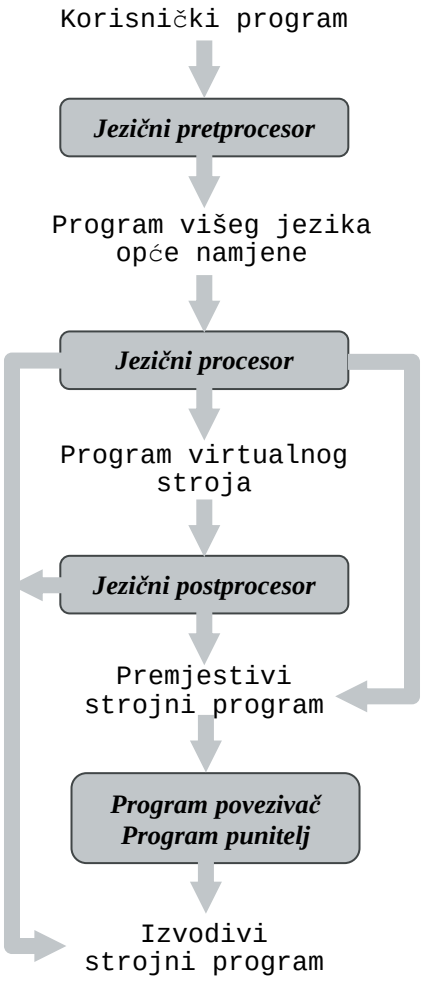
1 UVOD

Sve širi skup zadataka koji se rješavaju primjenom suvremenih digitalnih elektroničkih računala, kao što je na primjer stvaranje i pretraživanje raznih baza podataka, projektiranje i izgradnja raznorodnih sustava, te sve veći broj korisnika računala raznih struka kojima nije osnovni cilj izučavanje rada računala, nameću potrebu za razvojem različitih korisničkih programskih jezika prilagođenih različitim primjenama. Sve je veći broj korisnika računala koji ne poznaju ni osnovna načela programiranja u višim programskim jezicima kao što su C, Fortran ili Java. Za mnoga područja primjene predloženi su korisnički programski jezici posebne namjene koji omogućuju učinkovit i lak način učenja pravila jezika, bez potrebe poznavanja sklopovske i programske arhitekture računala. Danas vjerojatno nema korisnika računala koji se ne koristi nekim od programa pretraživača, bilo onog povezanog s lokalnom bazom podataka, pa sve do pretraživača podataka na globalnoj mreži Internet. Zadavanje upita programu pretraživaču omogućeno je primjenom *korisničkog jezika*. Upitom se definira odnos između ključnih riječi koje se pretražuju, odnosno upiti su naredbe koje se zadaju programu pretraživaču i one čine jednostavni korisnički program.

Nastoje se pojednostaviti pravila korisničkog jezika, tako da korisnik nema potrebe učiti ih unaprijed. Namjena korisničkog jezika je približiti jezik programiranja području primjene, osloboditi jezik programiranja zavisnosti o arhitekturi računala, omogućiti što lakše učenje pravila jezika, olakšati razvoj i razumijevanje programa, olakšati ispravljanje i pronalaženje pogrešaka u programu, olakšati održavanje i dokumentiranje programa, povećati prenosivost na računala različitih arhitektura i skratiti vrijeme rješavanja problema. Iako primjena korisničkih jezika donosi prednosti, njihova uporaba ima i nedostataka: produljuje se vrijeme potrebno za prevođenje korisničkog programa u izvodivi strojni program, strojni program je neučinkovit jer je generiran iz korisničkog programa koji je oslobođen svih detalja arhitekture računala, nemoguće je iskoristiti sve posebnosti građe računala na kojoj se izvodi program, te je potrebno izgraditi razvojnu okolinu koja omogućuje ispitivanje i nadzor izvođenja korisničkih programa.

Iako za korisnika jednostavan, zadan korisnički program prolazi mnoštvo složenih postupaka pripreme koji omogućuju njegovo izvođenje na računalu. Priprema korisničkog programa za izvođenje na računalima je složen višeslojni računalni proces postupnog prevođenja korisničkog programa u izvodivi strojni program. Na sreću, za većinu primjena priprema korisničkih programa za izvođenje je u potpunosti automatizirana. U samom računalu ugrađen je programski sustav koji je upravo namijenjen pripremi korisničkog programa za izvođenje.



Slika 1.1: Priprema korisničkog programa za izvođenje

Shematski prikaz višeslojnog procesa pripreme korisničkog programa za izvođenje prikazan je na slici 1.1. Pripremu korisničkog programa za izvođenje čini niz postupaka prevođenja programa iz jednog programskog jezika u drugi programski jezik. Tek prevođenjem *korisničkog* u *izvodivi strojni program* omogućeno je njegovo izvođenje na računalu. Izvodivi strojni program je niz nula i jedinica. Nizovi nula i jedinica su strojne naredbe procesora računala i one čine program koji je moguće izravno izvesti na zadanom računalu bez dodatne dorade.

Dva su osnova razloga neprevođenja korisničkog programa izravno u izvodivi strojni program: složenost postupka prevođenja korisničkog programa u izvodivi strojni program i građa funkcija koje se neposredno koriste u korisničkom programu. Na primjer, moguće je samo jednim upitom programu pretraživaču, koji je u stvari naredba korisničkog programa, pokrenuti izvođenje više stotina ili tisuća strojnih naredbi procesora računala.

Drugi razlog je građa funkcija koje se neposredno koriste u korisničkom programu. Naredbe korisničkog programa pokreću izvođenje različitih funkcija, koje su obično izgrađene primjenom jednog od programskih jezika opće namjene, na primjer C, Fortran ili Java. Ovisno o paradigmi koju koristi programski jezik, funkcije su potprogrami, objekti ili pomoćni programi. Budući da korisnički program koristi funkcije koje su ostvarene jednim od općih programskih jezika, te budući da su za te jezike izgrađeni jezični procesori, uobičajeno je da se korisnički program prvotno prevede u niz naredbi višeg programskog jezika opće namjene. U tako dobivenom programu moguće je izravno koristiti potrebne funkcije koje se naslovljavaju kao potprogrami, objekti, pomoćni programi, itd., dok uporaba jezičnog procesora omogućuje daljnju pripremu korisničkog programa za izvođenje.

Jezični procesor je središnji proces prevođenja korisničkog programa u izvodivi strojni program. Nazivi ostalih procesa prevođenja, pretprocesiranje i postprocesiranje, određuju se u odnosu na središnju ulogu jezičnog procesora. *Program višeg jezika opće namjene* je program zapisan jednim od viših programskih jezika kao što su C, Pascal, Fortran ili Java. Jezični procesori prevode program višeg jezika opće namjene izravno u izvodivi strojni program, ili u jedan od oblika koje je potrebno dodatno doraditi prije samog izvođenja na zadanom računalu: *program virtualnog stroja* ili *premjestivi strojni program*. Na primjer, jezični procesori viših programskih jezika Pascal i Java generiraju program virtualnih stogovnih strojeva. Java jezični procesor generira naredbe virtualnog stroja Java, a jezični procesor programskog jezika Pascal generira naredbe P-programa virtualnog stogovog stroja. Program virtualnog stroja je strojno nezavisni program, što znači da ga nije moguće izravno izvesti na nekom od računala, već ga je potrebno dodatno prevesti u izvodivi strojni program. Većina ostalih jezičnih procesora, osim onih najjednostavnijih koji izravno generiraju izvodivi strojni program, generira premjestivi strojni program. Naredbe premjestivog strojnog programa slične su naredbama izvodivog strojnog programa, samo što naredbe premjestivog strojnog programa nemaju do kraja izračunate numeričke vrijednosti memorijskih adresa. Memorijske adrese su u relativnom obliku i označavaju pomak od početne adrese memorije koja se dodijeli strojnom programu za vrijeme izvođenja. Za razliku od izvodivog strojnog programa koji se sprema i izvodi u unaprijed definiranom i određenom memorijskom prostoru, premjestivi strojni program moguće je uz jednostavnu dodatnu doradu adresa spremiti i izvesti u bilo kojem dijelu memorijskog prostora računala.

Budući da prethodi radu jezičnog procesora, prevođenje korisničkog jezika u jedan od viših programskih jezika naziva se pretprocesiranje. *Jezični pretprocesor* prevodi program zapisan jednim od korisničkih jezika posebne namjene u program zapisan jednim od viših programskih jezika opće namjene. Na primjer, SpecC jezik je proširenje višeg programskog jezika C opće namjene za potrebe projektiranja ugrađenih računalskih sustava, tj. SpecC je korisnički jezik posebne namjene. Program napisan u jeziku SpecC jezični pretprocesor prevodi u viši programski jezik C++ opće namjene, a onda se korištenjem jezičnog procesora C++ dobiveni program prevodi u strojni program za zadano ugrađeno računalo.

Prevođenje jezika virtualnog stroja u strojni jezik naziva se postprocesiranje. *Jezični postprocesor* prevodi strojno nezavisni program virtualnog stroja u strojni program. Na primjer, naredbe Java virtualnog stroja su mnemoničkog oblika i za svoje izvođenje koriste virtualni stogovni stroj. Naredbe računaju izraze tako da prethodno spreme operande na stog. Nakon što izračunaju tražene vrijednosti, rezultat računanja spremi se na stog. Želi li se Java virtualni program izvesti na zadanom računalu, potrebno je imati postprocesor, koji se naziva Java virtualni stroj, koji interpretira naredbe Java virtualnog programa i prevodi ih u naredbe procesora računala, odnosno u izvodivi strojni program. Postprocesiranje primjenom Java virtualnog stroja omogućuje izgradnju jedinstvenog Java jezičnog procesora za sve arhitekture računala. Opisano svojstvo omogućuje razmjenu programa generiranih primjenom jezičnog procesora Java putem globalne mreže Internet, i to između računala različitih arhitektura.

Osnovni zadatak *programa povezivača* i *programa punitelja* je prevođenje premjestivog strojnog programa u izvodivi strojni program. Tijekom prevođenja program punitelj izrađuje memorijske adrese prema dodijeljenom

memorijskom prostoru. Program povezivač uspostavlja veze između dijelova premjestivog strojnog programa koji su nezavisno prevedeni i gradi jedinstveni izvodivi strojni program.

Jezični procesori, pretprocesori i postprocesori slični su po građi i zadacima koje obavljaju. Budući da većina jezičnih procesora ima znatno složeniju strukturu od pretprocesora i postprocesora, u nastavku se razmatra samo građa, zadaci i svojstva jezičnih procesora. Jezični procesori prevode program zapisan jednim jezikom, koji se naziva *izvorni jezik*, u program zapisan drugim jezikom, koji se naziva *ciljni jezik*, odnosno jezični procesor prevodi *izvorni program* u *ciljni program*. Sam jezični procesor je program, te se jezik pomoću kojeg je izgrađen naziva *jezik izgradnje*. Na temelju definicije tri jezika, izvornog jezika L_i, ciljnog jezika L_c i jezika izgradnje L_g, jezični procesor *JP* prikazuje se na sljedeći način:

$$JP_{L_g}^{L_i \rightarrow L_c}$$

Svojstva formalnih jezika, automata i gramatika, koji su osnova procesa prevođenja, opisani su u udžbeniku S. Srbljića: "Jezični procesori 1: Uvod u teoriju formalnih jezika, automata i gramatika". U ovom udžbeniku primjenjuju se znanja o formalnim jezicima, automatima i gramatikama za gradnju jezičnih procesora. U nastavku uvodnog poglavlja u odjeljku 1.1 prikazan je rad jednostavnog jezičnog procesora, u odjeljku 1.2 opisan je način vrednovanja jezičnih procesora, u odjeljku 1.3 objašnjen je način projektiranja jezičnih procesora, u odjeljku 1.4 izvršena je razredba jezičnih procesora i u odjeljku 1.5 dat je kratki povijesni pregled razvoja jezičnih procesora.

U poglavljima 2-4, 6, 7 i 9 detaljno se opisuju pojedine faze rada jezičnog procesora. Programska okolina koja omogućuje izvođenje strojnog programa opisana je u poglavlju 5. U poglavlju 8 objašnjeni su postupci dorade premjestivog strojnog programa u izvodiv strojni program i priprema izvodivog strojnog programa za izvođenje.

1.1 Prikaz rada jednostavnog jezičnog procesora

Prevođenje izvornog programa u ciljni program ostvaruje se u dvije osnovne faze rada jezičnog procesora: *analiza izvornog programa* i *sinteza ciljnog programa*. Faza analize prethodi fazi sinteze kako bi se ustanovilo ima li u izvornom programu pogrešaka. Samo ispravni izvorni program moguće je prevesti u ciljni program. Nema li pogrešaka u izvornom programu, izvodi se sinteza ciljnog programa. Obje faze rada, analiza izvornog programa i sinteza ciljnog programa, složeni su procesi koji se izvode u više koraka.

Faza analize rastavlja izvorni program u sastavne dijelove, provjerava pravila jezika, prijavljuje pogreške i zapisuje izvorni program primjenom različitih struktura podataka u memoriju računala. Analiza izvornog programa izvodi se u više zasebnih koraka: *leksička analiza*, *sintaksna analiza* i *semantička analiza*. Pravila jezika nameću disciplinu u programiranje, što ima prednosti - čitljivost, ali i nedostatke - neučinkovitost. Uvijek je potrebno izbjegavati krajnosti:

> "*If the PL/I is the Fatal Disease,*
> *then perhaps Algol-68 is Capital Punishment*". [1]
>
> An Anonymous Compiler Writer

[1] PL/I jezik pruža široke mogućnosti i ostavlja veliku slobodu pri rješavanju različitih problema. Jedan te isti problem moguće je riješiti na više različitih načina, što korisniku omogućava osobni pristup i jedinstveno rješenje. Zbog omogućavanja osobnog pristupa, primjenom PL/I jezika postiže se veća učinkovitost programa, ali se zbog nečitljivosti otežava razmjena programa između korisnika. Nasuprot PL/I jeziku, Algol-68 je jezik koji ima stroga pravila u pisanju programa, čime se skoro u potpunosti istiskuje osobni pristup korisnika. Zbog čitljivosti, programi se lakše razmjenjuju između korisnika, ali se zato smanjuje mogućnost kreativnog rješenja i učinkovitost.

U fazi sinteze zapis izvornog programa, koji je generiran u fazi analize, postupno se prevodi u ciljni program. Sinteza ciljnog programa izvodi se u sljedećim koracima: *generiranje međukôda*, *strojno nezavisno optimiranje*, *generiranje strojnog programa*, *strojno zavisno optimiranje* i *priprema strojnog programa za izvođenje*.

U nastavku odjeljka prikazani su pojedini koraci rada jezičnog procesora, opisani su oblici koje poprima izvorni program za vrijeme prevođenja u ciljni program, objašnjeni su osnovni procesi prisutni u svim koracima rada jezičnog procesora, te su navedeni zadaci koje ti procesi obavljaju.

Leksička analiza

Pravila programskih jezika određuju dozvoljene oblike osnovnih elemenata jezika. Osnovni elementi programskih jezika su varijable, ključne riječi, konstante, operatori i specijalni znakovi. Pravila koja određuju dozvoljene oblike osnovnih elemenata jezika nazivaju se *leksička pravila*. Na primjer, leksičkim pravilima određuje se sljedeće: ime *varijable* je niz slova i brojaka koji započinje slovom; *ključne riječi* su ujedno i rezervirane riječi `ako`, `inače`, `za`, `dok`, `skoči`, `slučaj`, itd; *cjelobrojna konstanta* je niz dekadskih znamenaka; *decimalna konstanta* ima barem jedan broj ispred decimalne točke; itd.

Elementi jezika, kao što su varijable, ključne riječi i konstante, nazivaju se *leksičke jedinke*. Leksičke jedinke grupiraju se u više različitih klasa. Leksičke jedinke u istoj klasi imaju zajedničko pravilo koje određuje koji su dozvoljeni oblici za tu klasu. Na primjer, uobičajene klase leksičkih jedinki u programskim jezicima su:

IDN	-	označava klasu identifikatora (na primjer, identifikatori su imena varijabli, imena programa, imena polja, itd.);
KR	-	označava klasu ključnih riječi (na primjer, ključne riječi `ako`, `inače`, `za`, `dok`, `skoči`, itd.);
INT	-	označava klasu cjelobrojnih konstanti (na primjer, konstante 20, 1000 i 1200, itd.);
FLOAT	-	označava klasu konstanti zapisanu s posmačnim zarezom (na primjer, konstante 30.01, 0.005 i 1200.123, itd.);
SO	-	označava klasu specijalnih znakova i operatora (na primjer, specijalni znakovi `}`, `{`, `)`, `(`, `;`, itd; operatori `=`, `>`, `<`, itd.).

Leksička analiza je linearna analiza znakova izvornog programa koja provjerava jesu li leksičke jedinke u izvornom programu pravilno napisane. Znakovi izvornog programa slijedno se čitaju jedan po jedan s ciljem grupiranja ulaznih znakova u leksičke jedinke, prepoznavanja i određivanja klasa leksičkih jedinki, ispitivanja jesu li zadovoljena pravila koja određuju dozvoljene oblike leksičkih jedinki, izbacivanja bjelina i komentara, te ispisa pogrešaka.

Leksički analizator prevodi izvorni program u niz leksičkih jedinki. Dobiveni niz leksičkih jedinki je ulazni niz sintaksnog analizatora.

```
IzračunajCijenu ( )
{
    ako ( Količina > 20 )
        Cijena = 1000 ;
    inače
        Cijena = 1200 ;
    NovaCijena = Cijena ;
}
```

Slika 1.2: Primjer izvornog programa

Na primjer, izvorni program prikazan na slici 1.2 leksički analizator prevodi u niz uređenih parova oblika:

(*KlasaLeksičkeJedinke*, *LeksičkaJedinka*),

izbacuje sve bjeline iz izvornog programa (bjeline u izvornom programu označene su praznim kvadratićima u kojima nema ispisanog teksta), te prevodi izvorni program u niz leksičkih jedinki prikazan na slici 1.3.

(*IDN*, `IzračunajCijenu`), (*SO*, `(`), (*SO*, `)`), (*SO*, `{`),
(*KR*, `ako`), (*SO*, `(`), (*IDN*, `Količina`), (*SO*, `>`), (*INT*, `20`), (*SO*, `)`),
(*IDN*, `Cijena`), (*SO*, `=`), (*INT*, `1000`), (*SO*, `;`),
(*KR*, `inače`), (*IDN*, `Cijena`), (*SO*, `=`), (*INT*, `1200`), (*SO*, `;`),
(*IDN*, `NovaCijena`), (*SO*, `=`), (*IDN*, `Cijena`), (*SO*, `;`), (*SO*, `}`).

Slika 1.3: Niz leksičkih jedinki

Tijekom leksičke analize gradi se *tablica znakova* u koju se zapisuju dodatni parametri leksičkih jedinki. Za svaku leksičku jedinku stvara se zasebni zapis u tablici znakova s predviđenim mjestom za sve njezine dodatne parametre. Dok je njihov redoslijed u nizu leksičkih jedinki jednak njihovom redoslijedu u izvornom programu, leksičke jedinke u tablici znakova poredane su onim redoslijedom koji omogućuje njihovo lako i brzo pretraživanje i nadopunjavanje.

Dodatni parametri, na primjer, određuju tip varijable koji je `INTBroj` ili `FLOATBroj`, odnosno dodatni parametar određuje da li je vrijednost varijable cjelobrojna ili s posmačnim zarezom. Popuna tablice znakova započinje tijekom leksičke analize, a koristi se i nadopunjava u ostalim koracima rada jezičnog procesora. U našem primjeru, podatak o tipu konstante određuje se tijekom leksičke analize na temelju leksičkih pravila. Konstante se grupiraju u dvije različite klase: konstante zapisane s posmačnim zarezom (*FLOAT*) i cjelobrojne konstante (*INT*), dok se tip varijable određuje tijekom semantičke analize na temelju naredbi pridruživanja.

Sintaksna analiza

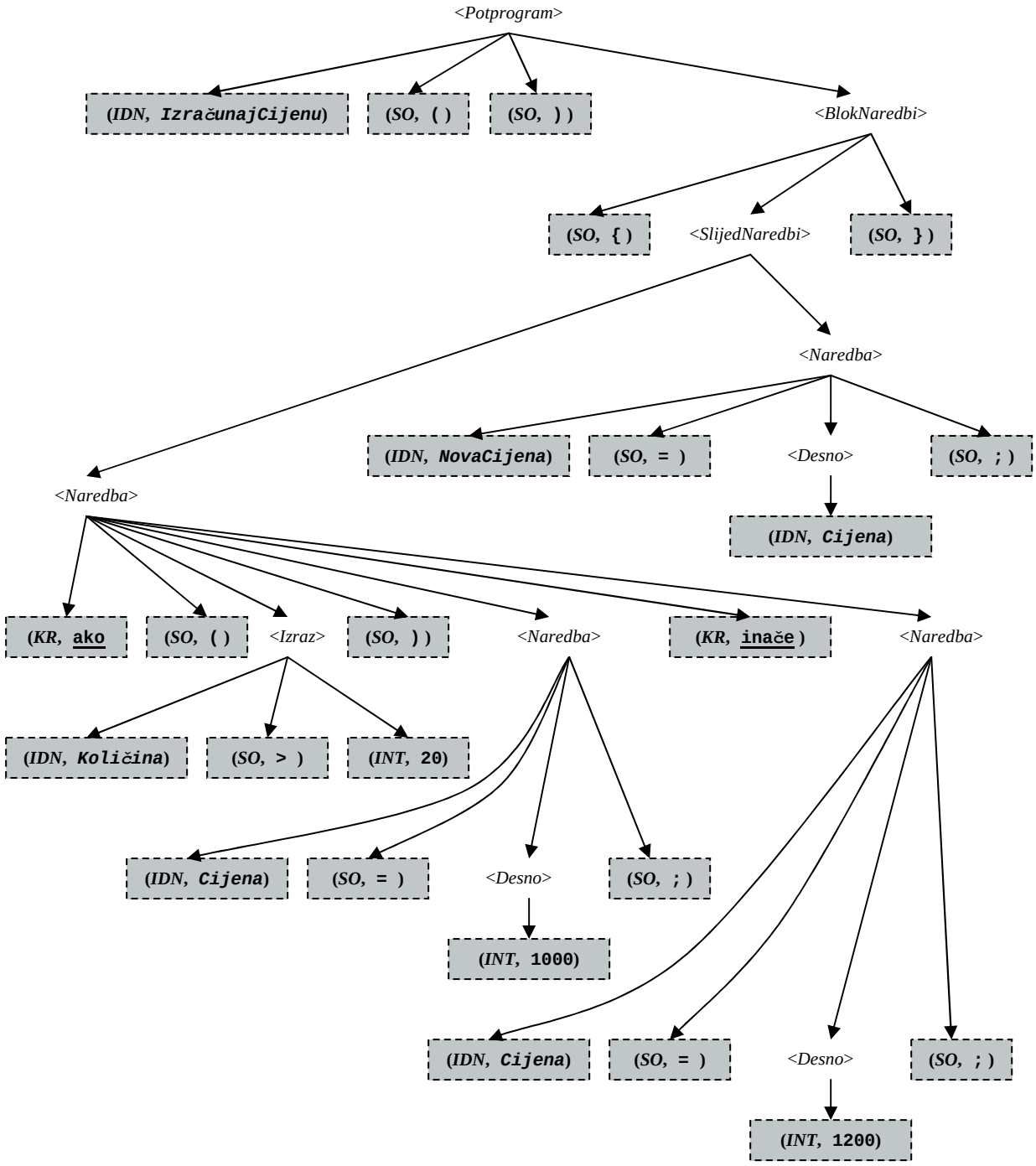
Programskim jezikom određena su pravila na koji način se koriste leksičke jedinke za gradnju ispravne naredbe. Na primjer, naredba grananja ima sljedeći oblik:

```
ako ( <Izraz> )
        <Naredba> ;
inače
        <Naredba> ;
```

Primjer pokazuje da u pravilima nisu zadane samo leksičke jedinke (u primjeru su leksičke jedinke **`ako`**, **`inače`**, **`(`**,**`)`**, i **`;`**), već su zadane i složenije strukture kao što su to <*Izraz*> i <*Naredba*>. Struktura <*Izraz*> zasebno se definira i predstavlja bilo koji logički izraz programskog jezika. U većini slučajeva pravila su rekurzivna, odnosno u definiciji naredbe ponovno se koristi naredba. U datom primjeru u definiciji naredbe grananja koristi se struktura <*Naredba*> koja predstavlja bilo koju naredbu programskog jezika, uključujući i naredbu grananja. Nadalje, pravilima je određena struktura čitavog programa. Na primjer, pravila određuju da se naredbe deklaracija varijabli nalaze na početku programa. Pravila koja određuju dozvoljene strukture naredbi, dozvoljene strukture dijelova programa i dozvoljene strukture čitavog programa nazivaju se *sintaksna pravila*, dok se dio jezičnog procesora koji provjerava ta pravila naziva *sintaksni analizator*.

Sintaksna analiza je hijerarhijska analiza. Leksičke jedinke koje generira leksički analizator grupiraju se u hijerarhijske skupine sa zajedničkim značenjem. Sintaksni analizator provjerava da li niz leksičkih jedinki zadovoljava sintaksnim pravilima zadanu hijerarhijsku strukturu izvornog programa. Najčešće se hijerarhijska struktura programa zadaje primjenom formalne gramatike i opisuje stablom. Sintaksni analizator gradi sintaksno stablo ili ispisuje poruku o pogrešci.

Za prethodno zadani izvorni program sintaksni analizator gradi stablo prikazano na slici 1.4.



Slika 1.4: Sintaksno stablo

Stablom je prikazano više hijerarhijskih skupina leksičkih jedinki koje su povezane zajedničkim značenjem. Na primjer, niz leksičkih jedinki:

(*IDN*, `Cijena`), (*SO*, =), (*INT*, `1000`) i (*SO*, `;`)

čine skupinu jedinki označenih oznakom <*Naredba*> sa značenjem naredbe pridruživanja:

```
Cijena = 1000;.
```

Ostale hijerarhijske skupine jedinki koje istodobno određuju sintaksnu strukturu i koje imaju točno određeno značenje su: <*Potprogram*>, <*BlokNaredbi*>, <*SlijedNaredbi*> i <*Desno*>. Iz samog naziva skupine moguće je odrediti njihovo značenje, dok podstablo kojemu su te oznake korijen određuje njihovu sintaksnu strukturu. Hijerarhijske skupine leksičkih jedinki određuju se na temelju značenja, što ovisi o sljedećem koraku rada jezičnog procesora koji interpretira značenje izvornog programa.

Sintaksno stablo sprema se na više različitih načina u memoriju računala: *izravno spremanje čitavog stabla primjenom za to pogodnih podatkovnih objekata kao što su mreže*, *izravno dinamičko spremanje dijelova stabla na stog* i *posredno dinamičko spremanje dijelova stabla na stog putem poziva rekurzivnih potprograma*. Optimirajući jezični procesori spremaju čitavo sintaksno stablo u memoriju računala. Sintaksno stablo naročito je pogodno za postupke optimiranja međukôda više razine, jer na izravan način sadrži podatke o strukturi izvornog programa. Podaci o strukturi izvornog programa koriste se tijekom analize tijeka izvođenja programa, analize toka podataka, analize zavisnosti podataka i analize pseudonima.

Ne koristi li jezični procesor postupke optimiranja međukôda više razine, dijelovi sintaksnog stabla spremaju se dinamički na stog tijekom sintaksne analize. Za potrebe sintaksne analize uzima se LIFO stog. Sintaksni analizator stalno uspoređuje sadržaj vrha stoga i pročitanu leksičku jedinku izvornog programa, te na temelju tih podataka primjenjuje određene akcije. Akcije sintaksnog analizatora su:

- `Pomak` kazaljke za čitanje leksičkih jedinki u izvornom programu na sljedeću leksičku jedinku,
- `Uzmi(NizZnakova)` s vrha stoga,
- `Stavi(NizZnakova)` na vrh stoga,
- `Ispiši("Sintaksna pogreška!")` ispisuje tekst sintaksne pogreške.

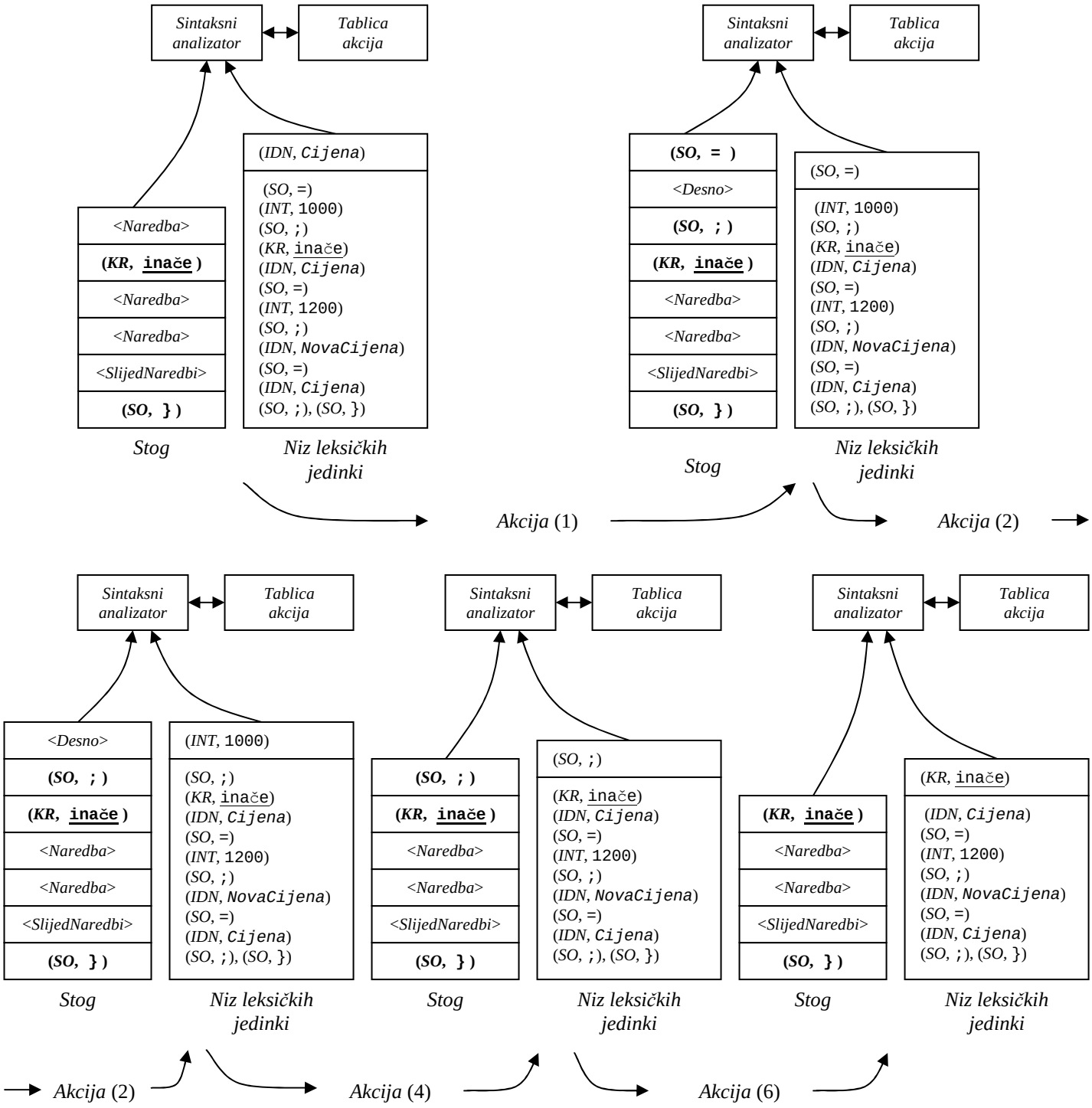
Na početku rada sintaksni analizator stavlja na stog oznaku strukture <*Program*> koja predstavlja strukturu čitavog izvornog programa. Kazaljka za čitanje leksičkih jedinki pokazuje na početnu leksičku jedinku izvornog programa. Akcije sintaksnog analizatora određuju se na temelju sintaksnih pravila. Na primjer, akcije sintaksnog analizatora za naredbu pridruživanja zadane su u tablici 1.1.

Leksička jedinka	*Sadržaj vrha stoga*	*Broj*	*Akcija*

(*IDN*, `xxx`)	<*Naredba*>	1	`Uzmi( <Naredba> ) s vrha stoga;`
			`Stavi( ;, <Desno>, = ) na vrh stoga;`
			`Pomakni kazaljku na sljedeću leksičku jedinku;`

(*SO*, =)	=	2	`Uzmi( = ) s vrha stoga;` `Pomakni kazaljku na sljedeću leksičku jedinku;`
(*IDN*, `xxx`)	<*Desno*>	3	`Uzmi( <Desno> ) s vrha stoga;` `Pomakni kazaljku na sljedeću leksičku jedinku;`
(*INT*, `xxx`)	<*Desno*>	4	`Uzmi( <Desno> ) s vrha stoga;` `Pomakni kazaljku na sljedeću leksičku jedinku;`
Sve ostale jedinke	<*Desno*>	5	`Ispiši("Sintaksna pogreška: Nepotpuna naredba pridruživanja!");`
(*SO*, ;)	;	6	`Uzmi( ; ) s vrha stoga;` `Pomakni kazaljku na sljedeću leksičku jedinku;`

Tablica 1.1: Tablica akcija sintaksnog analizatora za analizu naredbe pridruživanja



Slika 1.5: Dijelovi sintaksnog stabla spremljeni na stog tijekom sintaksne analize naredbe `Cijena = 1000;` primjenom akcija opisanih u tablici 1.1

Akcije navedene u tablici 1.1 spremaju dio sintaksnog stabla na stog. Na primjer, ako sintaksni analizator pročita u izvornom jeziku leksičku jedinku koja je u klasi identifikatora, a na vrhu stoga je oznaka <*Naredba*>, izvodi se akcija (1) tablice 1.1. S vrha stoga uzima se oznaka <*Naredba*>, a na stog se stavlja niz znakova `;`, `<Desno>` i `=`. Usporedbom tih znakova sa znakovima na slici 1.4 uoči se da su ti znakovi dio podstabla kojemu je korijen oznaka <*Naredba*>. Tijekom daljnje sintaksne analize ostale akcije uzimaju dio po dio stabla sa stoga. Sintaksna analiza

uspješno završava ako na kraju analize ostane prazni stog, a kazaljka pokazuje na kraj niza leksičkih jedinki. Slika 1.5 prikazuje dio stabla spremljenog na stog i dinamičke promjene sadržaja stoga tijekom analize naredbe pridruživanja `Cijena=1000;`. Dijelovi stabla spremljeni na stog predstavljaju korijene u tom trenutku još neobrađenih podstabla.

Koristi li se formalna gramatika, sintaksni analizator moguće je programski ostvariti tehnikom rekurzivnog spusta. Sintaksnim pravilima jezika pridružuju se potprogrami. Na primjer, produkcijama gramatike:

<*Naredba*> → (*IDN*, xxx) (*SO*, =) <*Desno*> (*SO*, ;)
<*Desno*> → (*IDN*, xxx) | (*INT*, xxx)

pridruže se potprogrami:

```
Naredba()
{
   ako( Znak.Klasa != IDN )
         Ispiši( " Sintaksna pogreška: Nedostaje identifikator " );

   Čitaj ( Znak );
   ako( Znak.Ime != = )
         Ispiši( " Sintaksna pogreška: Nedostaje znak = " );

   Čitaj ( Znak );
   Desno();

   ako(Znak.Ime != ; )
         Ispiši( " Sintaksna pogreška: Nedostaje znak ; " );

   Čitaj ( Znak );
}

Desno()
{
   slučaj ( Znak.Klasa )
   {
         IDN:
               Čitaj ( Znak );

         INT:
               Čitaj ( Znak );

         Svi ostali znakovi:
               Ispiši( " Sintaksna pogreška: Nepotpuna naredba
               pridruživanja! " );
   }
}
```

Slika 1.6: Potprogrami sintaksnog analizatora za analizu naredbe pridruživanja

Na slici 1.6 koriste se sljedeći potprogrami:

`Čitaj(Znak)`	potprogram slijedno čita ulazni niz leksičkih jedinki koji je pripremio leksički analizator i sprema vrijednost leksičke jedinke u globalnu varijablu znak,
`Ispiši()`	potprogram ispisuje zadanu poruku,
`Naredba()`	potprogram pridružen produkciji gramatike za analizu naredbe pridruživanja,
`Desno()`	potprogram pridružen produkciji gramatike za analizu desne strane naredbe pridruživanja.

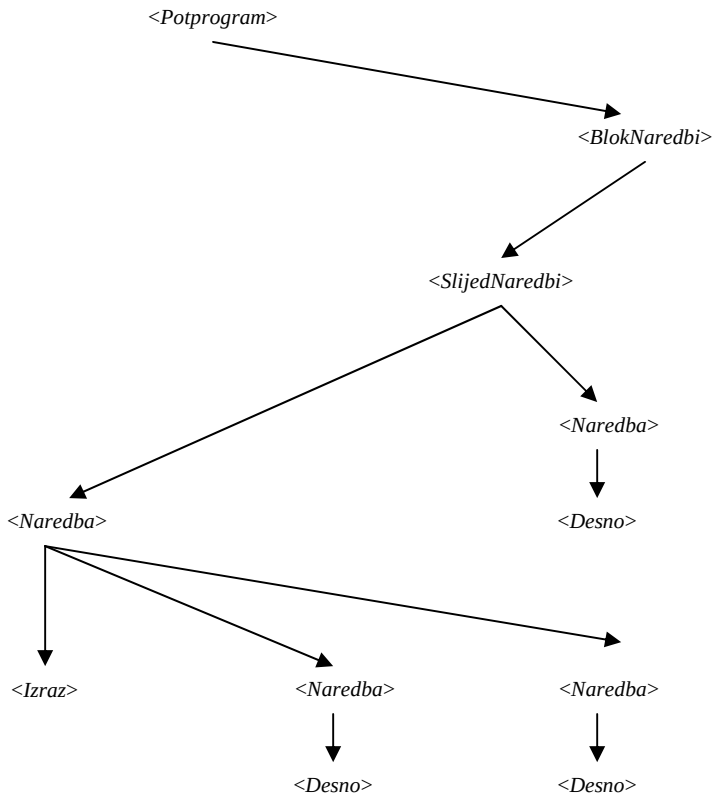
Prije poziva potprograma koji provjerava određeno sintaksno pravilo, potprogram `Čitaj(Znak)` globalnoj varijabli `Znak` pridruži vrijednost sljedeće leksičke jedinke izvornog programa. Na primjer, pri pozivu potprograma `Naredba()` izvode se sljedeće naredbe:

```
Čitaj ( Znak );
Naredba();
```

Prva naredba čita sljedeću leksičku jedinku i pridružuje njezinu vrijednost globalnoj varijabli `Znak`, a druga naredba poziva potprogram `Naredba()`.

Budući da je leksička jedinka uređeni par, `Znak.Klasa` poprima vrijednost klase leksičke jedinke (u našem primjeru *IDN*, *KR*, *INT*, *FLOAT* ili *SO*), a `Znak.Ime` poprima vrijednost bilo imena identifikatora (u našem primjeru `IzračunajCijenu`, `Cijena`, `NovaCijena`, itd.), vrijednost konstante (u našem primjeru `1000`, `1200`, itd.), podatak o ključnoj riječi (u našem primjeru `ako`, `inače`, itd.), ili podatak o specijalnom znaku ili operatoru (u našem primjeru `=`, `>`, itd.). Nadalje, treći podatak `Znak.Tip` poprima vrijednost prema tipu podatka varijable ili konstante (u našem primjeru `INTBroj` ili `FLOATBroj`). Ako tip varijable nije određen, vrijednost podatka `Znak.Tip` jednaka je NULL. Podatak `Znak.Klasa` koristi se tijekom sintaksne analize, podatak `Znak.Ime` koristi se tijekom sintaksne i semantičke analize, dok se podatak `Znak.Tip` koristi tijekom semantičke analize za provjeru tipova podataka varijabli i konstanti.

Slika 1.7 prikazuje stablo aktiviranja potprograma za analizu prikazanu sintaksnim stablom na slici 1.4. Usporedba slika 1.7 i 1.4 pokazuje podudarnost stabla aktiviranja potprograma s dijelovima sintaksnog stabla.



Slika 1.7: Stablo aktiviranja potprograma sintaksnog analizatora ostvarenog tehnikom rekurzivnog spusta za sintaksno stablo prikazano na slici 1.4

Semantička analiza

Semantička pravila su interpretacijska pravila koja povezuju izvođenje izvornog programa s ponašanjem računala. Na primjer, pridruži li se određenoj varijabli zbroj cjelobrojne i vrijednosti s posmačnim zarezom, zbroj poprima vrijednost s posmačnim zarezom. Nadalje, semantika jezika određuje skup dopuštenih značenja. Na primjer, semantička pravila nekih jezika ne dopuštaju množenje cjelobrojnih i vrijednosti s posmačnim zarezom, već je obje varijable potrebno prethodno pretvoriti u vrijednost s posmačnim zarezom ili cjelobrojnu. U tim jezicima operator množenja ima značenje cjelobrojnog ili množenja vrijednosti s posmačnim zarezom, ali nema značenje množenja cjelobrojnih i vrijednosti s posmačnim zarezom.

Semantički analizator sačinjava niz potprograma koji provjeravaju deklaraciju varijabli, tipove podataka u izrazima, indeksiranje polja, parametre potprograma, te tijek izvođenja programa. Potprogrami semantičkog analizatora izvode se istodobno s potprogramima sintaksnog analizatora, odnosno potprogrami sintaksnog analizatora pozivaju potprograme semantičkog analizatora. Semantička analiza je most između faze analize i faze sinteze.

Potprogrami semantičkog analizatora ugniježđeni su u hijerarhijsku strukturu i vežu se uz sintaksna pravila. Slika 1.8 prikazuje na koji se način proširuje potprogram analize naredbe pridruživanja i potprogram analize desne strane naredbe pridruživanja (pogledaj definiciju potprograma na slici 1.6). Definicija potprograma semantičkog analizatora prikazana je na slici 1.9.

```
Naredba( ZnakLijevo )
{
  ako( Znak.Klasa != IDN )
      Ispiši( " Sintaksna pogreška: Nedostaje identifikator " );

  Čitaj ( Znak );
  ako( Znak.Ime != = )
      Ispiši( " Sintaksna pogreška: Nedostaje znak = " );

  Čitaj ( Znak );
  Desno( ZnakLijevo );

  ako( Znak.Ime != ; )
      Ispiši( " Sintaksna pogreška: Nedostaje znak ; " );

  Čitaj ( Znak );
}

Desno( ZnakLijevo )
{
  slučaj ( Znak.Klasa )
  {
            IDN:
                  ProvjeriTipove (ZnakLijevo, Znak);
                  Čitaj ( Znak );

            INT:
                  ProvjeriTipove (ZnakLijevo, Znak);
                  Čitaj ( Znak );

            Svi ostali znakovi:
                  Ispiši( " Sintaksna pogreška: Nepotpuna naredba
                  pridruživanja! " );
  }
}
```

Slika 1.8: Pozivi potprograma semantičkog analizatora *ProvjeriTipove*() iz potprograma sintaksnog analizatora koji analizira desnu stranu naredbe pridruživanja

U našem primjeru leksički analizator razvrstava konstante u cjelobrojne (*INT*) i konstante zapisane s posmačnim zarezom (*FLOAT*), a zadatak semantičkog analizatora je određivanje tipa varijable. Varijabla se deklarira prvom naredbom koja pridružuje neku vrijednost toj varijabli. Zato se u početnom dijelu potprograma *ProvjeriTipove*() provjerava je li na desnoj strani naredbe pridruživanja varijabla (*Znak.Klasa*==**IDN**), i ako jest, semantički analizator provjerava je li toj varijabli već prethodno određen tip (*Znak.Tip*==**NULL**). Nije li tip varijable na desnoj strani naredbe pridruživanja određen, semantički analizator prijavljuje pogrešku. Ako je tip varijable određen, nastavlja se daljnja semantička analiza s ispitivanjem lijeve strane naredbe pridruživanja. Semantički analizator provjerava je li varijabli na lijevoj strani određen tip (*ZnakLijevo.Tip*==**NULL**). Ako varijabli na lijevoj strani nije određen tip, onda se tip toj varijable određuje prema tipu varijable na desnoj strani (*ZnakLijevo.Tip=Znak.Tip*). U posljednjem dijelu potprograma *ProvjeriTipove*() provjerava se da li lijeva i desna strana naredbe pridruživanja imaju različite tipove (*ZnakLijevo.Tip*!=*Znak.Tip*). Ako su tipovi različiti, semantički analizator ispisuje poruku da tipovi podataka u naredbi pridruživanja nisu odgovarajući.

```
ProvjeriTipove ( ZnakLijevo, Znak)
{
    ako (( Znak.Klasa == IDN ) && ( Znak.Tip == NULL ))
          Ispiši( "Semantička pogreška: NEDEFINIRAN TIP VARIJABLE Znak.Ime" );

    ako ( ZnakLijevo.Tip == NULL )
          ZnakLijevo.Tip = Znak.Tip;

    ako (ZnakLijevo.Tip != Znak.Tip )
          Ispiši( "Semantička pogreška: NEODGOVARAJUĆI TIPOVI PODATAKA
          ZnakLijevo.Ime i Znak.Ime" );
}
```

Slika 1.9: Potprogram *ProvjeriTipove*() semantičkog analizatora

Potprogram *ProvjeriTipove*() poziva se iz potprograma *Desno*() tijekom sintaksne analize u trenutku kada su poznati parametri obje strane naredbe pridruživanja (pogledaj sliku 1.8).

Prije poziva potprograma *Naredba*(), potrebno je izvesti dvije naredbe:

```
Čitaj ( Znak );
ZnakLijevo = Znak;
Naredba( ZnakLijevo );
```

Prva naredba pridružuje vrijednost leksičke jedinke koja se nalazi na lijevoj strani naredbe pridruživanja varijabli *ZnakLijevo*, druga naredba čita sljedeću jedinku i pridružuje njezinu vrijednost globalnoj varijabli *Znak*, a treća naredba poziva potprogram *Naredba*().

Budući da je potrebno vrijednosti parametra leksičke jedinke koja se nalazi na lijevoj strani znaka jednakosti u naredbi pridruživanja sačuvati sve do trenutka analiziranja leksičke jedinke koja se nalazi na desnoj strani znaka jednakosti, semantički analizator pokreće prenošenje parametra leksičkih jedinki iz jednog potprograma u ostale potprograme. U tehnici rekurzivnog spusta to se postiže primjenom ulazno-izlaznih parametara potprograma. Na primjer, parametri leksičke jedinke *ZnakLijevo* koja se nalazi lijevo od znaka jednakosti prenose se u potprogram *Naredba*() kao ulazni parametar *Naredba*(*ZnakLijevo*). Prenose li se parametri leksičke jedinke *ZnakLijevo* izravno putem stoga, potrebno je akcije semantičkog analizatora proširiti na način kako je to pokazano u tablici 1.2. U tablici su prikazane samo one akcije koje je potrebno promijeniti u odnosu na akcije koje su opisane u tablici 1.1. Akcija (1) sprema parametre leksičke jedinke *ZnakLijevo* na stog, a akcije (3) i (4) uzimaju parametre leksičke jedinke sa stoga i koriste ih u semantičkoj analizi naredbe pridruživanja.

Prenošenje parametara je osnovni proces semantičkog analizatora i moguće ga je promatrati na sintaksnom stablu. Promatrajući sintaksno stablo, parametri se prenose u dva osnovna smjera: *od vrha stabla prema dnu stabla* i *od dna stabla prema vrhu stabla*. Prethodni primjer leksičke jedinke *ZnakLijevo* primjer je prenošenja parametra od vrha stabla prema dnu stabla.

Prenošenje parametra od dna stabla prema vrhu stabla zorno se prikazuje na sljedećem primjeru semantičke analize. Prethodno je definirano da se tip varijable deklarira prvom naredbom pridruživanja koja toj varijabli pridružuje

neku vrijednost. Na primjer, tip varijable *Cijena* deklarira se naredbom *Cijena*=1000;. Potprogram semantičkog analizatora *ProvjeriTipove* na temelju cjelobrojne konstante 1000 određuje da je *Cijena* cjelobrojna varijabla. Podatak da je *Cijena* cjelobrojna varijabla prenosi se od dna stabla prema vrhu stabla sljedećim putem: od čvora <*Desno*> koji označava analizu desne strane naredbe *Cijena*=1000; do čvora <*Naredba*> koji označava analizu cjelokupne naredbe *Cijena*=1000;, a zatim do sljedećeg čvora <*Naredba*> koji označava analizu naredbe uvjetnog grananja **ako**(<*Izraz*>)<*Naredba*>;**inače**<*Naredba*>;. Nakon toga podatak o tipu varijable prenosi se od vrha stabla prema dnu stabla, i to preko čvora <*Naredba*> koji označava analizu naredbe *Cijena*=1200; do čvora <*Desno*> koji označava analizu desne strane naredbe *Cijena*=1200;. Potprogram *ProvjeriTipove* provjerava je li tip cjelobrojne konstane 1200 jednak tipu cjelobrojne varijable *Cijena*. Budući da je tip konstante jednak tipu varijable, semantički analizator ne prijavljuje pogrešku. Na sličan način tip varijable *Cijena* prenosi se sve do naredbe *NovaCijena=Cijena*; preko više čvorova od dna stabla prema vrhu stabla do čvora <*SlijedNaredbi*>, a zatim od vrha stabla prema dnu stabla sve do čvora <*Desno*> koji označava analizu desne strane naredbe *NovaCijena=Cijena*;. Na temelju podatka o tipu varijable *Cijena* određuje se tip varijable *NovaCijena*.

Leksička jedinka	*Sadržaj vrha stoga*	*Broj*	*Akcija*
(*IDN*, *xxx*)	<*Naredba*>	1	*Uzmi*(<*Naredba*>) s vrha stoga; *Stavi*(;, **{*ZnakLijevo*}**, <*Desno*>, =) na vrh stoga; *Pomakni* kazaljku na sljedeću leksičku jedinku;
(*IDN*, *xxx*)	<*Desno*>	3	*Uzmi*(<*Desno*>) s vrha stoga; *Uzmi*(**{*ZnakLijevo*}**) s vrha stoga stoga; *Pomakni* kazaljku na sljedeću leksičku jedinku;
(*INT*, *xxx*)	<*Desno*>	4	*Uzmi*(<*Desno*>) s vrha stoga; *Uzmi*(**{*ZnakLijevo*}**) s vrha stoga; *Pomakni* kazaljku na sljedeću leksičku jedinku;

Tablica 1.2: Tablica akcija sintaksnog i semantičkog analizatora za analizu naredbe pridruživanja

U datom programu, zbog nedeklarirane varijable *Količina*, potprogram semantičkog analizatora *ProvjeriTipove* ispisao bi sljedeću semantičku pogrešku: "Semantička pogreška: NEDEFINIRAN TIP VARIJABLE Količina".

Generiranje međukôda

Nakon što uspješno završe leksička, sintaksna i semantička analiza, većina jezičnih procesora generira međukôd. Generiranje međukôda je početni korak sinteze ciljnog programa. Međukôd je zapis izvornog programa koji se izravno sprema u memoriju računala i koji je jedan od međukoraka u cjelokupnom postupku sinteze ciljnog programa. Slično potprogramima semantičkog analizatora, potprogrami generatora međukôda pozivaju se iz potprograma sintaksnog analizatora. Često su potprogrami generatora međukôda proširenja potprograma semantičkog analizatora, te se u literaturi ovaj korak rada jezičnog procesora ponekad svrstava u semantičku analizu.

Izborom razine i oblika međukôda nastoji se u što većoj mogućoj mjeri pojednostaviti dva osnovna procesa prevođenja tijekom sinteze ciljnog programa: proces prevođenja izvornog programa u međukôd i proces prevođenja međukôda u ciljni program. Osnovne razine međukôda su: *međukôd više razine*, *međukôd srednje razine* i *međukôd niže razine*. Naredbe međukôda više razine sliče naredbama viših programskih jezika sa sačuvanim programskim petljama i sačuvanim apstraktnim tipovima podataka. Za razliku od viših programskih jezika, naredbe međukôda više razine oslobođene su svih sintaksnih dijelova koji nisu potrebni u daljnjem tijeku sinteze ciljnog programa, kao što su zagrade {} koje označavaju početak i kraj bloka naredbi, oznaka kraja naredbe ;, itd. Nadalje, u međukôdu više razine nema naredbi deklaracija, jer su svi podaci koje definiraju te naredbe već tijekom semantičke analize spremljeni u tablici znakova. Za razliku od međukôda više razine, naredbe međukôda niže razine slične su naredbama strojnog jezika i u većini slučajeva jedna naredba međukôda niže razine prevodi se izravno u jednu naredbu strojnog jezika. Međukôd srednje razine zadržava neka obilježja međukôda više razine kao što su simbolička imena varijabli, dok se složene programske petlje razlažu u više jednostavnih naredbi grananja koje su slične strojnim naredbama grananja. Nadalje,

apstrakni tipovi podataka kao što su višedimenzionalna polja razlažu se u osnovne podatkovne objekte ciljnog jezika, kao što su na primjer vektori (odnosno jednodimenzionalno polje), liste i mreže.

Po svojom obliku međukôd se dijeli na *linearni*, *postfiksni* i *grafički*. Oblik međukôda ovisi o linearizaciji sintaksnog stabla. Naredbe lineranog oblika moguće je slijedno izvoditi na procesoru računala. Budući da je tijekom generiranja međukôda sintaksna stablasta struktura tek djelomično linearizirana, naredbe međukôda u postfiksnom obliku nije moguće izravno slijedno izvoditi na procesoru računala. Za određivanje redoslijeda njihovog izvođenja potrebno je koristiti potisni stog. Grafički oblik međukôda sličan je sintaksnom stablu, jer sintaksna stablasta struktura nije linearizirana. Više je različitih oblika grafičkog međukôda (sažeto sintaksno stablo, izravni graf bez petlji, graf zavisnosti programa, itd.) i linearnog međukôda (troadresne naredbe ostvarene kao četvorke, troadresne naredbe ostvarene kao trojke, itd.).

Na primjer, troadresne naredbe su linearne strukture i u većini slučajeva međukôd je srednje razine. Sljedeće tri naredbe primjer su troadresnih naredbi srednjeg međukôda:

`X := Y` *Naredba pridruživanja. X* je ime varijable, a *Y* je ime varijable ili konstante. U stvarnom međukôdu imena su kazaljke koje pokazuju na mjesto u tablici znakova gdje su te varijable spremljene.

`goto L` *Naredba bezuvjetnog grananja.* Izvođenje programa nastavlja se troadresenom naredbom označenom oznakom *L*.

`if X relop Y goto L` *Naredba uvjetnog grananja*. Operator *relop* je relacijski operator koji se primjenuje na *X* i *Y*. Ako je uvjet zadovoljen, onda se izvođenje programa nastavlja naredbom označenom oznakom *L*. Nije li uvjet ispunjen, izvođenje programa se nastavlja naredbom koja slijedi neposredno iza naredbe grananja.

Tablica 1.3 prikazuje primjer troadresnih naredbi međukôda srednje razine. Izvorni program korišten u prethodnim primjerima proširen je naredbom `PosPor=8;`. Slike 1.10 i 1.11 prikazuju proširenje potprograma sintaksnog analizatora koji analizira desnu stranu naredbe pridruživanja potprogramom koji generira naredbe međukôda. Potprogram `ZapišiKôd()` nadopisuje naredbe u datoteku ili strukturu podatka u koju se sprema međukôd.

Naredbe izvornog programa	*Troadresni međukôd*
`IzračunajCijenu ()` `{`	
`ako ( Količina > 20 )`	`if Količina > 20 goto L1`
	`goto L2`

`{`
`Cijena = 1000;`
`PosPor = 8;`
`}`
	`L1: Cijena := 1000`
	`PosPor := 8`
	`goto L3`

`inače`
`{`
`Cijena = 1200;`
`PosPor = 8:`
`}`
	`L2: Cijena := 1200`
	`PosPor := 8`

`NovaCijena = Cijena;` `PosPor = PosPor / 2;`	`L3: NovaCijena := Cijena` `PosPor := PosPor / 2`
`}`	

Tablica 1.3: Troadresne naredbe međukôda srednje razine

Potprogrami generatora međukôda koriste mehanizme prenošenja parametara slične mehanizmima prenošenja parametara semantičkog analizatora. Na primjer, parametri su imena oznaka `L2` i `L3`. Ime oznake `L2` prenosi se iz čvora sintaksnog stabla koji označava analizu naredbe uvjetnog grananja **ako**(<*Izraz*>)<*Naredba*>;**inače** <*Naredba*>; pa sve do čvora stabla koji označava analizu naredbe pridruživanja `Cijena=1200;`. Ime oznake `L3` prenosi se iz čvora

sintaksnog stabla koji označava analizu naredbe pridruživanja `PosPor=8;` pa sve do čvora stabla koji označava analizu naredbe `NovaCijena=Cijena;`.

```
Desno( ZnakLijevo )
{
  slučaj ( Znak.Klasa )
  {
          IDN:
                GenMeđukôdPrid (ZnakLijevo, Znak);
                Čitaj ( Znak );

          INT:
                ProvjeriTipove (ZnakLijevo, Znak);
                GenMeđukôdPrid (ZnakLijevo, Znak);
                Čitaj ( Znak );

          Svi ostali znakovi:
                Ispiši( " Sintaksna pogreška: Nepotpuna naredba
                pridruživanja! " );
  }
}
```

Slika 1.10: Poziv potprograma generatora međukôda `GenMeđukôd()` iz potprograma sintaksnog analizatora koji analizira desnu stranu naredbe pridruživanja

```
GenMeđukôdPrid ( ZnakLijevo, Znak)
{
     ZapišiKôd ( "ZnakLijevo.Ime := Znak.Ime" );
}
```

Slika 1.11: Potprogram generatora međukôda `GenMeđukôdPrid()`

Strojno nezavisno optimiranje

Optimiranje je postupak poboljšanja kvalitete generiranog međukôda ili ciljnog programa. Kvaliteta programa mjeri se brzinom njegovog izvođenja i veličinom potrebnog memorijskog prostora. Kvaliteta se unapređuje učinkovitom uporabom cjevovoda i aritmetičko-logičkih jedinki procesora računala, memorijske hijerarhije, itd. Poboljšana kvaliteta ciljnog programa postižu se nizom pretvorbi. Zahtijeva se da sve primijenjene pretvorbe sačuvaju značenje programa. Pretvorbe se dijele u dvije osnovne grupe: *pripremne pretvorbe* i *postupci optimiranja*. Pripremne pretvorbe preuređuju međukôd ili ciljni program tako da se omogući uspješno i učinkovito izvođenje postupka optimiranja.

Različite razine međukôda prilagođene su različitim postupcima optimiranja. Postupci optimiranja obično se grupiraju u dvije klase ovisno o razini međukôda: *strojno nezavisno optimiranje* i *strojno zavisno optimiranje*. Strojno nezavisno optimiranje ne uzima u obzir svojstva arhitekture računala na kojemu se izvodi ciljni program. Za razliku od strojno nezavisnog optimiranja, strojno zavisno optimiranje uzima u obzir svojstva arhitekture računala. Nakon što se generira niži međukôd ili ciljni program, strojno zavisno optimiranje izabire odgovarajuće strojne naredbe, redoslijed njihovog izvođenja, itd.

Optimiranju prethodi niz analiza: analiza tijeka izvođenja programa, analiza toka podataka, analiza pseudonima i analiza zavisnosti podataka. Međukôd ili ciljni program optimira se na temelju prikupljenih podataka o tijeku izvođenja programa, načinu uporabe podataka i zavisnosti pojednih naredbi. Različite analize *moraju* osigurati da učinjene promjene u međukôdu ili ciljnom programu sačuvaju značenje programa.

Na primjer, *analiza tijeka izvođenja programa* rastavlja međukôd zadan u drugom stupcu tablice 1.3 u pet osnovnih blokova. Osnovni blok je niz naredbi koje se slijedno izvode, i to od početne naredbe u osnovnom bloku pa sve do završne naredbe. Osim završne naredbe u bloku nema drugih naredbi grananja. U zadanom programu, naredba prvog osnovnog bloka ispituje uvjet grananja (`if Količina>20 goto L1`), naredba drugog osnovnog bloka jest

bezuvjetni skok na oznaku L2 (goto L2), naredbe trećeg osnovnog bloka izvode se u slučaju ispunjena uvjeta (L1:*Cijena*:=1000, *PosPor*:=8, goto L3), naredbe četvrtog osnovnog bloka izvode se u slučaju neispunjenja uvjeta (L2:*Cijena*:=1300, *PosPor*:=8), a naredbe petog bloka čine nastavak programa (L3:*NovaCijena*:=*Cijena*, *PosPor*:=*PosPor*/2). Na temelju podataka dobivenih *analizom toka podataka* i *analizom zavisnosti podataka* zaključuje se da naredba *PosPor*:=8 u trećem i četvrtom bloku definira istu vrijednost varijabli *PosPor*. Nadalje, vrijednost varijable *PosPor* ne koristi se u trećem i četvrtom osnovnom bloku, već se ona koristi tek u petom osnovnom bloku (naredba *PosPor*:=*PosPor*/2). Tijekom generiranja naredbi strojnog jezika, naredbe međukôda koje definiraju vrijednost varijable prevode se u niz naredbi koje u registru čuvaju definiranu vrijednost. Da se izbjegne nepotrebno zauzeće registara u duljem vremenskom razdoblju, postupci optimiranja nastoje naredbe definiranja vrijednosti varijable pomaknuti što je moguće bliže mjestu samog korištenja varijable. Time se smanji vrijeme zauzetosti registra, te je isti registar moguće koristiti za druge potrebe. *Postupci pripremne pretvorbe* umeću novu definiciju varijable u peti osnovni blok kako je to prikazano u drugoj koloni tablice 1.4. Umetnuta naredba *PosPor*:=8 je u drugoj koloni bliža mjestu korištenja, odnosno bliža je naredbi *PosPor*:=*PosPor*/2, jer se nalazi u istom osnovnom bloku.

Nakon pripremne pretvorbe, ponovno se pokreću postupci analize. Postupci analize pronalaze višestruku definiciju naredbe pridruživanja *PosPor*:=8. *Postupak optimiranja* ostavlja samo jednu naredbu u petom osnovnom bloku međukôda, te odbacuje dvije naredbe *PosPor*:=8 iz trećeg i četvrtog osnovnog bloka međukôda. Rezultat postupaka optimiranja prikazan je u trećoj koloni tablice 1.4. Usporedbom prve i treće kolone tablice 1.4 zaključuje se da nastali međukôd ima manji broj naredbi (umjesto dvije naredbe pridruživanja ostaje samo jedna naredba) i smanjuje se vrijeme od trenutka definicije varijable do trenutka njezine uporabe.

Troadresni međukôd	*Pripremna pretvorba međukôda*	*Optimirani troadresni međukôd*
if *Količina*>20 goto L1	if *Količina*>20 goto L1	if *Količina*>20 goto L1
goto L2	goto L2	goto L2
L1: *Cijena* := 1000 ***PosPor* := 8** goto L3	L1: *Cijena* := 1000 ***PosPor* := 8** goto L3	L1: *Cijena* := 1000 goto L3
L2: *Cijena* := 1200 **PosPor := 8**	L2: *Cijena* := 1200 **PosPor := 8**	L2: *Cijena* := 1200
L3: *NovaCijena* := *Cijena* *PosPor* := *PosPor* / 2	**L3: *PosPor* := 8** *NovaCijena* := Cijena *PosPor* := *PosPor* / 2	**L3: *PosPor* := 8** *NovaCijena* := Cijena *PosPor* := *PosPor* / 2

Tablica 1.4: Strojno nezavisna optimizacija troadresnih naredbi međukôda srednje razine

Generiranje strojnog programa

Razina međukôda, njegova struktura i oblik imaju utjecaj na generiranje strojnog programa. *Izbor strukture međukôda* ima utjecaja na osnovni algoritam generatora strojnog programa. Ako je međukôd u postfiksnom obliku, redoslijed generiranja strojnih naredbi određuje se primjenom potisnog stoga. Ako je međukôd u grafičkom obliku, redoslijed generiranja strojnih naredbi određuje se obilaskom sintaksnog stabla. Budući da su naredbe lineranog međukôda već poredane onim redoslijedom kojim se izvode na procesoru računala, algoritam generatora strojnog programa je znatno jednostavniji, jer nije potrebno odrediti redoslijed generiranja strojnih naredbi.

Izbor razine međukôda ima utjecaja na složenost postupaka generiranja strojnog programa. Budući da se jedna naredba međukôda niže razine u većini slučajeva prevodi u samo jednu strojnu naredbu, generiranje strojnog programa na temelju međukôda niže razine je znatno jednostavnije od generiranja strojnog programa na temelju međukôda više razine. Tijekom generiranja strojnog programa na temelju međukôda više razine, osim izbora odgovarajućih strojnih naredbi, zamjene simboličkih registra stvarnim registrima i određivanja numeričkih vrijednosti memorijskih adresa, dodatno se izvode i postupci prevođenja međukôda više u nižu razinu: razlaganje složenih naredbi petlji u jednostavne naredbe grananja strojnog jezika, pretvorba složenih apstraktnih tipova podataka u jednostavne podatkovne objekte strojnog jezika, itd.

Izbor oblika međukôda ima utjecaja na učinkovitost generiranog strojnog programa. Složeniji oblici grafičkog međukôda, kao što je izravni graf bez petlji, znatno pospješuju postupke optimiranja strojnog programa, jer grafička struktura međukôda izravno sadrži informaciju o strukturi programa.

Generator strojnog programa priprema strojne naredbe i simulira njihovo izvođenje. *Priprema strojne naredbe* uključuje određivanje operacije naredbe i njezinih operanada. Tijekom pripreme naredbe izabere se odgovarajuća strojna naredba procesora računala, najpovoljnije mjesto operanada (registar ili memorija) i mjesto gdje se sprema rezultat izvođenja naredbe.

Strojne naredbe uzima simulator računala. Za potrebe *simulacije*, generator strojnog programa simulira izvođenje naredbe primjenom modela računala. U model računala uključeni su podaci o naredbenom skupu procesora, registrima procesora, procesorskom cjevovodu, aritmetičko-logičkim jedinkama procesora, memorijskim lokacijama, memorijskoj hijerarhiji, operacijskom sustavu, te o ostalim elementima koji čine arhitekturu računala. Stanje modela računala određeno je sadržajem registara procesora, stanjem aritmetičko-logičkih jedinki, stanjem procesorskog cjevovoda, sadržajem memorijskih lokacija, itd. Simulacija uzima strojne naredbe, interpretira ih na temelju stanja modela računala, te mijenja stanje modela računala ovisno o prethodnom stanju i ovisno o značenju naredbe.

Najjednostavniji model generiranja strojnog programa moguće je pokazati na primjeru pretvorbe troadresnih naredbi međukôda srednje razine u mnemoničke naredbe mikroporcesora Motorola MC 6800. Budući da su naredbe međukôda srednje razine, jedna troadresna naredba obično se prevodi u nekoliko mnemoničkih naredbi. Nadalje, troadresne naredbe su linearnog oblika, te nije potrebno dodatnim algoritmima odrediti redoslijed izvođenja mnemoničkih naredbi.

Za potrebe generiranja mnemoničkih naredbi koristi se jednostavan postupak, gdje se određeni tip troadresne naredbe prevodi uvijek u isti unaprijed određeni slijed mnemoničkih naredbi. Prethodno je u odjeljku dan primjer tri tipa troadresnih naredbi: naredba pridruživanja, naredba bezuvjetnog grananja i naredba uvjetnog grananja. Na primjer, troadresna naredba pridruživanja *X*:=*Y* uvijek se prevodi u sljedeći niz mnemoničkih naredbi:

X := *Y*	MOVE <*AdresaVarijable*Y>, D0 MOVE D0, <*AdresaVarijable*X>

gdje je D0 podatkovni registar. Na sličan način odredi se niz mnemoničkih naredbi za sve tipove troadresnih naredbi. Tablica 1.5 prikazuje generirani mnemonički program za primjer optimiranog međukôda danog u tablici 1.4.

Optimirani troadresni međukôd	*Mnemonički program*

if *Količina*>20 goto L1	MOVE #20, D0
	CMP *Količina*, D0
	BGT L1

goto L2	BRA L2
L1: *Cijena* := 1000	L1: MOVE #1000, D0 MOVE DO, *Cijena*
goto L3	BRA L3
L2: *Cijena* := 1200	L2: MOVE #1200, D0 MOVE DO, *Cijena*
L3: *PosPor* := 8	L3: MOVE #8, D0 MOVE D0, *PosPor*
NovaCijena := *Cijena*	MOVE *Cijena*, D0 MOVE D0, *NovaCijena*

PosPor := PosPor / 2	MOVE *PosPor*, D0
	DIVU #2, D0
	MOVE D0, *PosPor*

Tablica 1.5: Generiranje mnemoničkih naredbi procesora Motorola MC 6800 na temelju troadresnih naredbi međukôda srednje razine

Strojno zavisno optimiranje

Strojno zavisno optimiranje uzima u obzir svojstva arhitekture računala na kojem se izvodi strojni program. Ono je sastavni dio generatora strojnog programa koji se proširuje algoritmima ocjene korisnosti strojnih naredbi, općih i posebnih registara računala, redoslijeda izvođenja strojnih naredbi, itd. Mjerila korisnosti definiraju se na različite načine. Na primjer, ocjena korisnosti strojne naredbe zasniva se na broju procesorskih ciklusa potrebnih za njezino izvođenje. Ocjena korisnosti redoslijeda izvođenja strojnih naredbi uključuje trajanje izvođenja naredbi u procesorskim ciklusima, te iskoristivost procesorskog cjevovoda, aritmetičko-logičkih jedinki i memorijske hijerarhije. Postupci optimiranja preuređuju strojni program na temelju rezultata algoritama koji ocijenuju korisnost naredbi, registara, redoslijeda izvođenja naredbi, itd.

Najjednostavniji, ali ipak učinkovit postupak optimiranja zasnovan je na promatranju tri do pet slijednih strojnih naredbi unutar *prozorčića*. Prozorčić se postavi na početne naredbe strojnog programa, a zatim se miče redom na ostale naredbe sve dok se ne postavi na kraj programa. Unutar prozorčića optimira se redoslijed izvođenja naredbi, odbacuju se prekomjerne naredbe, odbacuju se nedohvatljive naredbe, izvode se algebarska pojednostavljenja, itd. Za izvođenje navedenih jednostavnih postupaka optimiranja nije potrebno analizirati tijek izvođenja programa, toka podatka, itd. Povećavanjem broja prolazaka prozorčića od početka pa do kraja strojnog programa postiže se bolja kvaliteta ciljnog programa.

Tablica 1.6 prikazuje promjene tijekom prva dva prolaska prozorčića, a tablica 1.7 prikazuje mnemonički program nakon sljedeća tri prolaska. Treća kolona te tablice prikazuje optimirani mnemonički program. Budući da nema *podatkovne zavisnosti*, u prvom prolasku prozorčića *zamijene* se *mjesta naredbi* `MOVE #8,D0` i `MOVE D0,PosPor` (prijevod naredbe `PosPor=8;`) s naredbama `MOVE Cijena,D0` i `MOVE D0,NovaCijena` (prijevod naredbe `NovaCijena=Cijena;`). Rezultat zamjene mjesta naredbi prikazan je u drugoj koloni tablice 1.6. Nakon zamjene, naredba `MOVE PosPor,D0` jest *prekomjerna*, jer je podatak o vrijednosti varijable PosPor spremljen u registru `D0` i nije potrebno imati naredbu dohvata vrijednosti varijable *PosPor* iz memorije u registar `D0`. Zato se u drugom prolasku odbacuje naredba `MOVE PosPor,D0`. Treća kolona tablice 1.6 prikazuje mnemonički program nakon odbačene prekomjerne naredbe.

Mnemonički program	*Zamjena mjesta*	*Odbacivanje prekomjerne naredbe*

`MOVE #20,D0`
`CMP Količina,D0`
`BGT L1`
	`MOVE #20,D0`
	`CMP Količina,D0`
	`BGT L1`
		`MOVE #20,D0`
		`CMP Količina,D0`
		`BGT L1`

`BRA L2`	`BRA L2`	`BRA L2`
`L1: MOVE #1000,D0` `MOVE DO,Cijena`	`L1: MOVE #1000,D0` `MOVE DO,Cijena`	`L1: MOVE #1000,D0` `MOVE DO,Cijena`
`BRA L3`	`BRA L3`	`BRA L3`
`L2: MOVE #1200,D0` `MOVE DO,Cijena`	`L2: MOVE #1200,D0` `MOVE DO,Cijena`	`L2: MOVE #1200,D0` `MOVE DO,Cijena`
`L3: MOVE #8,D0` **`MOVE D0,PosPor`**	`L3: MOVE Cijena,D0` `MOVE D0,NovaCijena`	`L3: MOVE Cijena,D0` `MOVE D0,NovaCijena`
`MOVE Cijena,D0` **`MOVE D0,NovaCijena`**	`MOVE #8,D0` **`MOVE D0,PosPor`**	`MOVE #8,D0` **`MOVE D0,PosPor`**

`MOVE PosPor, D0`
`DIVU #2,D0`
`MOVE D0,PosPor`
	`MOVE PosPor,D0`
	`DIVU #2,D0`
	`MOVE D0,PosPor`
		`DIVU #2,D0`
		`MOVE D0,PosPor`

Tablica 1.6: Optimiranje mnemoničkih naredbi procesora Motorola MC 6800 primjenom prozorčića (Prva dva prolaska prozorčića)

Rezultati sljedeća tri prolaska prikazana su u tablici 1.7. Budući da naredbe `DIVU #2,D0` i `MOVE D0,PosPor` spremaju u memoriju novu vrijednost varijable `PosPor` (pogledaj treću kolonu tablice 1.6), vrijednost koju spremaju naredbe `MOVE #8,D0` i `MOVE D0,PosPor` ne koristi nijedna naredba. Naredba `MOVE D0,PosPor` naziva se *mrtva naredba*, jer se rezultat te naredbe ne koristi u daljnjem izvođenju mnemoničkog programa. Postupak odbacivanja mrtve naredbe iz mnemoničkog programa prikazan je u prvoj koloni tablice 1.7. U sljedećem prolasku

operacija dijeljenja `DIVU #2,D0` zamijeni se operacijom posmaka bitova `ROR #1,D0`. Trajanje izvođenja operacije posmaka bitova znatno je kraće od trajanja operacije dijeljenja. Naredbe koje se dulje izvode nazivaju se *skupim operacijama*, a one koje se kraće izvode nazivaju se *jeftinim operacijama*. Zamijenjena jeftina operacija prikazana je u drugoj koloni tablice 1.7.

Odbacivanje mrtve naredbe	*Zamjena skupe operacije jeftinom operacijom*	*Optimirani mnemonički program Algebarsko pojednostavljenje*

`MOVE #20,D0`
`CMP Količina,D0`
`BGT L1`
	`MOVE #20,D0`
	`CMP Količina,D0`
	`BGT L1`
		`MOVE #20,D0`
		`CMP Količina,D0`
		`BGT L1`

`BRA L2`	`BRA L2`	`BRA L2`
`L1: MOVE #1000,D0` `MOVE DO,Cijena`	`L1: MOVE #1000,D0` `MOVE DO,Cijena`	`L1: MOVE #1000,D0` `MOVE DO,Cijena`
`BRA L3`	`BRA L3`	`BRA L3`
`L2: MOVE #1200,D0` `MOVE DO,Cijena`	`L2: MOVE #1200,D0` `MOVE DO,Cijena`	`L2: MOVE #1200,D0` `MOVE DO,Cijena`
`L3: MOVE Cijena,D0` `MOVE D0,NovaCijena`	`L3: MOVE Cijena,D0` `MOVE D0,NovaCijena`	`L3: MOVE Cijena,D0` `MOVE D0,NovaCijena`
`MOVE #8,D0`	**`MOVE #8,D0`**	**`MOVE #4,D0`**
`DIVU #2,D0` `MOVE D0,PosPor`	**`ROR #1,D0`** `MOVE D0,PosPor`	`MOVE D0,PosPor`

Tablica 1.7: Optimiranje mnemoničkih naredbi procesora Motorola MC 6800 primjenom prozorčića (Druga tri prolaska prozorčića)

Posljednji prolazak prozorčića utvrđuje da naredbe `MOVE #8,D0` i `ROR #1,D0` računaju konstantnu vrijednost 4. Te dvije naredbe zamijene se jednom naredbom `MOVE #4,D0` koja izračunatu vrijednost 4 izravno sprema u registar D0. Treća kolona tablice 1.7 prikazuje optimirani mnemonički program.

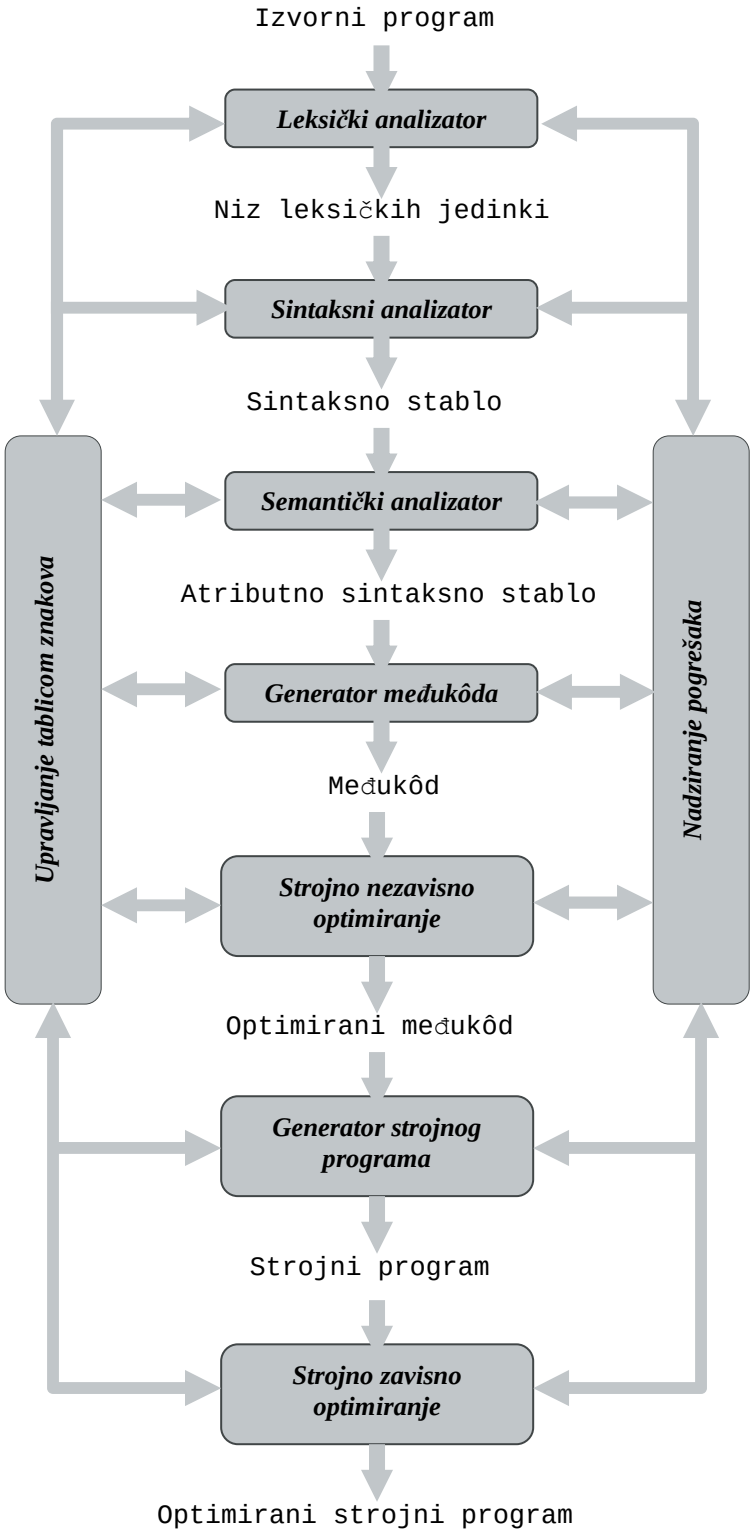
U trećoj koloni tablice 1.7 ima još jedna mogućnost, i to odbacivanje naredbe `MOVE Cijena,D0` koja je označena oznakom `L3`. U prikazanom dijelu programa ta naredba je prekomjerna, jer prethodne naredbe `MOVE #1000,D0` i `MOVE #1200,D0` spremaju podatak o vrijednosti varijable `Cijena` u registar `D0`. Međutim, postupak optimiranja zasnovan na malom prozorčiću *ne* analizira tijek izvođenja programa, te nije u mogućnosti utvrditi sljedeću činjenicu: izvođenju naredbe `MOVE Cijena,D0` neposredno prethodi izvođenje jedne od naredbi `MOVE #1000,D0` i `MOVE #1200,D0`. Zato jednostavni postupak optimiranja zasnovan na prozorčiću ne odbacuje naredbu `MOVE Cijena,D0`. Prekomjerne naredbe u jednostavnom postupku optimiranja zasnovanom na prozorčiću moguće je pronaći isključivo u naredbama koje slijede jedna iza druge.

Upravljanje tablicom znakova i nadziranje pogrešaka

Na slici 12 prikazani su svi koraci rada jezičnog procesora. Ako se rezultat pojedinog koraka rada jezičnog procesora, kao što su to niz leksičkih jedinki, sintaksno stablo ili međukôd, spremi u memoriju računala, onda se rad jezičnog procesora raspodijeli u više cjelina, odnosno kaže se da je jezični procesor višeprolazan. Na primjer, ako se niz leksičkih jedinki i neoptimirani međukôd spreme u datoteku, jezični procesor je troprolazan. U prvom prolasku obavi se leksička analiza, u drugom prolasku obavlja se sintaksna analiza, semantička analiza i generiranje međukôda, a u trećem prolasku obavlja se optimiranje međukôda, generiranje strojnog programa i optimiranje strojnog programa.

Na slici se posebice ističu dva procesa koja imaju značajnu ulogu: upravljanje tablicom znakova i nadziranje pogrešaka. *Tablica znakova* je podatkovna struktura sa zapisima parametara svih leksičkih jedinki koje se nalaze u izvornom programu. Tablice znakova su velike podatkovne strukture koje se dinamički mijenjaju tijekom analize izvornog programa i sinteze ciljnog programa. Potrebno je pažljivo organizirati strukturu tih tablica, kao i algoritme pretraživanja, proširivanja i popunjavanja tih tablica. Učinkovito upravljanje tablicom znakova je od velike važnosti za rad jezičnog procesora.

Otkrivanje pogrešaka u izvornom programu, određivanje mjesta pogreške i ispis odgovarajućih poruka jedan je od osnovnih zadataka jezičnog procesora. *Nadzor nad pogreškama* prisutan je u svim fazama rada jezičnog procesora. Dio nadzora nad pogreškama je i *postupak oporavka od pogreške*. Nakon uočene pogreške, postupak oporavka od pogreške omogućava daljnju analizu izvornog programa. U tom slučaju analiza se ne zaustavlja na prvoj pronađenoj pogrešci u izvornom programu, već se ona nastavlja s ciljem pronalaženja ostalih pogrešaka. Postupak oporavka od pogreške je težak problem koji se zasebno rješava za svaku fazu rada jezičnog procesora.



Slika 1.12: Koraci rada jezičnog procesora