sa memorijske lokacije cycles
MULF R2, 60.0        # Pomnozi vrednost registra R2 sa konstantom 60.0
                     # i rezultat smesti u R2
MOVF R1, start_time  # Prebaci u registar R1 vrednost sa memorijske
                     # lokacije start_time
ADDF R1, R2          # Saberi vrednosti registara R1 i R2, rezultat smesti u R1
MOVF cur_time, R1    # Prebaci na memorijsku lokaciju curr_time vrednost registra R1
```

Pitanja i zadaci za vežbu

Pitanje 1.33. *Šta je kompilator, a šta interpretator? Koje su osnovne prednosti, a koje su mane korišćenja kompilatora u odnosu na korišćenje interpretatora?*

Pitanje 1.34. *Ako se koristi kompilator, da li je za izvršavanje programa korisniku neophodno dostaviti izvorni kôd programa? Ako se koristi interpretator, da li je za izvršavanje programa korisniku neophodno dostaviti izvorni kôd programa?*

Pitanje 1.35. *U čemu je razlika između interpretatora i kompilatora?*

Pitanje 1.36. *Da li se, generalno, brže izvršava program koji se interpretira ili onaj koji je preveden kompilatorom? Zašto?*

Pitanje 1.37. *Na koji načine se interpretacija i kompilacija mogu kombinovati?*

Pitanje 1.38. *Prvim implementiranim kompilatorom se smatra onaj napravljen za jezik Fortran. Kada je on nastao?*

Pitanje 1.39. *Navesti osnovne slojeve kompilatora i objasniti šta se u kojem sloju dešava. Zašto postoje ovi slojevi i koja je njihova uloga?*

Pitanje 1.40. *Navesti osnovne faze u prevođenju programskih jezika.*

Pitanje 1.41. *Koja faza kompilacije najviše utiče na efikasnost izvršivog programa?*

Pitanje 1.42. *Šta je rezultat leksičke analize programa? Šta leksički analizator dobija kao svoj ulaz, a šta vraća kao rezultat svog rada?*

Pitanje 1.43. *Šta je to leksička analiza i koji se poslovi vrše tokom nje?*

[5] U pitanju je pojednostavljena fiktivna arhitektura i izmišljene instrukcije, ali po ugledu na postojeće arhitekture i instrukcije.

Pitanje 1.44. *Navesti niz leksema i niz odgovarajućih tokena koji su rezultat rada leksičkog analizatora nad izrazom* `a=3;`

Pitanje 1.45. *Kako se zovu programi koji vrše leksičku analizu?*

Pitanje 1.46. *U kom vidu se opisuju pravila koja se primenjuju u fazi leksičke analize?*

Pitanje 1.47. *U opisu regularnih jezika, kako se označava ponavljanje jednom ili više puta?*

Pitanje 1.48. *Navesti nekoliko primera reči koje pripadaju regularnom jeziku* `[a-z][0-9]+[0-9]`.

Pitanje 1.49. *Zapisati jezik studentskih naloga (na primer,* `mi10123`, `mr10124`, `aa10125`*) na studentskom serveru Alas u vidu regularnog izraza.*

Pitanje 1.50. *Navesti regularni izraz koji opisuje:*
(a) *niske u kojima se na početku mogu (a ne moraju) pojavljivati cifre (od 0 do 9), zatim sledi neprazan niz karaktera* `a`*, a zatim sledi neprazan niz cifara.*
(b) *niske u kojima se na početku može (a ne mora) pojaviti neka cifra (od 0 do 9), zatim sledi neprazan niz karaktera* `a`*, a zatim sledi niz (moguće prazan) karaktera* `b`.
(c) *neprazne konačne niske u kojima se na početku može (a ne mora) pojaviti karakter* `a`*, a zatim sledi neprazan niz cifara.*
(d) *konačne niske u kojima se na početku može (a ne mora) pojaviti karakter* `a` *ili karakter* `b`*, a zatim sledi niz cifara (moguće i prazan).*
(e) *neprazne konačne niske u kojima se pojavljuju ili samo karakteri* `a` *ili samo karakteri* `b`.
(f) *neprazne konačne niske u kojima se pojavljuju samo karakteri* `a` *i* `b`.

Pitanje 1.51. *Koja analiza se u kompilaciji programa vrši nakon leksičke analize?*

Pitanje 1.52. *Šta je to sintaksička analiza?*

Pitanje 1.53. *Šta je zadatak sintaksičke analize programa? Šta sintaksički analizator dobija kao svoj ulaz, a šta vraća kao rezultat svog rada?*

Pitanje 1.54. *Kako se zovu programi koji vrše sintaksičku analizu?*

Pitanje 1.55. *U kom obliku se opisuju pravila koja se primenjuju u fazi sintaksičke analize?*

Pitanje 1.56. *Da li su regularni jezici su obavezno i kontest-slobodni?*
Da li su kontekstno-slobodni jezici su obavezno i regularni?
Da li metajezik BNF ima istu izražajnost kao regularni jezici?
Da li metajezik BNF ima istu izražajnost kao kontekstno-slobodni jezici?
Da li metajezik BNF ima istu izražajnost kao metajezik EBNF?

Pitanje 1.57. *Od čega dolazi skraćenica EBNF?*

Pitanje 1.58. *Kako se u EBNF-u, zapisuje izraz koji se može ponavljati proizvoljan broj (nula ili više)?*

Pitanje 1.59. *Koji od sledećih metajezika je najizražajniji:*
(a) *kontekstno-slobodne gramatike;* (b) *BNF;* (c) *EBNF;* (d) *svi navedeni metajezici imaju istu izražajnost.*

Pitanje 1.60. *Zapisati jezik* $\{a^n b^n | n > 0\}$ *u vidu konteksno-slobodne gramatike i u vidu EBNF izraza.*

Pitanje 1.61. *Koji su osnovni ciljevi optimizacije međukoda?*

Pitanje 1.62. *Navesti primere optimizacija međukoda.*

Pitanje 1.63. *Koja je poslednja faza u kompiliaciji i za šta je ona odogovorna?*

Pitanje 1.64. *Ukoliko je raspoloživ kôd nekog programa na nekom višem programskom jeziku (na primer, na jeziku C), da li je moguće jednoznačno konstruisati odgovarajući mašinski kôd? Ukoliko je raspoloživ mašinski kôd nekog programa, da li je moguće jednoznačno konstruisati odgovarajući kôd na jeziku C?*