

# Sadržaj

<b>Predgovor</b>	<b>1</b>
<b>1 Uvod</b>	<b>3</b>
1.1 Vrste programskih prevodilaca . . . . .	3
1.2 Struktura kompilatora . . . . .	5
1.3 Hibridni prevodioci . . . . .	7
1.4 Prenosivost koda . . . . .	7
1.5 Višejezički kompilator . . . . .	9
1.6 Proces prevođenja programa . . . . .	9
1.7 Kratak osvrt na istoriju kompilatora . . . . .	13
1.8 Pitanja . . . . .	14
<b>2 Elementi teorije formalnih jezika</b>	<b>15</b>
2.1 Opis jezika . . . . .	15
2.2 Elementi jezika . . . . .	16
2.2.1 Azbuka . . . . .	16
2.2.2 Reč . . . . .	16
2.2.3 Proizvod reči . . . . .	17
2.2.4 Eksponent reči . . . . .	17
2.2.5 Transponovana reč . . . . .	17
2.2.6 Delovi reči . . . . .	18
2.3 Formalni jezik . . . . .	18
2.3.1 Operacije nad jezicima . . . . .	19
2.4 Opis jezika . . . . .	21
2.4.1 Izvođenje reči . . . . .	21
2.4.2 Elementi gramatike . . . . .	23
2.4.3 Direkto izvođenje i redukcija . . . . .	23
2.5 Formalne gramatike . . . . .	24
2.6 Hierarhija gramatika po Čomskom . . . . .	27
2.6.1 Gramatike tipa jedan ili konteksne gramatike . . . . .	27
2.6.2 Gramatike tipa dva ili beskonteksne gramatike . . . . .	28
2.6.3 Gramatike tipa tri ili regularne gramatike . . . . .	29
2.6.4 Rečenične forme . . . . .	30
2.6.5 Fraze . . . . .	31

2.7	Normalne forme gramatika . . . . .	31
2.7.1	Normalne forme za kontekstne gramatike . . . . .	31
2.7.2	Normalne forme za beskontekstne gramatike . . . . .	32
2.8	Pitanja . . . . .	32
2.9	Zadaci . . . . .	32
<b>3</b>	<b>Automati kao uređaji za prepoznavanje jezika</b>	<b>35</b>
3.1	Opšti model automata . . . . .	35
3.2	Tjuringova mašina . . . . .	37
3.3	Linearno ograničen automat . . . . .	38
3.4	Magacinski automat . . . . .	39
3.5	Konačni automati . . . . .	42
3.5.1	Određivanje jezika koji se prepoznaje datim konačnim au- tomatom . . . . .	44
3.5.2	Veza između konačnih automata i gramatika tipa tri . . . .	46
3.6	Regularni izrazi . . . . .	49
3.7	Preslikavanje regularnih izraza u konačne automate . . . . .	52
3.8	Pitanja . . . . .	55
3.9	Zadaci . . . . .	55
<b>4</b>	<b>Leksički analizator</b>	<b>59</b>
4.1	Namena leksičkog analizatora . . . . .	59
4.1.1	Tokeni . . . . .	61
4.1.2	Šabloni . . . . .	62
4.1.3	Tabela simbola . . . . .	63
4.2	Realizacija leksičkog analizatora . . . . .	64
4.3	Programska realizacija leksičkog analizatora . . . . .	67
4.3.1	Tehnika dvostrukog bafera . . . . .	68
4.4	Pitanja . . . . .	70
4.5	Zadaci . . . . .	70
<b>5</b>	<b>Sintaksna analiza</b>	<b>73</b>
5.1	Redosled primene pravila . . . . .	73
5.2	Osnovni algoritam za <i>top-down</i> sintaksnu analizu . . . . .	79
5.2.1	Problem leve rekurzije . . . . .	81
5.2.2	Eliminisanje leve rekurzije . . . . .	83
5.3	Osnovni algoritam za <i>bottom-up</i> analizu . . . . .	86
5.4	Pitanja . . . . .	90
5.5	Zadaci . . . . .	91
<b>6</b>	<b>Prediktivna analiza – LL-1 analizatori</b>	<b>93</b>
6.1	Proste LL-1 gramatike . . . . .	94
6.1.1	Leva faktORIZACIJA . . . . .	97
6.2	Pomoćne funkcije za definisanje opštih LL-1 gramatika . . . .	97
6.2.1	Funkcija First . . . . .	98
6.2.2	Funkcija Follow . . . . .	99

6.3	LL-1 gramatike bez $\epsilon$ -pravila . . . . .	100
6.4	LL-1 gramatike sa $\epsilon$ -pravilima . . . . .	103
6.5	Rešeni zadaci . . . . .	106
6.6	Pitanja . . . . .	107
6.7	Zadaci . . . . .	108
<b>7</b>	<b>Analiza zasnovana na relacijama prvenstva</b>	<b>111</b>
7.1	Relacije prvenstva . . . . .	111
7.1.1	Relacija istog prioriteta . . . . .	111
7.1.2	Relacija nižeg prioriteta . . . . .	112
7.1.3	Relacija višeg prioriteta . . . . .	112
7.2	Operatorska gramatika prvenstva . . . . .	113
7.2.1	Postupak sintaksne analize . . . . .	119
7.2.2	Funkcije prvenstva . . . . .	121
7.3	Opšta pravila prioriteta . . . . .	122
7.4	Gramatike prvenstva . . . . .	124
7.4.1	Postupak sintaksne analize . . . . .	126
7.5	Pitanja . . . . .	132
7.6	Zadaci . . . . .	133
<b>8</b>	<b>LR analizatori</b>	<b>135</b>
8.1	Opis analizatora . . . . .	135
8.1.1	<i>Shift-reduce</i> algoritam analize . . . . .	137
8.2	Realizacija LR analizatora . . . . .	139
8.2.1	Vidljivi prefiksi . . . . .	139
8.2.2	LR( $k$ ) gramatike . . . . .	140
8.2.3	Tipovi LR analizatora . . . . .	140
8.3	Realizacija SLR analizatora . . . . .	141
8.3.1	LR(0) članovi . . . . .	141
8.3.2	Funkcija zatvaranja (closure) . . . . .	142
8.3.3	Funkcija iniciranja novih skupova LR(0) članova (goto) . . . . .	142
8.3.4	Postupak sinteze LR analizatorata . . . . .	142
8.4	Rešeni zadaci . . . . .	147
8.5	Pitanja . . . . .	155
8.6	Zadaci . . . . .	156
<b>9</b>	<b>Tabele simbola</b>	<b>159</b>
9.1	Predstavljanje simbola u tabeli simbola . . . . .	160
9.2	Predstavljanje simbola korišćenjem <i>flat</i> pristupa . . . . .	160
9.3	Predstavljanje simbola korišćenjem objektno-orijentisanog pristupa . . . . .	163
9.4	Predstavljanje tabele simbola . . . . .	163
9.5	Tabele simbola i oblast važenja simboličkih imena . . . . .	167
9.6	Pitanja . . . . .	173
9.7	Zadaci . . . . .	173
<b>10</b>	<b>Generatori leksičkih i sintaksnih analizatora</b>	<b>175</b>

10.1	Lex – generator leksičkih analizatora . . . . .	175
10.1.1	Pravila u Lex specifikaciji . . . . .	176
10.1.2	Definicije u Lex specifikaciji . . . . .	179
10.1.3	Specijalna početna stanja . . . . .	179
10.1.4	Definicije makroa . . . . .	180
10.1.5	Korisnički potprogrami . . . . .	181
10.1.6	Upotreba Lex generatora . . . . .	181
10.1.7	Primer kompletne Lex specifikacije . . . . .	182
10.2	Yacc – generator sintakasnih analizatora . . . . .	186
10.2.1	Definicije u Yacc specifikaciji . . . . .	187
10.2.2	Definicija startnog simbola gramatike . . . . .	187
10.2.3	Definicije terminalnih simbola gramatike . . . . .	187
10.2.4	Definicije prioriteta i asocijativnosti operatora . . . . .	188
10.2.5	Definicije tipova atributa terminalnih i neterminalnih simbola . . . . .	188
10.2.6	Pravila u Yacc specifikaciji . . . . .	189
10.2.7	Korisnički potprogrami . . . . .	190
10.2.8	Upotreba Yacc generatora . . . . .	191
10.2.9	Oporavak od grešaka . . . . .	192
10.2.10	Generisanje interpretatora pomoću Yacc-a . . . . .	196
10.3	JFlex . . . . .	200
10.3.1	Opcije kojima se podešavaju parametri generisanog leksičkog analizatora . . . . .	201
10.3.2	Deklaracije posebnih početnih stanja . . . . .	203
10.3.3	Definicije makroa . . . . .	203
10.3.4	Pravila . . . . .	203
10.3.5	Korišćenje JFlex-a . . . . .	203
10.3.6	Primer JFlex specifikacije . . . . .	204
10.4	Cup – generator sintakasnih analizatora . . . . .	207
10.4.1	Cup specifikacija . . . . .	209
10.4.2	Uvođenje referenci na attribute simbola u smenama . . . . .	211
10.4.3	Prevođenje Cup specifikacije . . . . .	212
10.4.4	Korišćenje generisanog analizatora . . . . .	213
10.5	Pitanja . . . . .	214
10.6	Zadaci . . . . .	215
<b>11</b>	<b>Atributne gramatike</b>	<b>217</b>
11.1	Sintaksno upravljane definicije . . . . .	217
11.2	Generisani atributi . . . . .	218
11.3	Nasleđeni atributi . . . . .	219
11.4	Provera tipova podataka ( <i>Type Checking</i> ) . . . . .	225
11.4.1	Semantička pravila za proveru tipova podataka . . . . .	225
11.5	Pitanja . . . . .	228
11.6	Zadaci . . . . .	229
<b>12</b>	<b>Međukodovi</b>	<b>231</b>

12.1	Mesto međukoda u procesu prevodenja . . . . .	231
12.2	Apstraktno sintaksno stablo . . . . .	232
12.3	Implementacija apstraktnog sintaksnog stabla . . . . .	235
12.4	Troadresni međukod . . . . .	235
12.4.1	<i>Quadruples</i> struktura . . . . .	238
12.4.2	<i>Triples</i> struktura . . . . .	238
12.4.3	<i>Indirect triples</i> struktura . . . . .	240
12.5	Troadresni međukod za naredbe dodeljivanja . . . . .	241
12.6	Troadresni međukod upravljačkih naredbi . . . . .	245
12.7	Pitanja . . . . .	252
12.8	Zadaci . . . . .	253
<b>13</b>	<b>Optimizacija koda</b>	<b>255</b>
13.1	Osnovna optimizacija koda ( <i>peephole</i> optimizacija) . . . . .	255
13.1.1	Eliminisanje redundantnih naredbi . . . . .	256
13.1.2	Optimizacija toka . . . . .	257
13.1.3	Nedosegljiv kod . . . . .	258
13.1.4	Algebarska uprošćenja . . . . .	259
13.1.5	Direktna redukcija . . . . .	259
13.1.6	Korišćenje mašinskih idioma . . . . .	259
13.2	Dodatna optimizacija . . . . .	260
13.2.1	Zajednički podizrazi . . . . .	263
13.2.2	Zajednički podizrazi na globalnom nivou . . . . .	264
13.2.3	Prostiranje naredbi za kopiranje . . . . .	264
13.2.4	Eliminisanje neaktivnog koda . . . . .	265
13.2.5	Optimizacija petlji . . . . .	265
13.3	Pitanja . . . . .	267
<b>14</b>	<b>Generisanje koda</b>	<b>271</b>
14.1	Odredišna mašina – procesor Intel 8086 . . . . .	272
14.1.1	Osnovni skup instrukcija . . . . .	273
14.1.2	Načini adresiranja . . . . .	274
14.2	Upravljanje memorijom u toku izvršenja programa . . . . .	276
14.2.1	Organizacije memorije . . . . .	276
14.2.2	Aktivacioni slogovi . . . . .	277
14.3	Statička alokacija memorije . . . . .	279
14.4	Alokacija memorije pomoću steka . . . . .	280
14.5	Dinamička alokacija memorije . . . . .	281
14.5.1	Generisanje koda za poziv potprograma . . . . .	281
14.5.2	Generisanje sekvence poziva . . . . .	285
14.5.3	Generisanje sekvence povratka . . . . .	286
14.5.4	Smeštanje promenljivih i pristup promenljivama u razli- čitim delovima memorije . . . . .	289
14.6	Realizacija generatora koda . . . . .	291
14.6.1	Algoritam generisanja koda . . . . .	291
14.6.2	Funkcija getreg . . . . .	292

14.6.3	Generisanje koda za neke specijalne naredbe . . . . .	293
14.6.4	Generisanje koda za naredbe sa indeksnim promenljivama	293
14.6.5	Generisanje koda za naredbe sa pokazivačima . . . . .	293
14.6.6	Generisanje koda za naredbe uslovnog skoka . . . . .	294
14.7	Pitanja . . . . .	297
14.8	Zadaci . . . . .	298
<b>15</b>	<b>Realizacija kompilatora</b>	<b>299</b>
15.1	Tehnika butstrepanja . . . . .	299
15.1.1	Kompilator za novi programski jezik . . . . .	301
15.1.2	Prenošenje kompilatora sa jedne na drugu mašinu . . . .	301
15.1.3	Optimizacija postojećeg kompilatora . . . . .	302
15.1.4	Razvoj kompilatora za programski jezik Pascal . . . . .	302
15.1.5	Razvoj kompilatora za programski jezik FORTRAN . . .	303
15.2	Struktura kompilatora . . . . .	308
15.2.1	C kompilator za mašinu PDP 11 . . . . .	308
15.2.2	FORTTRAN H kompilator . . . . .	308
15.3	Pitanja . . . . .	309
<b>16</b>	<b>Dodaci</b>	<b>311</b>
16.1	Java bajtkod . . . . .	311

# Predgovor

*Sadržaj ove knjige definisan je u skladu sa programom predmeta Programski prevodioci koji je već čitav niz godina obuhvaćen studijskim programom Računarstvo i informatika na Elektronskom fakultetu u Nišu. Autori su profesori koji su do sada učestvovali u realizaciji nastave iz ovog predmeta: prof. Živko Tošić koji je postavio osnove ovog predmeta i uvrstio ga u studijski program Računarstvo i informatika, prof. Milena Stanković koja je dugo godina bila nastavnik iz ovog predmeta i nekoliko puta prilagođavala njegov sadržaj zahtevima akreditacija koje su vršene, kao i prof. Suzana Stojković koja sada učestvuje u realizaciji nastave iz ovog predmeta, a dugi niz godina je bila zadužena i za realizaciju računskih i laboratorijskih vežbi.*

*Sadržaj udžbenika obuhvata klasične teme iz oblasti programskih prevodilaca kao što su formalni jezici i gramatike u funkciji opisa sintakse programskih jezika i realizacija leksičkog i sintaksnog analizatora, ali i novije teme koje se odnose na semantičku analizu kao deo procesa prevodenja programa. Takođe, obuhvaćeno je i korišćenje softverskih alata za generisanje leksičkih i sintaksnih analizatora.*

*Sva poglavlja udžbenika sadrže niz primera koji su kompletno urađeni, a dati sa ciljem da se olakša razumevanje razmatranih algoritama. Poglavlja se završavaju pitanjima i zadacima koji se odnose na sadržaj poglavlja, a data su sa ciljem da se studentima olakša priprema pisanog i usmenog dela ispita.*

*Autori posebno žele da zahvale recenzentima dr Zoranu Budimcu, redovnom profesoru na Departmanu za matematiku i informatiku Prirodno matematičkog fakulteta u Novom Sadu, i dr Svetozaru Rančiću, docentu na Departmanu za računarske nauke Prirodno matematičkog fakulteta u Nišu, na uloženom trudu da pročitaju i ocene rukopis udžbenika, kao i na korisnim predlozima i primedbama.*

*Takođe, veliku zahvalnost dugujemo studentu Lazaru Ljubenoviću, koji je kompletan tekst knjige dobrovoljno prebacio iz Word-a u L<sup>A</sup>T<sub>E</sub>X i svojom upornošću znatno doprineo tome da mukotrpan posao konačnog uobličavanja teksta bude doveden do kraja.*

*Takođe, iskreno se zahvaljujemo i svim studentima sa smjera za Računarsku tehniku i informatiku Elektronskog fakulteta u Nišu, koji su tokom korišćenja delova udžbenika u predštampi ukazali na razne greške koje su pronašli i time*

*doprineli da se znatno smanji broj grešaka ostao u konačnom tekstu.*

*Izvinjavamo se svim budućim korisnicima udžbenika ako tokom njegovog korićenja naiđu na štamparske greške, kojih verovatno još uvek ima i pored nastojanja da budu svedene na minimum. Bićemo zahvalni za sve ukazane greške predloge i primedbe od korisnika ovog udžbenika.*

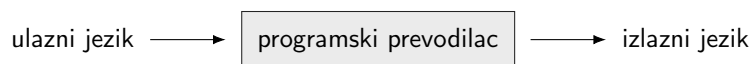
U Nišu, april 2017.



# Glava 1

## Uvod

Uvek kada softversku aplikaciju razvijamo nekim višim programskim jezikom treba nam softver koji će taj program da prevedu u mašinski jezik, zato što svi računari, ma koliko složeni bili, obrađuju instrukcije predstavljene u mašinskom (binarnom) obliku. Zadatak tog softvera je utoliko složeniji koliko je jezik sa kojeg se prevodi visokog nivoa. Programski prevodioci su softverske komponente koje generalno prevode programe pisane u jednom programskom jeziku u odgovarajuće programe pisane drugim programskim jezikom (slika 1.1).



SLIKA 1.1: Programski prevodilac

Obično je jezik sa kog se prevodi višeg nivoa od jezika na koji se vrši prevođenje, mada, u opštem slučaju, možemo da prevodimo sa jednog jezika visokog nivoa na drugi jezik visokog nivoa, ali i da se kod niskog nivoa prevodi u kod višeg nivoa (tako npr. rade krosasembleri).

### 1.1 Vrste programskih prevodilaca

Programski prevodioci su se razvijali paralelno sa programskim jezicima i u velikoj meri su uslovlili razvoj programskih jezika, ali i razvoj računara generalno.

Tabela 1.1 daje klasifikaciju programskih jezika prema njihovom nivou i nekoj hronologiji razvoja.

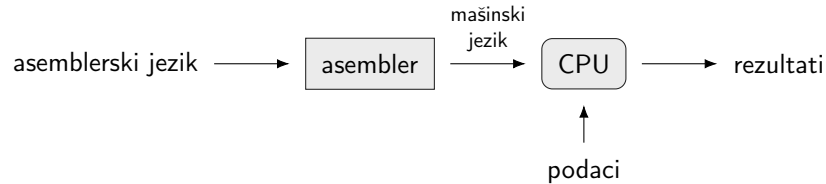
Mašinski jezik je programski jezik najnižeg nivoa i podrazumeva da se program sastoji od naredbi i podataka napisanih u binarnom obliku (preko nizova jedinica i nula) uz korišćenje apsolutnih adresa podataka. Na ovom nivou programski

TABELA 1.1: Klasifikacija programskih jezika

MAŠINSKI ZAVISNI		MAŠINSKI NEZAVISNI	
I generacija	II generacija	III generacija	IV generacija
mašinski jezik	MAŠINI ORIJENTISANI JEZICI		
<i>jezici niskog nivoa</i> →	asemblerski jezici	makroasemblerski jezici	
		<i>jezici visokog nivoa</i> →	proceduralni jezici
		<i>jezici veoma visokog nivoa</i> →	problemski jezici

prevodilac nije potreban – naredbe se upisuju direktno u operativnu memoriju i izvršavaju.

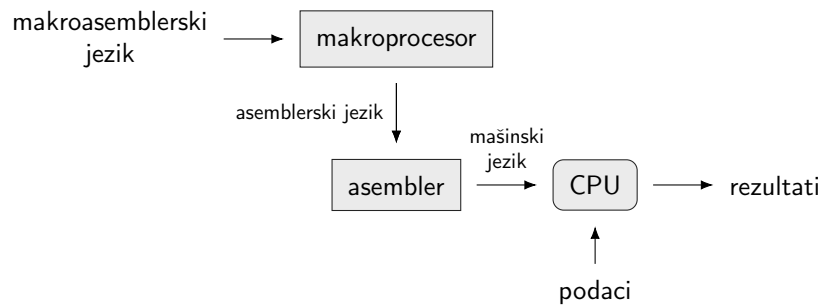
Drugu generaciju jezika čine asemblerski jezici kod kojih se mašinske naredbe predstavljaju simbolički, uz korišćenje simboličkih adresa naredbi i podataka. Ovi jezici su višeg nivoa od mašinskih ali su i dalje mašinski zavisni. Naime, asemblerski jezik se definiše zajedno sa procesorom računara tako da struktura procesora bitno utiče na strukturu asemblerskog jezika i obrnuto. Za prevođenje zapisa sa nivoa asemblerskog jezika na mašinski nivo potreban je programski prevodilac koji se u ovom slučaju naziva assembler (*assembler*, slika 1.2) odakle dolazi i naziv ovih jezika.



SLIKA 1.2: Assembler

Makroasemblerski jezici su nešto višeg nivoa od asemblerskih jezika i omogućavaju skraćeno zapisivanje programa tako što se grupa asemblerskih naredbi predstavi jednom makro-naredbom. Kod programa napisanih na ovom nivou, najpre se koristi makroprocesor čiji je zadatak da pozive makro-naredbi zameni odgovarajućim sekvencama asemblerskih naredbi. Nakon toga se dobijeni asemblerski kod prevodi u mašinski pomoću odgovarajućeg assemblera (slika 1.3).

Kako je asemblerski jezik niskog nivoa, a posebno zbog činjenice da svakoj asemblerskoj naredbi odgovara jedna mašinska naredba, realizacija assemblera nije komplikovan zadatak. Slično je i sa makroprocesorom, čiji je zadatak prvenstveno da pokupi i zapamti definicije makro-naredbi i da zameni pozive makro-naredbi odgovarajućim sekvencama asemblerskih naredbi.



SLIKA 1.3: Makroprocesor

U slučaju viših programskih jezika proces prevođenja na mašinski nivo je složeniji. U te svrhe se kao programski prevodioci koriste kompilatori (slika 1.4) i interpretatori (slika 1.5). Osnovna razlika između kompilatora i interpretatora je u tome što kompilator najpre vrši analizu programa na osnovu koje generiše mašinski kod celog programa. Interpretatori koriste drugu strategiju: prevode naredbu po naredbu programa i, čim formiraju celinu koja može da se izvrši, izvršavaju je.

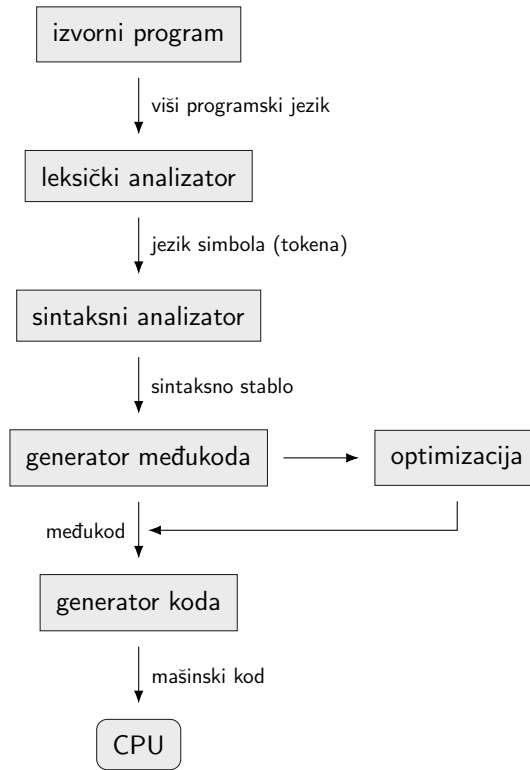
Koncept kompiliranja programa se primenjuje kada se razvijaju aplikacije koje se kasnije koriste kao gotovi proizvodi dok je koncept interpretiranja pogodniji u slučaju interaktivne komunikacije sa računarom.

U okviru ovog predmeta razmatraćemo uglavnom probleme vezane za realizaciju kompilatora.

## 1.2 Struktura kompilatora

Kompilator započinje analizu programa fazom leksičke analize. U ovoj fazi identifikuju se leksičke celine i nastoji se da se uprosti zapis programa kako bi se pripremio za sintaksnu analizu. Na primer, za proces sintaksne analize nije važno da li je neka promenljiva u programu nazvana *A* ili *B* već je važno da se na određenom mestu javlja poromenljiva. Imena promenljivih kao i imena drugih elemenata programa su bitna u narednim koracima prevođenja, zato se ta imena, kao i mnoge druge informacije o entitetima koje imenuju, pamte u posebnoj strukturi koja se naziva tabela simbola (*symbol table*).

Nakon leksičke analize sledi faza sintaksne analize. Zadatak ove faze je da se utvrdi da li je program napisan u skladu sa pravilima kojima je definisan programski jezik koji je korišćen. Danas se koriste formalni opisi programskih jezika preko određenih meta-jezika. U suštini, zadatak sintaksnog analizatora je da raščlani složene naredbe jezika na elementarne. Kao rezultat sintaksne analize dobija se sintakšno stablo koje pokazuje koja pravila i kojim redosledom treba primeniti da bi se generisala određena naredba jezika.

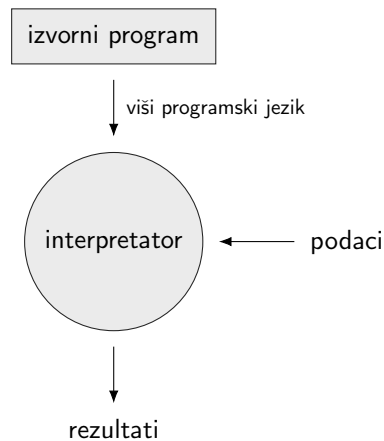


SLIKA 1.4: Kompilator

Na osnovu sintaksnog stabla generiše se međukod programa. Međukod je reprezentacija programa koja je mašinski nezavisna ali se jednostavno preslikava u mašinski ili asemblerski kod za određenu mašinu.

Kod koji kompilator generiše u prvom prolazu obično sadrži puno redundantnosti pa se zato vrši optimizacija koda. Ova optimizacija može da bude mašinski nezavisna, kada se vrši na nivou međukoda ali i mašinski zavisna, kada se vrši na nivou generisanog koda.

Savremeni kompilatori pored sintaksne vrše i semantičku analizu koja se često integriše sa sintaksnom analizom ili se izvršava kao nezavisna faza. Zadatak semantičke analize je da utvrdi da li su u pisanju programa ispoštovana dodatna, semantička pravila koja se ne mogu formalno definisati kroz formalni opis jezika. Takva pravila su recimo pravila vezana za koncept jakih tipova podataka ili pravila o tome kojim stvarnim parametrima mogu da se zamenjuju fiktivni parametri i sl. Ova dodatna pravila se obično definišu kao procedure koje se izvršavaju prilikom primene određenog sintaksnog pravila.



SLIKA 1.5: Interpretator

### 1.3 Hibridni prevodioci

Pored čistih kompilatora i interpretatora danas se koriste i hibridni prevodioci (slika 1.6) koji predstavljaju neku vrstu mešavine kompilatora i interpretatora, vrše analizu koda kao kompilatori, generišu međukod programa i nakon toga interpretiraju taj međukod.

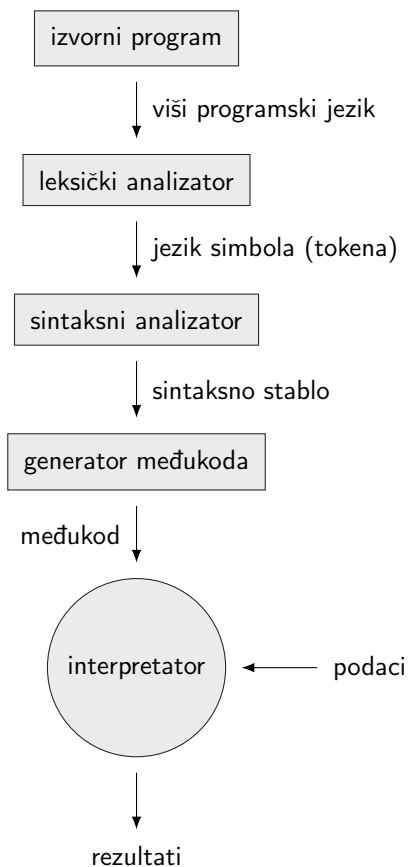
### 1.4 Prenosivost koda

Koncept korišćenja međukoda kod kompilatora ima niz prednosti. Kako je međukod mašinski nezavisan, jednostavno se može preslikati u bilo koji drugi mašinski jezik. To znači da se na osnovu jednom generisanog međukoda može generisati mašinski kod za različite mašine i ostvariti prenosivost koda.

Ovaj koncept se na primer standardno koristi kod programskog jezika Java, kod kojeg se najpre generiše međukod programa (*bytecode*) a zatim taj međukod interpretirana na Java virtuelnoj mašini kao interpretatoru. Primena ovog koncepta omogućila je laku prenosivost Java koda sa jedne mašine na drugu, što je i bila osnovna ideja kod projektovanja ovog jezika. Naime, programski jezik Java je projektovan tako da se može koristiti na različitim računarskim platformama, a pre svega kod ugrađenih (*embedded*) računarskih sistema.

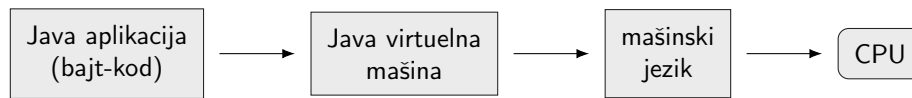
Danas se programski jezik Java koristi na više različitih načina):

1. Prvi slučaj je, već pomenuti način, kada se Java program prevodi u međukod i interpretira na Java virtuelnoj mašini koja generiše mašinske naredbe koje se izvršavaju u procesoru računara (slika 1.7a).

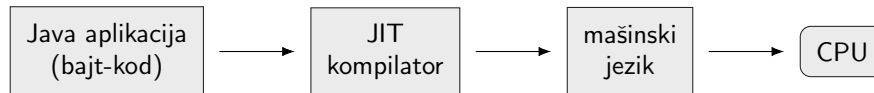


SLIKA 1.6: Hibridni prevodilac

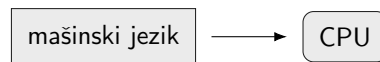
2. U drugom slučaju koristi se tzv. *just in time* Java kompilator koji bajtkod programa preslikava u mašinski kod namenjen određenoj mašini, po principu po kome rade i drugi kompilatori (slika 1.7b).
3. U trećem slučaju Java kod se ne preslikava u standardni međukod već se direktno generiše mašinski kod za procesor na kome će se izvršavati (slika 1.7c).
4. U poslednjem slučaju interpretator Java međukoda je realizovan kao poseban Java čip koji interpretira naredbe međukoda kao što standardni procesor izvršava mašinske naredbe (slika 1.7d).



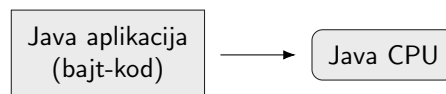
(A) Java virtuelna mašina (interpretiranje – sporo): prevodi instrukcije u mašinski jezik po redosledu pojavljivanja.



(B) Java *just-in-time* kompilator (brži): konveruje sve instrukcije u mašonski jezik kad je instrukcija potrebna (dinamičko prevođenje), a zatim izvršava mašinski jezik.



(C) Java kompilator (brži): izvorni kod se prevodi u mašinski kod i izvrđava kao C ili C++ program.



(D) Java procesor (najbrži): Java procesor izvršava Java bajt-kod na način na koji drugi CPU izvršavaju odgovarajući mašinski kod.

SLIKA 1.7

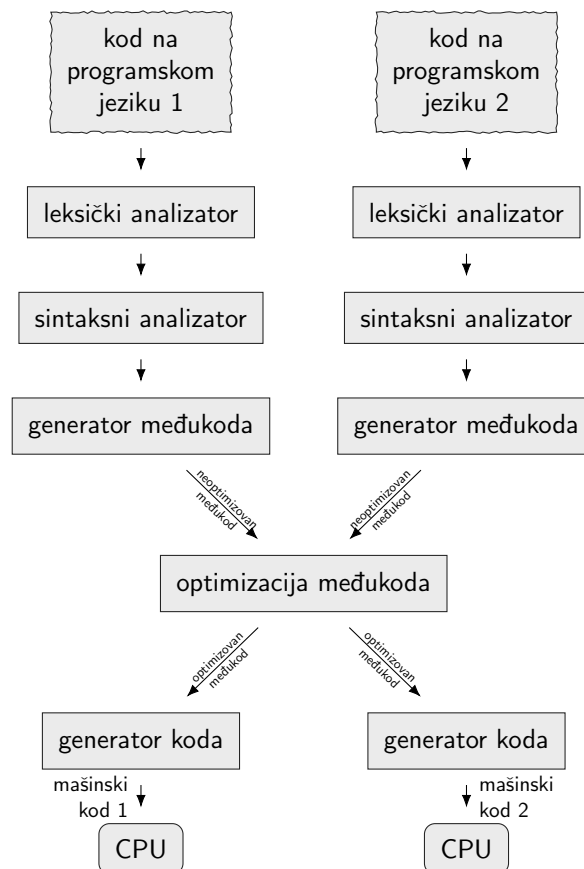
## 1.5 Višejezički kompilator

Korišćenje međukoda, takođe, omogućava realizaciju višejezičnih kompilatora (slika 1.8). To su kompilatori koji imaju posebne delove za analizu koda za više različitih programskih jezika, pri čemu se kod svih ovih analizatora generiše isti međukod programa. Posle toga, taj međukod može da se prevodi u mašinski jezik određenog računara, jednog ili više različitih procesora.

## 1.6 Proces prevođenja programa

U nastavku ovog poglavlja, kako bi donekle stekli utisak o radu kompilatora, pratićemo jednu naredbu napisanu na hipotetičkom višem programskom jeziku kroz sve faze kompilatora.

Kao primer uzećemo naredbu dodeljivanja:



SLIKA 1.8: Višejezički kompilatori

```
pos := init + r * 60.0
```

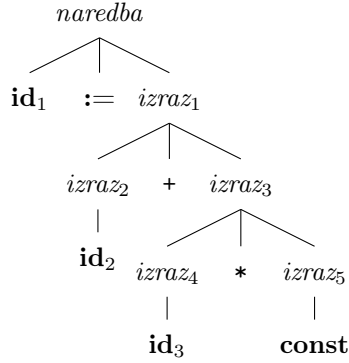
**Leksička analiza.** Leksički analizator analizira znak po znak ove naredbe i nastoji da identifikuje leksičke celine (lekseme), pri čemu svaku leksemu zamenjuje odgovarajućim simbolom (tokenom). U našem primeru rezultat analize biće sledeća sekvenca:

```
id1 := id2 + id3 * const
```

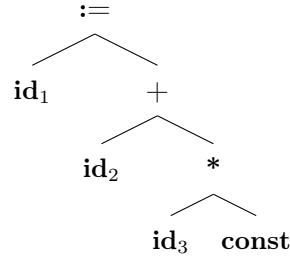
Leksički analizator je prepoznao identifikatore `pos`, `init` i `r` i zamenio ih simbolom `id`. Takođe su prepoznati simboli `:=`, `+` i `*`, koji ostaju nepromenjeni, kao i konstanta `60.0` koja je zamenjena tokenom `const`.

Leksički analizator takođe generiše tabelu simbola, kao posebnu strukturu u kojoj čuva podatke o elementima programa koji su prepoznati. Na primer, u sledećoj tabeli simbola se čuvaju imena identifikatora za naš primer.





SLIKA 1.9: Sintaksko stablo za naredbu  
pos := int + r \* 60.0



SLIKA 1.10: Apstraktno sintaksko stablo  
za naredbu pos := int + r \* 60.0

IME	ATRIBUT <sub>1</sub> VRSTA ENTITETA	ATRIBUT <sub>2</sub> TIP	ATRIBUT <sub>3</sub> ...
1 pos	var	real	...
2 init	var	real	...
3 r	var	real	...

**Sintakсна analiza.** Zadatak sintaksnog analizatora je da utvrdi da li je data naredba napisana u skladu sa formalnim opisom jezika. U te svrhe sintakсни analizator koristi skup pravila kojima se opisuje jezik. U našem primeru to bi bio sledeći skup pravila:

1. Svaki identifikator je izraz.
2. Svaka konstanta je izraz. Ako su  $izraz_1$  i  $izraz_2$  izrazi, tada je su izrazi i:
  - (a)  $izraz_1 + izraz_2$
  - (b)  $izraz_1 * izraz_2$
3. Ako je  $id_1$  identifikator i ako je  $izraz_1$  izraz, tada je  $id_1 := izraz_1$  naredba.

Postupkom sintaksne analize određuje se skup i redosled pravila koja treba primeniti da bi se generisala zadata reč. Redosled primene pravila se može predstaviti sintaksnim stablom (slika 1.9 za razmatrani primer). Na osnovu sintaksnog stabla generiše se apstraktno sintakšno stablo (AST – *abstract syntax tree*) u kome se u čvorovima nalaze operatori a grane vode do operanada (slika 1.10). Ovo stablo predstavlja osnovu za generisanje ciljnog koda.

**Semantička analiza.** Sintakсни analizator može da bude obogaćen dodatnim semantičkim procedurama kojima se proveravaju neka dodatna semantička pravila kojima je dopunjen opis jezika. U našem slučaju, semantički analizator

može da proverava da li su u naredbi dodeljivanja ispoštovana pravila jakih tipova podataka. U te svrhe analizator može da koristi apstraktno sintaksno stablo obogaćeno atributima. Ovakvo stablo se naziva označeno sintaksno stablo (*annotated syntax tree*); slika 1.11 za naš primer. Drugi način za pamćenje dodatnih atributa simboličkih imena je tabela simbola.

**Generisanje međukoda.** Na osnovu apstraktnog sintaksnog stabla generiše se međukod programa. U slučaju razmatrane naredbe dodeljivanja, može da bude generisana sledeća sekvenca naredbi hipotetičkog asemblerskog jezika.

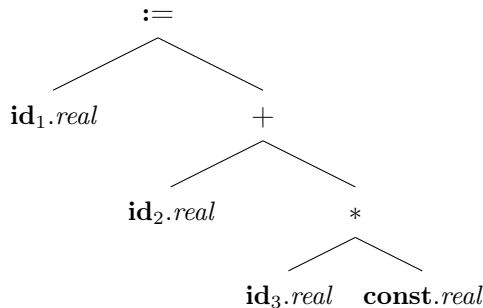
```
t1 := real(60)
t2 := r * t1
t3 := init + t2
pos := t3
```

Promenljive `t1`, `t2` i `t3` su takozvane privremene promenljive, odnosno promenljive koje generiše kompilator za smeštaj međurezultata.

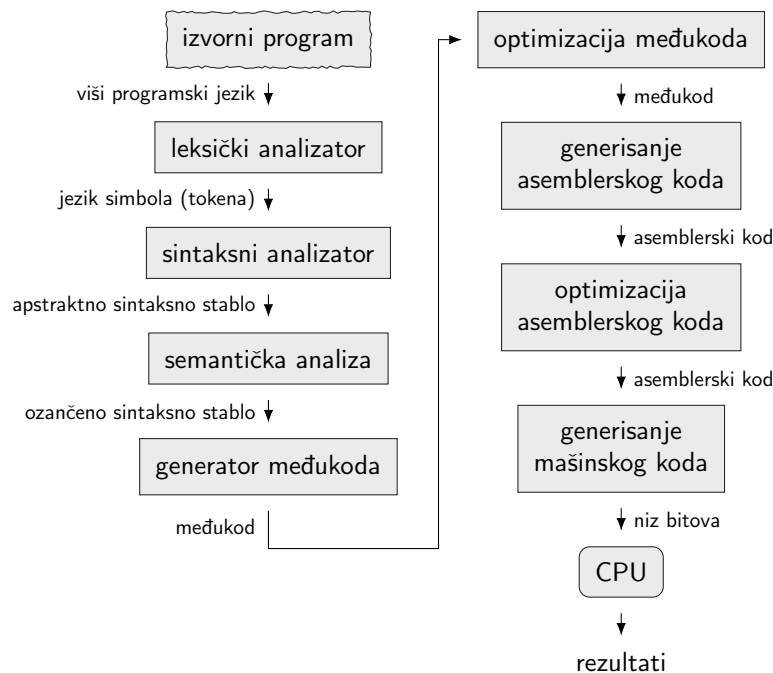
**Generisanje koda.** Na osnovu međukoda generiše se ciljni kod programa; to može da bude zapis u asemblerskom jeziku ciljne mašine ili direktno mašinski jezik. Asemblerski kod za posmatrani primer bi bio:

```
MOV    r, R2
MULV   #60.0, R2
MOV    init, R1
ADD     R2, R1
MOV     R1, pos
```

Pored navedenih koraka, relani kompilatori obično uključuju i optimizaciju koda kako na nivou međukoda tako i na nivou generisanog asemblerskog ili mašinskog koda. Slika 1.12 prikazuje potpunu strukturu kompilatora.



SLIKA 1.11: Označeno sintaksno stablo za naredbu `pos := int + r * 60.0`



SLIKA 1.12: Potpuna struktura kompilatora

## 1.7 Kratak osvrt na istoriju kompilatora

Reč kompilator (*compiler*) je prvi put upotrebila Grace Murray Hopper. Originalno značenje glagola kompilirati (*to compile*) je izvršiti odabir reprezentativnog materijala i dodati ga kolekciji. Na taj način se danas, na primer, taj glagol pravilno koristi kada kažemo kompilacija kompaktnih diskova. U ranoj fazi razvoja programskih jezika i kompilatora, prevođenje programskih jezika je viđeno na isti način: kada se u ulaznom kodu nalazi npr.  $A + b$ , onda se bira `LOAD a, R1` i `ADD b, R1` i dodaje na prethodno generisanu sekvencu naredbi. U toj fazi kompilator je prevodio deo po deo programa i dodavao ga kodu. Današnji kompilatori, posebno oni za savremene objektno-orijentisane jezike, su mnogo složeniji programi koji obavljaju mnogo složenije transformacije ulaznog koda. Takođe uključuju i niz specifičnih semantičkih analiza ili automatsku paralelizaciju programa.

Sve do kraja 1950. godine, programiralo se korišćenjem mašinski zavisnih jezika, dok su mašinski nezavisni jezici samo naučno razmatrani i predlagani. U to vreme kapaciteti računara su bili ograničeni i sami programi kratki. Međutim, počinju da se javljaju ideje o korišćenju viših programskih jezika i razvijeno je nekoliko eksperimentalnih kompilatora. Prvi kompilator bio je napisan od strane Grace Hopper 1952. godine za programski jezik A-0. Prvi kompletan kompilator koji je imao širu primenu i bitno uticao na dalji razvoj je kompilator za

FORTRAN realizovan 1957. godine od strane IBM-ovog tima kojim je rukovodio John Backus. U to vreme, negde oko 1960. godine, COBOL je bio prvi jezik za koji su bili realizovani kompilatori za različite procesore i koji je mogao da se koristi na više različitih mašina.

Nakon ovog početnog perioda veoma brzo je prihvaćena ideja o korišćenju viših programskih jezika za razvoj programa. Time su programeri oslobođeni zavisnosti od mašine i mogli su više da se posvete algoritmima za rešavanje problema. Kako su programski jezici i arhitekture računara postajali sve kompleksniji i programi samih kompilatora su bili sve složeniji.

Prvi kompilatori su bili pisani na asemblerskom jeziku. Prvi tzv. *self-hosting* kompilator, gde je isti programski jezik korišćen i za razvoj kompilatora, bio je realizovan za programski jezik Lisp, na MIT-u od strane Tim Harta i Mika Levina. Za generisanje ovog kompilatora poslužio je prethodno napravljeni interpretator za Lisp. Posle toga, negde od 1970. godine, postaje uobičajena praksa da se kompilatori razvijaju na jeziku za koji se i prave. Pri tome se starije verzije kompilatora koriste za dobijanje novih što je poznato kao tehnika butstrepanja. Naravno, prva verzija nekog kompilatora mora da bude napisana na nekom drugom programskom jeziku.

## 1.8 Pitanja

1. Objasniti namenu programskih prevodilaca.
2. Šta je to assembler?
3. Čemu služe makroprocesori?
4. Objasniti razliku između kompilatora i interpretatora.
5. Navesti osnovne faze rada kompilatora.
6. Objasniti šta je to međukod.
7. Šta su to višezjezički kompilatori?
8. Nacrtati potpunu šemu strukture kompilatora.
9. Kada su nastali prvi kompilatori?

## Glava 2

# Elementi teorije formalnih jezika

Danas se podrazumeva da se programski jezik formalno definiše već prilikom njegovog projektovanja. Formalni opis jezika na neki način predstavlja njegovu standardizaciju i osnovu za pravilnu primenu i dalji razvoj jezika. Takođe, formalni opis je jedan od uslova za uspešnu realizaciju kompilatora za određeni programski jezik.

### 2.1 Opis jezika

Razmotrićemo šta se sve u okviru programskog jezika može formalno opisati.

Najpre se opisuju leksički elementi jezika. To su leksičke celine, lekseme, koje imaju neki smisao u jeziku; na primer, broj, identifikator, ključna reč i sl. Ove leksičke celine prepoznaje leksički analizator i prosleđuje ih sintaksnom analizatoru.

Na sledećem nivou se vrši opis sintakse programskog jezika. Formalnim opisom sintakse jezika definišu se pravilne konstrukcije, odnosno koje kombinacije reči su dozvoljene i u kom kontekstu. Opis sintakse jezika je osnova sintaksnom analizatoru da proveri da li su naredbe programskog jezika pravilno napisane i da ih preslika u strukturu iz koje će moći da budu generisane naredbe asemblerskog koda.

Treći nivo formalnog opisa jezika može da se odnosi na semantiku. Odnosno, već na nivou formalnog opisa jezika mogu da se definišu neka pravila koja će se koristiti za semantičku proveru programa.

Za sva ova tri nivoa opisa jezika koriste se formalni jezici i formalne gramatike.

U nastavku će biti dati osnovni elementi teorije formalnih jezika. To je šira naučna oblast koja se može posmatrati i kao lingvistička disciplina. Nalazi široku primenu u mnogim oblastima informatike kao što su prepoznavanja uzoraka, obrada prirodnih jezika i sl.

## 2.2 Elementi jezika

### 2.2.1 Azbuka

Prilikom opisa jezika, polazi se od azbuke kao osnovnog pojma. Neka je  $\mathbf{V}$  konačan neprazan skup elemenata. Elementi skupa  $\mathbf{V}$  su simboli ili slova, a sam skup apstraktna azbuka ili samo azbuka.

#### Primer 2.1 Primeri azbuka

Primeri azbuka:

- $\mathbf{V} = \{0, 1\}$  – azbuci  $\mathbf{V}$  pripadaju samo cifre 0 i 1;
- $\mathbf{V} = \{a, b, c\}$  – azbuci pripadaju samo slova  $a$ ,  $b$  i  $c$ .

Za opis programskih jezika se koriste nešto složenije azbuke. Na primer, jednu takvu azbuku mogu da čine sledeći simboli:

- specijalni znaci:  $+$ ,  $-$ ,  $*$ ,  $:=$ , ...
- reči kao što su: **begin**, **end**, **if**, **then**, ...

Napomena: u ovom slučaju se svaka reč tretira kao jedan simbol.

### 2.2.2 Reč

Reč u kontekstu formalnih jezika se definiše kao konačan niz simbola azbuke  $\mathbf{V}$ . Niz koji ne sadrži nijedan simbol naziva se **prazna reč** i označava sa  $\varepsilon$ .

#### Primer 2.2 Reči

Neka je  $\mathbf{V} = \{a, b, c\}$ . Sledeći nizovi su primeri reči azbuke  $\mathbf{V}$ :  $\varepsilon$ ,  $a$ ,  $b$ ,  $c$ ,  $aa$ ,  $bb$ ,  $cc$ ,  $ab$ ,  $ac$ ,  $abc$ ,  $aabc$ .

Reči su uređeni nizovi tako da se reč  $ab$  razlikuje od reči  $ba$  ( $ab \neq ba$ ).

Reč se može i formalno definisati sledećim skupom pravila:

1.  $\varepsilon$  je reč nad azbukom  $\mathbf{V}$ .

2. Ako je  $x$  reč azbuke  $V$  i ako je  $a$  element azbuke  $V$ , tada je i  $xa$  reč azbuke  $V$ .
3. Niz  $y$  je reč nad azbukom  $V$  ako i samo ako je dobijen primenom pravila 1 i 2.

Za označavanje reči korišćemo završna mala slova abecede napisana boldirano:  $u, v, w, x, y, z$ .

Broj slova (simbola) u reči definiše se kao dužina reči. Za označavanje dužine reči  $x$  koristi se simbol  $|x|$ .

**Primer 2.3** Dužina reči

Važi  $|abc| = 3$ ,  $|aaaa| = 4$ , dok je  $|\varepsilon| = 0$ .

### 2.2.3 Proizvod reči

Ako su  $x$  i  $y$  dve reči azbuke  $V$ , proizvod ili spajanje reči je operacija kojom se stvara nova reč tako što se na jednu reč nadovezuje druga reč.

**Primer 2.4** Proizvod reči

Neka je  $x = aA$  i  $y = ab$ . Proizvodom ovih reči nastaje reč  $z = xy = aAab$ .

Takođe važi da je:  $\varepsilon x = x\varepsilon = x$ . Prazna reč je neutralni element za operaciju proizvoda (nadovezivanja) reči.

### 2.2.4 Eksponent reči

Višestruki proizvod iste reči se definiše kao eksponent reči.

**Primer 2.5** Eksponent reči

Važe sledeće jednakosti:

$$\begin{aligned} xx &= x^2, & xxx &= x^3, \\ x &= x^1, & x^0 &= \varepsilon. \end{aligned}$$

### 2.2.5 Transponovana reč

Transponovana reč reči  $x$ , u oznaci  $x^T$ , definiše se sledećim pravilima:

1.  $\varepsilon^\top = \varepsilon$
2.  $(\mathbf{x}a)^\top = a\mathbf{x}^\top$

Rezultat operacije transponovanja reči je reč u kojoj su slova ispisana u obrnutom redosledu u odnosu na originalnu reč.

#### Primer 2.6 Transponovana reč

Neka je data reč  $\mathbf{x} = baba$ . Tada je  $\mathbf{x}^\top = abab$ . Ako je  $\mathbf{x} = radar$ , tada je  $\mathbf{x}^\top = radar$ . Ovakve reči, kod kojih su original i transponovana reč isti, nazivaju se palindromi.

### 2.2.6 Delovi reči

Razlikujemo i sledeće pojmove:

**Prefiks** reči  $\mathbf{x}$  je niz koji se dobija izbacivanjem nijednog ili više krajnjih slova reči  $\mathbf{x}$ .

**Sufiks** reči  $\mathbf{x}$  je niz koji se dobija izbacivanjem nijednog ili više početnih simbola reči  $\mathbf{x}$ .

**Podniz** reči  $\mathbf{x}$  je reč koja se dobija kada se izbaci neki prefiks i/ili neki sufiks reči  $\mathbf{x}$ . Svaki prefiks i svaki sufiks reči  $\mathbf{x}$  su podnizovi reči  $\mathbf{x}$ , dok svaki podniz reči  $\mathbf{x}$  ne mora da bude ni sufiks ni prefiks reči  $\mathbf{x}$ . Za svaku reč  $\mathbf{x}$  reči  $\mathbf{x}$  i  $\varepsilon$  su njeni prefiksi, sufiksi i podnizovi.

Svaki niz koji se dobija izbacivanjem nijednog ili više sukcesivnih simbola iz reči  $\mathbf{x}$ , naziva se podsekvencom reči  $\mathbf{x}$ .

#### Primer 2.7 Delovi reči

Neka je data reč *banana*.

Reči  $\varepsilon$ , *b*, *ba*, *ban*, *bana*, *banan* i *banana* su njeni prefiksi; *b*, *ba*, *ban*, *bana*, *banan* su pravi prefiksi.

Reči  $\varepsilon$ , *a*, *na*, *ana*, *nana*, *anana* i *banana* su njeni sufiksi; *a*, *na*, *ana*, *nana*, *anana* su pravi sufiksi.

## 2.3 Formalni jezik

Neka je data azbuka  $\mathbf{V}$ . Skup svih reči nad azbukom  $\mathbf{V}$  se označava sa  $\mathbf{V}^*$ . Napomenimo da je skup  $\mathbf{V}^*$  uvek beskonačan, bez obzira na to kako je definisana



azbuka  $\mathbf{V}$ . Čak i u slučaju kada azbuka  $\mathbf{V}$  sadrži samo jedan simbol  $\mathbf{V} = \{a\}$ , skup  $\mathbf{V}^* = \{\varepsilon, a, aa, aaa, aaaa, \dots\}$  je beskonačan.

Formalni jezik nad azbukom  $\mathbf{V}$  je bilo koji podskup skupa  $\mathbf{V}^*$ . Bilo koji podskup reči nad azbukom  $\mathbf{V}$ , bilo da je konačan ili beskonačan, predstavlja jezik. Kako je  $\mathbf{V}^*$  uvek beskonačan skup, broj njegovih podskupova je takođe beskonačan. Drugim rečima, ma koliko da je azbuka siromašna, čak i kada se sastoji od samo jednog simbola, nad njom se može definisati beskonačan broj različitih formalnih jezika.

Naglasimo još jednom, formalni jezik  $\mathbf{L}$  nad azbukom  $\mathbf{V}$  je bilo koji skup reči nad tom azbukom.

Prema ovoj definiciji, formalni jezik je i prazan skup reči  $\emptyset$ , kao i skup  $\{\varepsilon\}$  koji sadrži samo praznu reč  $\varepsilon$ .

#### Primer 2.8 Formalni jezik

Neka je data azbuka  $\mathbf{V}$  koju čine sva slova latiničnog alfabeta

$$\mathbf{V} = \{a, b, c, \dots, x, y, z\}.$$

Tada skupu  $\mathbf{V}^*$  pripadaju svi nizovi koji mogu da se formiraju od slova latinice. Svaki podskup tog skupa reči je jedan formalni jezik. Na primer, sledeći skupovi su formalni jezici:

1. skup nizova koji se sastoji samo od samoglasnika;
2. skup nizova koji se sastoji samo od suglasnika;
3. skup nizova koji sadrže najmanje jedan samoglasnik i najmanje jedan suglasnik;
4. skup nizova koji su palindromi (čitaju se isto sa leve i sa desne strane);
5. skup nizova čija dužina je manja od 10;
6. skup nizova koji se sastoje samo od jednog slova;
7. skup  $\{ja, ti, on, mi, vi, oni\}$ ;
8. prazan skup.

Uočite da su prva četiri jezika beskonačna, dok su ostali konačni.

### 2.3.1 Operacije nad jezicima

Ako je  $\mathbf{L}$  formalni jezik onda se može definisati i njegov reverzni jezik u oznaci  $\mathbf{L}^R$ , kao skup svih transponovanih (reverznih) reči jezika  $\mathbf{L}$ :

$$\mathbf{L}^R = \{\mathbf{x}^T \mid \mathbf{x} \in \mathbf{L}\}.$$

Ako su  $\mathbf{L}$  i  $\mathbf{M}$  formalni jezici, onda je  $\mathbf{L} \cup \mathbf{M}$  unija ova dva jezika i obuhvata sve reči koje pripadaju ili jeziku  $\mathbf{L}$  ili jeziku  $\mathbf{M}$ :

$$\mathbf{L} \cup \mathbf{M} = \{\mathbf{x} \mid \mathbf{x} \in \mathbf{L} \vee \mathbf{x} \in \mathbf{M}\}.$$

Ako su  $\mathbf{L}$  i  $\mathbf{M}$  formalni jezici, onda je  $\mathbf{L} \circ \mathbf{M}$  proizvod ova dva jezika i obuhvata sve reči koje nastaju operacijom nadovezivanja reči jezika  $\mathbf{M}$  na reči jezika  $\mathbf{L}$ .

$$\mathbf{L} \circ \mathbf{M} = \{\mathbf{xy} \mid \mathbf{x} \in \mathbf{L} \wedge \mathbf{y} \in \mathbf{M}\}.$$

Potpuno zatvaranje jezika  $\mathbf{L}$ , koje se označava sa  $\mathbf{L}^*$ , je skup svih reči koje nastaju kao proizvodi reči jezika  $\mathbf{L}$  uključujući i  $\varepsilon$ :

$$\mathbf{L}^* = \bigcup_{i=0}^{\infty} \mathbf{L}^i.$$

Pozitivno zatvaranje jezika  $\mathbf{L}$ , za koje se koristi oznaka  $\mathbf{L}^+$ , definiše se kao:

$$\mathbf{L}^+ = \bigcup_{i=1}^{\infty} \mathbf{L}^i.$$

To znači da pozitivno zatvaranje isključuje iz potpunog zatvaranja praznu reč, tj. važi:  $\mathbf{L}^+ = \mathbf{L}^* \setminus \varepsilon$ .

### Primer 2.9 Operacije nad jezicima

Neka su date azbuke

$$\mathbf{L} = \{A, B, C, \dots, Z, a, b, c, \dots, z\} \text{ i } \mathbf{D} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

Kako se slova azbuke mogu posmatrati kao reči dužine 1, onda je svaki od skupova  $\mathbf{L}$  i  $\mathbf{D}$  i formalni jezik. Operacijama nad ovim jezicima mogu se definisati sledeći jezici:

- $\mathbf{L} \cup \mathbf{D}$  skup slova i cifara,
- $\mathbf{L} \circ \mathbf{D}$  skup svih reči koje se sastoje od slova iza kog stoji cifra,
- $\mathbf{L}^4$  skup svih četvoroslovnih reči,
- $\mathbf{L}^*$  skup svih nizova slova uključujući i  $\varepsilon$ ,
- $\mathbf{L} \circ (\mathbf{L} \cup \mathbf{D})^*$  skup nizova slova i cifara koji započinju slovom,
- $\mathbf{D}^+$  skup svih celih brojeva, nizova koji se sastoje od jedne ili više cifara.

## 2.4 Opis jezika

Možemo da identifikujemo dva osnovna problema u radu sa formalnim jezicima. Prvi je problem formalnog opisa skupa reči koje pripadaju jeziku, a drugi je problem utvrđivanja da li zadata reč pripada određenom jeziku.

Prvi problem se rešava pomoću gramatika. Gramatike su sredstvo za opis jezika. Drugi problem može da se posmatra kao problem prepoznavanja jezika i to je ključni problem koji se rešava pomoću automata kao uređaja za prepoznavanje jezika.

Gramatika je skup pravila kojima se definiše koje reči neke azbuke pripadaju određenom jeziku. Skup pravila gramatike može da bude zadat na bilo koji način, uz uslov da je skup reči jezika jednoznačno definisan.

### Primer 2.10 Pravila gramatike

Naredbe jednog programskog jezika se formalno mogu posmatrati kao reči tog jezika. Sledećim skupom pravila definiše se šta se može smatrati naredbom u kontekstu ovog formalnog jezika.

$$\begin{aligned} \text{izraz} &\rightarrow \text{id} \\ \text{izraz} &\rightarrow \text{const} \\ \text{izraz} &\rightarrow \text{izraz} + \text{izraz} \\ \text{izraz} &\rightarrow \text{izraz} * \text{izraz} \\ \text{izraz} &\rightarrow (\text{izraz}) \\ \text{naredba} &\rightarrow \text{id} := \text{izraz} \\ \text{naredba} &\rightarrow \text{while } (\text{izraz}) \text{ do } \text{naredba} \\ \text{naredba} &\rightarrow \text{if } (\text{izraz}) \text{ then } \text{naredba} \end{aligned}$$

### 2.4.1 Izvođenje reči

Reči jezika izvode se primenom pravila gramatike. Obično se definiše startni simbol i nastoji se da se primenom pravila gramatike od tog simbola generišu reči jezika. Na primer, u primeru 2.10 može se kao startni simbol uzeti pojam *naredba*. Postupak generisanja reči primenom pravila gramatike naziva se izvođenje reči.

### Primer 2.11 Izvođenje reči

Pokazaćemo izvođenje reči `# alfa := alfa*beta+pi+90.0 #` koja pripada jeziku definisanom skupom pravila iz primera 2.10. Analizator ovu reč vidi

kao  $\# \text{id}_1 := \text{id}_1 * \text{id}_2 * \text{id}_3 + \text{const} \#$ .

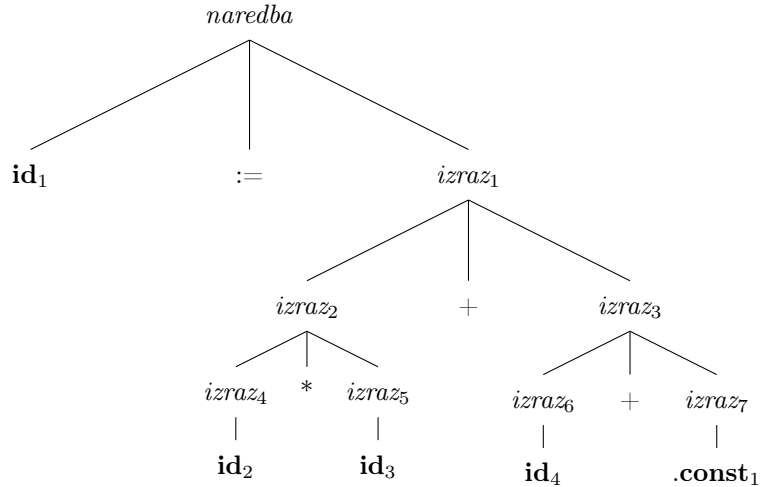
```

naredba1 → id1 := izraz1
           → id1 := izraz2 + izraz3
           → id1 := izraz4 * izraz5 + izraz3
           → id1 := id1 * izraz5 + izraz3
           → id1 := id1 * izraz6 + izraz7 + izraz3
           → id1 := id1 * id2 + izraz7 + izraz3
           → id1 := id1 * id2 + id3 + izraz3
           → id1 := id1 * id2 + id3 + const

```

**Napomena:** Brojevi koji su dopisani na reči koje označavaju određene lekseme koriste se da bi se razlikovale različite instance iste lekseme.

Postupak izvođenja reči jezika primenom pravila gramatike se može prikazati preko strukture koje se naiva sintaksno stablo. Pri tome svaki neterminalni čvor u stablu sa svojim granama predstavlja jedno primenjeno pravilo. U završnim (terminalnim) čvorovima predstavljena su slova izvedene reči. Slika 2.1 prikazuje sintaksno stablo koje se dobija prilikom analize naredbe dodeljivanja iz primera 2.11. Može se reći da je cilj sintaksne analize generisanje sintaksnog stabla.



SLIKA 2.1: Sintaksno stablo za reč iz primera 2.11

### 2.4.2 Elementi gramatike

Da bi se definisala gramatika nekog jezika potrebni su sledeći elementi:

1. **Skup terminalnih simbola** – Skup završnih simbola koji ulaze u sastav reči jezika koji se definiše.
2. **Skup neterminalnih simbola** – Skup pomoćnih simbola koji se koriste za definisanje pravila za generisanje reči jezika ali ne ulaze u same reči jezika.
3. **Startni simbol** – neterminalni simbol od kojeg započinje generisanje svih reči jezika. Obično se ovim simbolom definiše glavna sintaksna kategorija koja je opisana jezikom. Na primer program u okviru programskih jezika.
4. **Skup pravila** oblika  $x \rightarrow y$ , kojima se stvaraju nizovi u procesu generisanja reči jezika koji se opisuje.

Za označavanje elemenata ovih skupova obično koristimo sledeću notaciju:

- **Terminalni simboli:**

- mala slova abecede:  $a, b, c, \dots$
- simboli operatora:  $*, +, -, \dots$
- specijalni znaci:  $(, ), <, >, \dots$
- cifre:  $0, 1, \dots, 9$ ;
- boldirane reči kao što su **id**, **if**, **then**, i sl.

- **Neterminalni simboli:**

- velika slova:  $A, B, C, \dots$
- $S$ , ako se pojavljuje je obično startni simbol;
- reči napisane malim slovima italik: *expr*, *nar*, ili između zagrada:  $\langle \text{str} \rangle$ ,  $\langle \text{nar} \rangle$

- **Pravila:**

- $x ::= y$  ili  $x \rightarrow y$

### 2.4.3 Direkto izvođenje i redukcija

Ukoliko postoji reč  $z = \mathbf{pxq}$  i pravilo  $\mathbf{xy}$ , onda se nakon primene ovog pravila reč  $z$  preslikava u reč  $w = \mathbf{pyq}$  i kažemo da se reč  $w$  direktno izvodi od reči  $z$ . To se obično označava sa

$$z \rightarrow w.$$

Na osnovu pravila  $\mathbf{x} \rightarrow \mathbf{y}$  važi i da se  $\mathbf{y}$  direktno izvodi od  $\mathbf{x}$ . Takođe važi i da se svaka reč direktno izvodi od same sebe, odnosno  $\mathbf{x} \rightarrow \mathbf{x}$ .

Ovaj proces se može posmatrati i u suprotnom smeru, odnosno kao direktna redukcija reči  $\mathbf{w}$  na reč  $\mathbf{z}$ . Isto tako se  $\mathbf{y}$  direktno redukuje na  $\mathbf{x}$ .

Ukoliko se  $\mathbf{x}$  preslikava u  $\mathbf{y}$  primenom više smena, odnosno ukoliko postoji skup smena oblika

$$\mathbf{x} \rightarrow \mathbf{u}_1, \quad \mathbf{u}_1 \rightarrow \mathbf{u}_2, \quad \mathbf{u}_2 \rightarrow \mathbf{u}_3, \quad \dots, \quad \mathbf{u}_n \rightarrow \mathbf{y},$$

umesto termina direktno izvođenje i direktna redukcija koriste se termini izvođenje i redukcija. Kažemo  $\mathbf{y}$  se izvodi od  $\mathbf{x}$  i  $\mathbf{y}$  se redukuje na  $\mathbf{x}$ . Izvođenje je višestruko primenjeno direktno izvođenje i obično se označava sa:

$$\mathbf{x} \xrightarrow{*} \mathbf{y}.$$

## 2.5 Formalne gramatike

Noam Chomsky je još u svojim ranim radovima dao sledeću definiciju formalnih gramatika:

*Svaka formalna gramatika  $\mathbf{G}$  se može definisati kao torka*

$$\mathbf{G} = (\mathbf{V}_n, \mathbf{V}_t, S, \mathbf{P}),$$

*gde je  $\mathbf{V}_n$  skup neterminalnih simbola,  $\mathbf{V}_t$  skup terminalnih simbola,  $S$  startni simbol i  $\mathbf{P}$  skup smena definisan na sledeći način:*

$$\mathbf{x} \rightarrow \mathbf{y}, \quad \text{gde je } \mathbf{x} \in \mathbf{V}^* \mathbf{V}_n \mathbf{V}^* \wedge \mathbf{y} \in \mathbf{V}^*,$$

*pri čemu je  $S \in \mathbf{V}_n$ ,  $\mathbf{V} = \mathbf{V}_t \cup \mathbf{V}_n$  i  $\mathbf{V}_t \cap \mathbf{V}_n = \emptyset$ .*

Uočimo da je uslov da reč na levoj strani pravila mora da sadrži bar jedan neterminalni simbol.

Formalni jezik  $\mathbf{L}$  definisan gramatikom  $\mathbf{G}$  je skup reči izvedenih iz startnog simbola gramatike koje se sastoje od terminalnih simbola:

$$\mathbf{L}(\mathbf{G}) = \{\mathbf{w} \mid S \xrightarrow{*} \mathbf{w}, \mathbf{w} \in \mathbf{V}_t^*\}.$$

### Primer 2.12 Formalna gramatika

Data je gramatika

$$\mathbf{G} = (\{S, A, B, C, D\}, \{a, b\}, S, \mathbf{P}),$$

gde je  $\mathbf{P}$  sledeći skup smena:

1.  $S \rightarrow CD$

$$2. C \rightarrow aCa$$

$$3. C \rightarrow bCB$$

$$4. AD \rightarrow aD$$

$$5. BD \rightarrow bD$$

$$6. Aa \rightarrow bA$$

$$7. Aa \rightarrow bA$$

$$8. Ba \rightarrow aB$$

$$9. Bb \rightarrow bB$$

$$10. C \rightarrow \varepsilon$$

$$11. D \rightarrow \varepsilon$$

Datom gramatikom je opisan jezik

$$\mathbf{L}(\mathbf{G}) = \{\mathbf{ww} \mid \mathbf{w} \in \{a, b\}^*\}.$$

Primer izvođenja reči *abab*:

$$S \xrightarrow{1} CD \xrightarrow{2} aCaD \xrightarrow{3} abCBaD \xrightarrow{10} abBaD \xrightarrow{8} abaBD \xrightarrow{5} ababD \xrightarrow{11} abab$$

Gramatika mora da bude redukovana. Redukovana gramatika je gramatika koja ne sadrži beskorisna pravila, kao ni beskonačna pravila. Beskorisna pravila su ona koje sadrže simbole koji ne mogu da budu izvedeni iz startnog simbola gramatike, a beskonačna ona koja sadrže neterminalne simbole koji nikada ne mogu da budu ukinuti. To se formalno može definisati preko sledećih pravila.

- Za svaki neterminal  $A$  postoji izvođenje  $S \xrightarrow{*} \alpha A \beta$ , gde je  $S$  startni simbol gramatike, a  $\alpha, \beta \in \mathbf{V}^*$ .
- Za svaki neterminal  $A$  postoji izvođenje  $A \xrightarrow{*} \alpha$ , gde je  $\alpha \in \mathbf{V}_t^*$ .

Gramatika ne mora da bude jednoznačna. Kod nejednoznačnih gramatika za jedan ulazni niz je moguće kreirati veći broj sintaksnih stabala.

#### Primer 2.13 Neredukovana gramatika

Gramatika  $\mathbf{G} = (\{W, X, Y, Z\}, \{a\}, \mathbf{W}, \mathbf{P})$  gde je  $\mathbf{P}$  sledeći skup pravila:

$$1. W \rightarrow aW$$

$$2. W \rightarrow Z$$

$$3. W \rightarrow X$$

$$4. Z \rightarrow aZ$$

$$5. X \rightarrow a$$

$$6. Y \rightarrow aa$$

je primer neredukovane gramatike iz sledećih razloga:

- Pravilo 2 je neupotrebljivo pravilo jer je  $Z$  beskonačan simbol, nikada se ne ukida.
- Pravilo 4 je neupotrebljivo iz istog razloga.
- Pravilo 6 je neupotrebljivo zato što ne postoji izvođenje kojim se generiše neterminalni simbol  $Y$ .

### Primer 2.14 Nejednoznačna gramatika

Neka je data gramatika

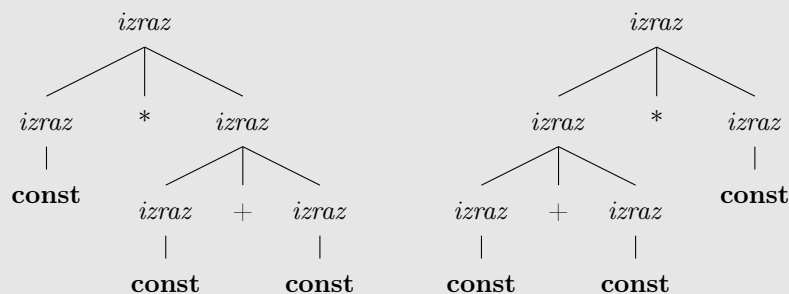
$$\mathbf{G} = (\{Izraz\}, \{\mathbf{const}, +, *\}, Izraz, \mathbf{P}),$$

gde je  $\mathbf{P}$  sledeći skup pravila:

1.  $Izraz \rightarrow Izraz + Izraz$
2.  $Izraz \rightarrow Izraz * Izraz$
3.  $Izraz \rightarrow \mathbf{const}$

U nastavku prikazana su dva sintaksna stabla koja odgovaraju izvođenju izraza

$$\mathbf{const} * \mathbf{const} + \mathbf{const}.$$



Kako stabla na slici nisu ista, gramatika  $\mathbf{G}$  nije jednoznačna.

Jednoznačna gramatika ekvivalentna gramatici  $\mathbf{G}$  može da se definiše na sledeći način:

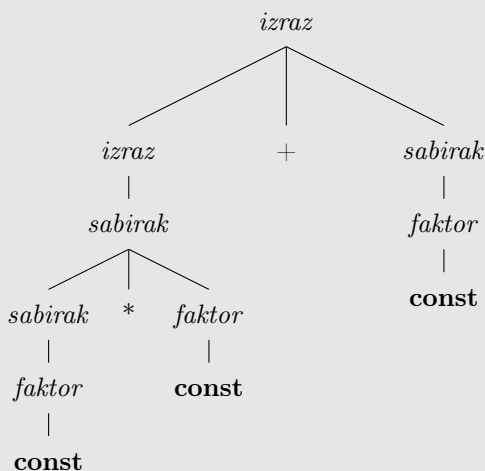
$$\mathbf{G}_1 = (\{Izraz, Sabirak, Faktor\}, \{\mathbf{const}, +, *\}, Izraz, \mathbf{P}),$$



gde je  $\mathbf{P}$  skup pravila:

1.  $Izraz \rightarrow Izraz + Sabirak$
2.  $Izraz \rightarrow Sabirak$
3.  $Sabirak \rightarrow Sabirak * Faktor$
4.  $Sabirak \rightarrow Faktor$
5.  $Faktor \rightarrow \mathbf{const}$

U ovom slučaju za izraz  $\mathbf{const} * \mathbf{const} + \mathbf{const}$ , moguće je jednoznačno kreirati sintaksno stablo koje je prikazano u nastavku.



## 2.6 Hierarhija gramatika po Čomskom

Još u svojim ranim radovima, Čomski je identifikovao četiri tipa formalnih gramatika i jezika. Gramatike koje zadovoljavaju gore date opšte uslove je nazvao gramatikama tipa nula. Praktično, svaka formalna gramatika je gramatika tipa nula. Ostali tipovi gramatika imaju strože definisane uslove koje treba da zadovolje pravila. To znači da se tim tipovima gramatika može definisati uži skup jezika ali se zato može jednostavnije vršiti analiza i prepoznavanje takvih jezika.

### 2.6.1 Gramatike tipa jedan ili konteksne gramatike

Gramatike tipa jedan su formalne gramatike koje pored opštih uslova koji zadovoljavaju gramatike tipa nula imaju pravila  $\mathbf{x} \rightarrow \mathbf{y}$  koja zadovoljavaju sledeći uslov:

$$|\mathbf{x}| \leq |\mathbf{y}|.$$

To znači da se reči sa leve strane pravila mogu preslikavati samo u reči koje su jednake ili veće dužine. Drugim rečima, nisu dozvoljena pravila oblika:

$$\mathbf{x} \rightarrow \varepsilon.$$

Gramatike tipa jedan, kao i gramatike tipa nula se obično nazivaju i konteksnim gramatikama. To je zbog toga što se neterminal koji je obavezan na levoj strani smene kod ovih gramatika nalazi u nekom kontekstu i čitav taj kontekst se preslikava u novu reč.

**Primer 2.15** Gramatika tipa jedan – kontekсна gramatika

Gramatika  $\mathbf{G} = (\{0, 1\}, \{S, A, B, C\}, S, \mathbf{P})$  gde je  $\mathbf{P}$  skup smena definisan sa:

1.  $S \rightarrow 0B0$
2.  $C0 \rightarrow 100$
3.  $B \rightarrow 0BC$
4.  $B \rightarrow 1$
5.  $C1 \rightarrow 1C$

je kontekсна gramatika tipa jedan.

Primer izvođenja reči jezika ove gramatike:

$$\begin{aligned} S &\xrightarrow{1} 0B0 \xrightarrow{3} 000BCC0 \xrightarrow{3} 0000BCC0 \xrightarrow{2} 0000BCC100 \\ &\xrightarrow{5} 0000BC1C00 \xrightarrow{0} 000BC11000 \xrightarrow{5} 0000B1C1000 \\ &\xrightarrow{5} 0000B11C000 \xrightarrow{2} 0000B1110000 \xrightarrow{4} 000011110000 \end{aligned}$$

### 2.6.2 Gramatike tipa dva ili beskonteksne gramatike

Gramatike tipa dva su gramatike kod kojih su pravila definisana tako da se na levoj strani nalazi samo jedan neterminal. Neterminali se preslikavaju u reči. Kako se neterminali prilikom izvođenja posmatraju izolovano od konteksta u kome se nalaze, ove gramatike se nazivaju beskonteksnim gramatikama. Znači smene kod ovih gramatika su oblika

$$A \rightarrow \mathbf{y}, \quad \text{gde je } A \in \mathbf{V}_n \text{ i } \mathbf{y} \in \mathbf{V}^*.$$

Za opis programskih jezika se obično koriste beskonteksne gramatike.

**Primer 2.16** Gramatika tipa dva – beskontekсна gramatika

Gramatika  $\mathbf{G}$  definisana je na sledeći način:

$$\mathbf{G} = (\{E, A\}, \{ (, ), +, -, *, /, \text{id} \}, E, \mathbf{P}),$$

gde je  $\mathbf{P}$  skup smena:

$$\begin{aligned} E &\rightarrow EAE \mid (E) \mid -E \mid \text{id} \\ A &\rightarrow + \mid - \mid * \mid / \end{aligned}$$

Datom gramatikom definisan je jezik koji čine aritmetički izrazi. Primer reči koja pripada jeziku definisanom ovom gramatikom:  $\text{id}_1 * (\text{id}_2 + \text{id}_3 * \text{id}_4)$ . Izvođenje ove reči bi bilo:

$$\begin{aligned} E &\rightarrow EAE \\ &\rightarrow \text{id}_1 AE \\ &\rightarrow \text{id}_1 * E \\ &\rightarrow \text{id}_1 * (E) \\ &\rightarrow \text{id}_1 * (EAE) \\ &\rightarrow \text{id}_1 * (\text{id}_2 AE) \\ &\rightarrow \text{id}_1 * (\text{id}_2 + E) \\ &\rightarrow \text{id}_1 * (\text{id}_2 + EAE) \\ &\rightarrow \text{id}_1 * (\text{id}_2 + \text{id}_3 AE) \\ &\rightarrow \text{id}_1 * (\text{id}_2 + \text{id}_3 * E) \\ &\rightarrow \text{id}_1 * (\text{id}_2 + \text{id}_3 * \text{id}_4) \end{aligned}$$

**2.6.3 Gramatike tipa tri ili regularne gramatike**

Gramatike tipa tri su formalne gramatike koje zadovoljavaju opšte uslove ali su ograničene na smene oblika:

$$A \rightarrow aB \text{ ili } A \rightarrow a, \quad \text{gde je } A, B \in \mathbf{V}_n \text{ i } a \in \mathbf{V}_t.$$

Naziv regularne gramatike dolazi odatle što se gramatikama tipa tri generišu regularni izrazi koji se efikasno mogu prepoznavati konačnim automatima. Regularne gramatike imaju široku primenu u mnogim oblastima u kojima se problem svodi na prepoznavanje nizova. U kontekstu programskih prevodilaca koriste se za opis leksičkih elemenata jezika (identifikatora, konstanti, literala i sl) koje prepoznaje leksički analizator, o čemu će biti reči kasnije.

**Primer 2.17** Gramatika tipa tri – regularna gramatika

Gramatika  $\mathbf{G}$  definisana je na sledeći način:

$$\mathbf{G} = (\{0, 1\}, \{A, B, C, D, E, F, G\}, A, \mathbf{P}),$$

pri čemu je  $\mathbf{P}$  skup smena:

$$\begin{aligned} A &\rightarrow 0B \mid 1D & B &\rightarrow 0C \mid 1F & C &\rightarrow 0C \mid 1F \mid \varepsilon \\ D &\rightarrow 1E \mid 0F & E &\rightarrow 1E \mid 0F \mid \varepsilon & F &\rightarrow 0G \mid 1G \\ G &\rightarrow 0G \mid 1G \mid \varepsilon \end{aligned}$$

Primer izvođenja:

$$A \rightarrow 0B \rightarrow 00C \rightarrow 000C \rightarrow 0001F \rightarrow 00011G \rightarrow 00011\varepsilon.$$

Dakle, reč 00011 pripada jeziku definisanom datom gramatikom.

Gramatike tipa nula su najopštije i praktično su sinonim za algoritam. Sva algoritamska preslikavanja se mogu opisati gramatikama tipa nula.

Gramatike tipa tri pokrivaju najuži skup jezika ali je za ove jezike najjednostavnije rešiti problem prepoznavanja. Prepoznaju se pomoću konačnih automata.

Između jezika definisanih različitim tipovima gramatika postoji sledeća relacija

$$L(3) \subset L(2) \subset L(1) \subset L(0).$$

**2.6.4 Rečenične forme**

Svi nizovi (reči) koji nastaju u postupku generisanja jezika su rečenične forme tog jezika. Sve rečenične forme jezika se redukuju na startni simbol.

**Primer 2.18** Rečenične forme

Gramatika  $\mathbf{G}$  definisana je na sledeći način:

$$\mathbf{G} = (\{E, A\}, \{(\,, \,), +, -, *, /, \text{id}\}, E, \mathbf{P}),$$

gde je  $\mathbf{P}$  skup smena:

$$E \rightarrow EAE \mid (E) \mid -E \mid \text{id} \qquad A \rightarrow + \mid - \mid * \mid /$$

Primeri rečeničnih formi:

$$E \rightarrow EAE \rightarrow E + E \rightarrow E + EAE.$$

### 2.6.5 Fraze

Fraze se definišu za beskonteksne gramatike. Ako je **xuy** rečenična forma, **u** je fraza za neterminal  $A$  ako se izvodi iz neterminala  $A$ . To znači da sve reči izvedene od istog neterminalnog simbola čine skup fraza tog simbola. Fraza koja ne sadrži druge fraze istog neterminala je prosta fraza.

#### Primer 2.19 Fraza

Neka je data gramatika iz prethodnog primera.

Posmatrajmo izvođenje:

$$E \rightarrow EAE \rightarrow (E)AE \rightarrow ((E))AE \rightarrow ((-E))AE \rightarrow (-\mathbf{id})AE$$

Reči  $EAE$ ,  $(E)$ ,  $((E))$ ,  $((-E))$ ,  $(-\mathbf{id})$  su primeri fraza neterminala  $E$ ; sve nastaju od neterminala  $E$ . Pri tome su  $EAE$ ,  $(E)$ ,  $((-E))$  i  $(-\mathbf{id})$  proste fraze, dok  $((E))$  nije prosta fraza zato što sadrži frazu  $(E)$ .

## 2.7 Normalne forme gramatika

Pod normalnim formama gramatika podrazumevamo standardni način zadavanja gramatika određenog tipa. To znači da se smene jedne gramatike mogu prikazati na neki drugi način a da se pri tome ne promeni jezik koji definišu. Tako dolazimo i do pojma ekvivalentnih gramatika. Dve gramatike su ekvivalentne ako definišu isti jezik.

Nekada je potrebno smene gramatike predstaviti na baš određeni način da bi se mogao primeniti neki određeni postupak analize. U te svrhe, za određene tipove gramatika, definisane su različite normalne forme.

### 2.7.1 Normalne forme za konteksne gramatike

Ako je  $\mathbf{G}$  konteksna gramatika, onda postoji njoj ekvivalentna konteksna gramatika  $\mathbf{G}_1$  u kojoj je svaka smena oblika:

$$\mathbf{x}A\mathbf{y} \rightarrow \mathbf{x}\mathbf{r}\mathbf{y}, \quad \text{gde je } A \in \mathbf{V}_n, \ x, \mathbf{y} \in \mathbf{V}^* \text{ i } \mathbf{r} \in \mathbf{V}^+$$

Na osnovu ove normalne forme jasno se vidi zbog čega su ove gramatike dobile naziv kontekstne gramatike. Vidi se da se neterminal  $A$  zamenjuje nizom  $r$  samo ako se nađe u kontekstu reči  $\mathbf{x}$  i  $\mathbf{y}$ .

### 2.7.2 Normalne forme za beskonteksne gramatike

Svaka beskonteksna gramatika  $\mathbf{G}$  ima ekvivalentnu gramatiku  $\mathbf{G}_1$  u kojoj svaka smena ima jedan od sledećih oblika:

$$S \rightarrow \varepsilon, \quad A \rightarrow BC, \quad A \rightarrow a \quad A, B \in \mathbf{V}_n, a \in \mathbf{V}_t.$$

Ova normalna forma se naziva i Normalna forma Čomskog.

Svaka beskonteksna gramatika  $\mathbf{G}$  ima ekvivalentnu gramatiku  $\mathbf{G}_1$  u kojoj svaka smena ima jedan od sledećih oblika:

$$S \rightarrow \varepsilon, \quad A \rightarrow a, \quad A \rightarrow aB, \quad A \rightarrow aBC \quad A, B, C \in \mathbf{V}_n, a \in \mathbf{V}_t.$$

## 2.8 Pitanja

1. Definirati pojam azbuke.
2. Dati formalnu definiciju pojma reči.
3. Navesti osnovne delove reči.
4. Navesti osnovne operacije koje se mogu izvršavati nad rečima.
5. Dati formalnu definiciju pojma jezik.
6. Dati primere jezika definisanih nad azbukom  $\mathbf{V} = \{a, b, c, d, 0, 1\}$ .
7. Navesti osnovne operacije nad jezicima.
8. Šta su to formalne gramatike i kako se definišu?
9. Navesti osnovne tipove gramatika po Čomskom.
10. Šta su to kontekstne, a šta beskonteksne gramatike?
11. Šta su to normalne forme gramatika?

## 2.9 Zadaci

1. Odrediti skup  $\mathbf{V}^*$  azbuke  $\mathbf{V} = \{\mathbf{alo}, \mathbf{hula}\}$ .
2. Za datu reč  $\mathbf{w} = \mathbf{go}$  odrediti  $\mathbf{w}^2$  i  $\mathbf{w}^3$ .

3. Neka je  $\mathbf{L}_1 = \{\mathbf{ja, ti, on, ona, mi, vi, oni}\}$ , a  $\mathbf{L}_2 = \{\mathbf{spava, radi}\}$ . Odrediti jezike  $\mathbf{L}_1^{\mathbf{R}}$  i  $\mathbf{L}_1 \circ \mathbf{L}_2$ .
4. Gramatika  $\mathbf{G}$  je zadata sledećim skupom smena:

$$\begin{aligned} A &\rightarrow \mathbf{cifra} B \mid +C \mid -C \\ B &\rightarrow \mathbf{cifra} B \mid , D \\ C &\rightarrow \mathbf{cifra} B \\ D &\rightarrow \mathbf{cifra} D \mid \varepsilon \end{aligned}$$

Kom tipu pripada ova gramatika? Dati izvođenje reči

$$+ \mathbf{cifra} \mathbf{cifra} , \mathbf{cifra} \mathbf{cifra}$$

koja pripada jeziku opisanom datom gramatikom. Prikazati sintaksno stablo koje odgovara datom izvođenju.

5. Gramatika  $\mathbf{G}$  je zadata sledećim skupom smena:

$$\begin{aligned} S &\rightarrow S + I \mid I \\ I &\rightarrow (S) \mid \mathbf{fun}(S) \mid \mathbf{id} \end{aligned}$$

Odrediti tip gramatike i dati primere nekih rečeničnih formi i fraza za neterminal  $S$ .





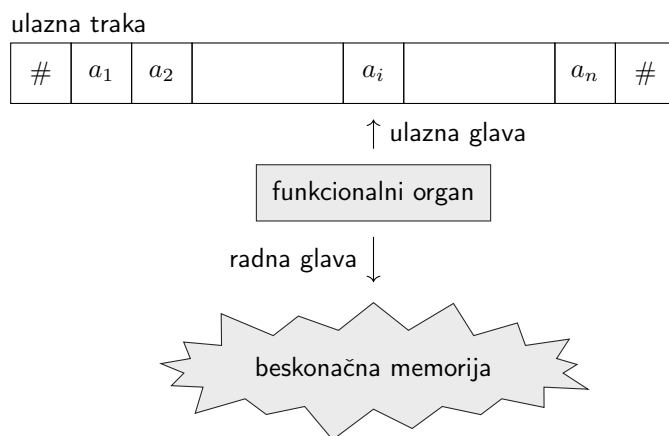
## Glava 3

# Automati kao uređaji za prepoznavanje jezika

Problem prepoznavanja jezika, odnosno utvrđivanja da li zadata reč pripada jeziku opisanom određenom gramatikom, rešava se automatima kao uređajima za prepoznavanje jezika. U ovom poglavlju biće reči o osnovnim karakteristikama četiri osnovna tipa automata i biće razmatrana njihova veza sa formalnim gramatikama i jezicima koje mogu da prepoznaju.

### 3.1 Opšti model automata

Generalno gledano, svi automati se mogu predstaviti šemom datom na slici 3.1.



SLIKA 3.1: Šema automata kao uređaja za prepoznavanje jezika

Niz simbola (reč) koji se prepoznaje zapisan je na ulaznoj traci. Po ulaznoj traci se kreće ulazna glava koja u jednom trenutku čita jedan simbol sa trake. Ova glava se u opštem slučaju može kretati i napred i nazad (desno i levo).

Funkcionalni organ automata je njegov najznačajniji deo. U zavisnosti od toga kako je opisan jezik koji se prepoznaje, funkcionalni organ može da se nađe u nekom od konačnog broja različitih stanja. Funkcionalni organ beleži predistoriju prepoznavanja i za to koristi memoriju koja je u ovom opštem modelu predstavljena kao beskonačna memorija. Automat sa beskonačnom memorijom nije praktično izvodljiv tako da se realni automati razlikuju od ovog modela po tome što koriste neki određeni tip memorije.

Proces prepoznavanja reči započinje tako što se ulazna glava pozicionira na početni simbol reči koja se prepoznaje (levi graničnik #), a funkcionalni organ se nalazi u svom početnom stanju. U svakom koraku prepoznavanja, u zavisnosti od toga gde je pozicionirana ulazna glava (koji simbol čita), stanja funkcionalnog organa, kao i simbola u radnoj memoriji koji čita radna glava, događaju se sledeće promene u automatu:

- menja se stanje funkcionalnog organa,
- u radnu memoriju se upisuje jedan ili više simbola,
- radna glava se pomera napred, nazad ili ostaje na poziciji na kojoj se nalazi,
- ulazna glava se pomera za jedno mesto napred ili nazad ili ostaje na poziciji na kojoj se nalazi.

Proces prepoznavanja reči je uspešno završen u trenutku kada se ulazna glava pozicionira na krajnji simbol reči koja se prepoznaje (desni graničnik #) i funkcionalni organ se nađe u jednom od svojih završnih stanja.

U opštem slučaju automati se mogu podeliti na četiri klase:

- 2N – dvosmerni nedeterministički,
- 1N – jednosmerni nedeterministički,
- 2D – dvosmerni deterministički i
- 1D – jednosmerni deterministički.

Podela na jednosmerne i dvosmerne izvršena je na osnovu mogućnosti pomerenja ulazne glave. Ako se ulazna glava kreće samo napred onda su to jednosmerni automati, a u slučaju kada je dozvoljeno i vraćanje glave unatrag onda su to dvosmerni automati. Podela na determinističke i nedeterminističke izvršena je u odnosu na preslikavanje stanja funkcionalnog organa koje se vrši u toku prepoznavanja. Ukoliko je stanje u koje prelazi automat, za određeni ulazni simbol i određeno zatečeno stanje, jednoznačno definisano onda je to deterministički automat. Ukoliko je ovo preslikavanje višeznačno onda je to nedeterministički automat.

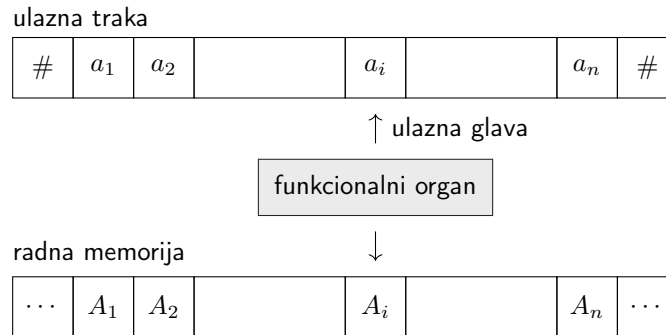
Postoje četiri osnovna tipa automata, što donekle odgovara podeli formalnih gramatika i jezika na tipove. Ova četiri tipa automata su:

- Tjuringova mašina,
- linearno ograničeni automat,
- magacinski automat i
- konačni automat.

Osnovna razlika između ovih tipova je u organizaciji memorije koja se koristi za beleženje predistorije prepoznavanja, što uslovljava i tipove jezika koji se mogu prepoznavati određenim tipom automata.

## 3.2 Tjuringova mašina

Slika 3.2 prikazuje šemu Tjuringove mašine kao osnovnog automata za prepoznavanje jezika. Za razliku od opšteg modela automata Tjuringova mašina ima radnu memoriju u vidu beskonačne trake.



SLIKA 3.2: Model Tjuringove mašine

2N Tjuringova mašina se može opisati torkom

$$T_m = (\mathbf{Q}, \mathbf{V}, \mathbf{S}, \mathbf{P}, q_0, b, \mathbf{F}),$$

gde je:

- $\mathbf{Q}$  – skup stanja operativnog organa,
- $\mathbf{V}$  – skup simbola na ulaznoj traci,
- $\mathbf{S}$  – skup simbola na radnoj traci,
- $\mathbf{F}$  – skup završnih, krajnjih stanja (podskup skupa  $\mathbf{Q}$ ),
- $q_0$  – početno stanje operativnog organa,

- $b$  – oznaka za prazno slovo,
- $P$  – preslikavanje definisano sa

$$\mathbf{Q} \times (\mathbf{V} \cup \#) \times \mathbf{S} \rightarrow \{\mathbf{Q} \times (\mathbf{S} \setminus \{b\}) \times \{-1, 0, +1\} \times \{-1, 0, +1\}\}.$$

Ako je  $(q_j, B, d_1, d_2) \in P(q_i, a, A)$ , gde je  $q_j, q_i \in \mathbf{Q}$ ,  $a \in \mathbf{V}$ ,  $A \in \mathbf{S}$  i  $d_1, d_2 \in \{-1, 0, 1\}$ , odnosno ako ulazna glava čita simbol  $a$ , funkcionalni organ se nalazi u stanju  $q_i$ , a radna glava iznad simbola  $A$ , automat će preći u stanje  $q_j$ , na radnu traku će upisti simbol  $B$  i izvršiće se pomeranje ulazne glave i radne glave u skladu sa vrednostima  $d_1$  i  $d_2$ . Pri tome važi sledeća notacija: ako je  $d_i = 0$ , nema pomeranja odgovarajuće glave; za  $d_i = 1$  glava se pomera za jedno mesto napred (desno), a za  $d_i = -1$ , glava se pomera za jedno mesto nazad (levo).

Mašina započinje prepoznavanje tako što je ulazna glava pozicionirana na početnom graničnom simbolu, funkcionalni organ se nalazi u početnom stanju i radna traka je prazna. Prepoznavanje reči se završava uspešno ako se automat nađe u nekom od završnih stanja (element skupa  $\mathbf{F}$ ) u trenutku kada je ulazna glava na krajnjem graničnom simbolu  $\#$ .

Tjuringova mašina je deterministička ako se svaki trojka  $(q, a, X)$  gde je  $q \in \mathbf{Q}$ ,  $a \in \mathbf{V}$  i  $X \in \mathbf{S}$ , preslikava u najviše jedan element  $(p, Y, d_1, d_2)$ , pri čemu je  $p \in \mathbf{Q}$ ,  $Y \in \mathbf{S}$  i  $d_1, d_2 \in \{-1, 0, 1\}$ .

Dokazano je da su sve četiri klase Tjuringovih mašina (2N, 1N, 2D, 1D) međusobno ekvivalentne i svaka od njih prepoznaje jezike tipa nula. To u suštini znači da je za prepoznavanje bilo kog jezika tipa nula moguće definisati bilo koji tip Tjuringove mašine. Odnosno, ako je moguće definisati jedan tip Tjuringove mašine, onda je moguće definisati i ekvivalentne mašine ostalih tipova. Jedini problem je to što model Tjuringove mašine podrazumeva beskonačnu memoriju, što u praksi praktično nije izvodljivo. Zbog toga Tjuringova mašina ostaje samo teorijski model, a u praksi se koriste automati sa konačnim memorijama, što kao posledicu ima ograničenje jezika koji se mogu prepoznavati.

Može se reći da je pojam Tjuringove mašine ekvivalent pojmovima algoritam i jezici tipa nula. To znači da se sva algoritamska preslikavanja mogu opisati jezicima tipa nula, pri čemu je moguće i definisati odgovarajuće Tjuringove mašine za prepoznavanje takvih jezika.

### 3.3 Linearno ograničen automat

Linearno ograničeni automati su u suštini Tjuringove mašine sa konačnom trakom kao radnom memorijom. Naime, 2N Tjuringova mašina kod koje može da se odredi konstanta  $k$  takva da je dužina reči koja se upisuje na radnu traku najviše  $k$  puta veća od dužine reči koja se prepoznaje, naziva se linearno ograničeni automat.

Odnosno, ako se prepoznaje reč  $\mathbf{w} \in \mathbf{V}^*$ , gde je  $|\mathbf{w}| = n$ , na radnu traku se upisuje reč dužine  $r$ , pri čemu je  $r \leq n \cdot k$ .

Posledica ovog ograničenja je da linerano ograničeni automati ne prepoznaju sve jezike tipa nula. Odnosno, postoje jezici tipa nula za koje nije moguće definisati linearno ograničeni automat. Naime, dokazana su sledeća tvrđenja:

- Klase 2N i 1N linearno ograničenih automata su međusobno ekvivalentne i svaka od njih prepoznaje jezike tipa jedan. Drugim rečima, za svaki jezik tipa jedan može se definisati nedeterministički linearno ograničeni automat koji ga prepoznaje.
- Klase 2D i 1D linearno ograničenih automata su međusobno ekvivalentne. To znači da se za svaki 2D linearno ograničeni automat može definisati ekvivalentan 1D linearno ograničeni automat.
- Ekvivalentnost 2N i 2D linearno ograničenih automata do sada nije dokazana.
- Svaka klasa linearno ograničenih automata ne prepoznaje sve jezike tipa jedan.

### 3.4 Magacinski automat

Magacinski automat kao radnu memoriju koristi memoriju organizovanu u vidu magacina. Šema strukture ovog automata data je na slici 3.3.

2N Magacinski automat je definisan torkom  $\text{Ma} = (\mathbf{Q}, \mathbf{V}, \mathbf{S}, \mathbf{P}, q_0, Z_0, \mathbf{F})$  gde je:

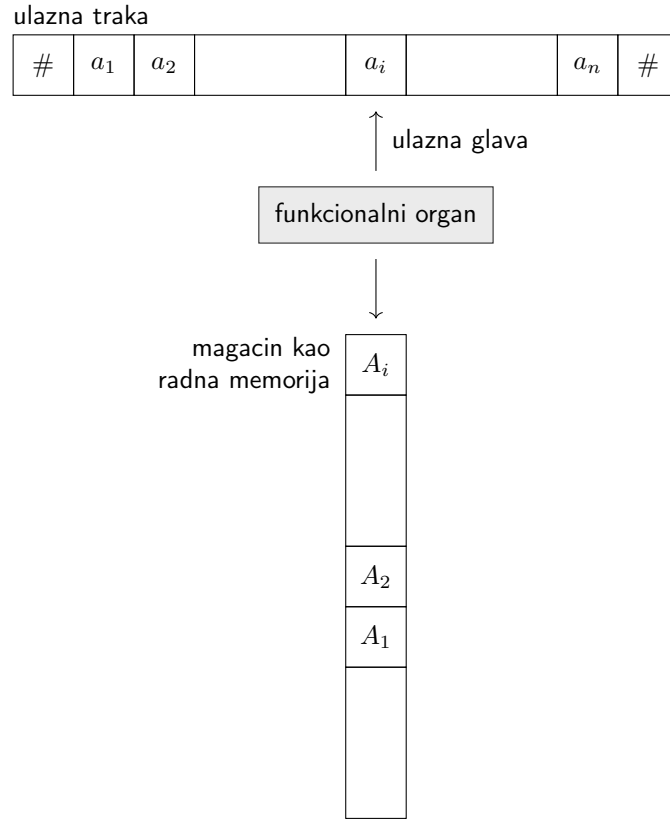
- $\mathbf{Q}$  – skup stanja operativnog organa,
- $\mathbf{V}$  – skup simbola na ulaznoj traci,
- $\mathbf{S}$  – skup magacinskih simbola,
- $\mathbf{F}$  – skup završnih, krajnjih stanja (podskup skupa  $\mathbf{Q}$ ),
- $q_0$  – početno stanje operativnog organa,
- $Z_0$  – početni simbol na vrhu magacina,
- $\mathbf{P}$  – preslikavanje definisano sa:

$$\mathbf{Q} \times (\mathbf{V} \cup \#) \times \mathbf{S} \rightarrow \{\mathbf{Q} \times \mathbf{S}^* \times \{-1, 0, +1\}\}.$$

Za opis konfiguracije magacinskog automata u određenom trenutku obično se koristi sledeća struktura:

$$(q, \#a_1a_2 \cdots \bar{a}_i \cdots a_n\#, X_1X_2 \cdots X_m)$$

$$q \in \mathbf{Q}, a_i \in V, i = 1, 2, \dots, n \wedge X_j \in S, j = 1, 2, \dots, m,$$



SLIKA 3.3: Magacinski automat

koju čine stanje u kome se nalazi funkcionalni organ, niz na ulaznoj traci čije se prepoznavanje vrši i niz simbola u radnom magacinu. Pri tome je sa  $X_m$  označen simbol koji se nalazi u vrhu magacina.

Ako je  $(p, \mathbf{r}, d) \in P(q, a_i, X_m)$ , gde su  $p, q \in \wedge Q$ ,  $a_i \in V$ ,  $\mathbf{r} \in \mathbf{S}^*$  i  $d \in \{-1, 0, 1\}$ , u automatu se događa sledeće preslikavanje:

$$(q, \#a_1a_2 \cdots \overline{a_i} \cdots \#, X_1X_2 \cdots X_m) \mapsto (p, \#a_1a_2 \cdots \overline{a_i} + d \cdots a_n\#, X_1X_2 \cdots X_{m-1}, \mathbf{r})$$

Napomenimo da se u magacin upisuje reč od jednog slova, više slova ili se ne upisuje ništa kada je  $\mathbf{r} = \varepsilon$ .

### Primer 3.1 Magacinski automat

Neka je data gramatika  $\mathbf{G} = (\{0, 1, a\}, \{S\}, S, \mathbf{P})$ , sa skupom smena  $\mathbf{P}$ :

$$S \rightarrow 0S1 \qquad S \rightarrow a.$$

Prepoznavanje jezika definisanog gramatikom  $\mathbf{G}$  može da se izvrši magacinskim automatom koji je opisan sa:

$$\mathbf{Ma} = (\mathbf{Q}, \mathbf{V}, \mathbf{S}, \mathbf{P}, q_0, Z_0, \mathbf{F}),$$

gde je:

$$\begin{aligned} \mathbf{Q} &= \{q_0, q_1\}, & \mathbf{V} &= \{0, 1, a\}, & \mathbf{S} &= \{A, B\}, \\ Z_0 &= A, & \mathbf{F} &= q_1; \end{aligned}$$

a skup smena  $P$  ima sledeće elemente:

$$\begin{aligned} (q_0, \#, A) &\rightarrow (q_0, A), & (q_0, 0, A) &\rightarrow (q_0, BA), \\ (q_0, a, A) &\rightarrow (q_1, \varepsilon), & (q_1, 1, B) &\rightarrow (q_1, \varepsilon). \end{aligned}$$

Početno stanje automata je  $q_0$ , a kraj preslikavanja definisan je sa  $(q_1, \#, \varepsilon)$ .

Pokazaćemo prepoznavanje dve karakteristične reči jezika definisanog gramatikom  $\mathbf{G}$ .

1. Prepoznavanje reči  $\#000a111$ :

$$\begin{aligned} (q_0, \overline{\#}000a111\#, A) &\rightarrow (q_0, \#000a111\#, A) \\ &\rightarrow (q_0, \#000a111\#, BA) \\ &\rightarrow (q_0, \#000a111\#, BBA) \\ &\rightarrow (q_0, \#000a111\#, BBBA) \\ &\rightarrow (q_1, \#000a111\#, BBB) \\ &\rightarrow (q_1, \#000a111\#, BB) \\ &\rightarrow (q_1, \#000a111\#, B) \\ &\rightarrow (q_1, \#000a111\overline{\#}, \varepsilon). \end{aligned}$$

2. Prepoznavanje reči  $\#a\#$ :

$$(q_0, \overline{\#}a\#, A) \rightarrow (q_0, \#a\#, A) \rightarrow (q_1, \#a\overline{\#}, \varepsilon).$$

Magacinski automati su pogodni za prepoznavanje beskonteksnih jezika. Dokazana su sledeća tvrđenja:

- 1N magacinskim automatima prepoznaju se samo beskontekсни jezici tipa dva.

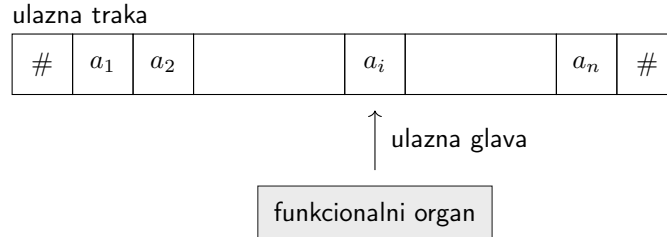
- 2N magacinskim automatima mogu se prepoznavati svi beskontesni i neki (ali ne svi) kontekсни jezici.
- Postoje beskontekсни jezici koji se ne mogu prepoznati 1D magacinskim automatima.
- 2D magacinski automati pored beskonteksnih mogu da prepoznaju i neke konteksne jezike.

Magacinski automati se koriste u sintaksoj analizi programskih jezika.

### 3.5 Konačni automati

Konačni automati su uređaji za prepoznavanje regularnih jezika. Za razliku od drugih tipova nemaju radnu memoriju već se predistorija preslikavanja pamti samo preko promene stanja funkcionalnog organa (slika 3.4).

Kod dvosmernih konačnih automata, na osnovu stanja operativnog organa i simbola koji čita ulazna glava, menja se stanje automata i ulazna glava pomera za jedno mesto ulevo, udesno ili ostaje na svom mestu. Kod jednosmernih, menja se samo trenutno stanje automata.



SLIKA 3.4: Šema konačnog automata

Jednosmeran deterministički konačan automat je opisan torkom  $Ka = (Q, V, P, q_0, F)$ , gde je:

- $Q$  – skup stanja operativnog organa,
- $V$  – skup simbola na ulaznoj traci,
- $F$  – skup završnih, krajnjih stanja (podskup skupa  $Q$ ),
- $q_0$  – početno stanje operativnog organa,
- $P$  – preslikavanje pravilima oblika:  $Q \times \{V \cup \#\} \rightarrow Q$ .

Za ulazni niz  $w \in V^*$  kažemo da je prepoznat automatom ako automat prevodi iz početnog u neko od njegovih krajnjih stanja, odnosno ako je  $P(q_0, w) \in F$ .



Skup svih reči koje prepoznaje  $Ka$  nazivamo regularnim i označavaćemo ih sa  $\mathbf{T}(Ka)$ . Važi:

$$\mathbf{T}(Ka) = \{\mathbf{x} \mid \mathbf{P}(q_0, \mathbf{x}) \in \mathbf{F}\}.$$

**Primer 3.2** Konačni automat

Neka je  $Ka = (\mathbf{Q}, \mathbf{V}, \mathbf{P}, q_0, \mathbf{F})$ , definisan na sledeći način:

$$\mathbf{Q} = \{q_0, q_1, q_2, q_3\},$$

$$\mathbf{V} = \{0, 1\},$$

$$\mathbf{F} = \{q_0\},$$

a skup  $\mathbf{P}$  ima sledeće elemente:

$$P(q_0, 0) = q_2,$$

$$P(q_0, 1) = q_1,$$

$$P(q_1, 0) = q_3,$$

$$P(q_1, 1) = q_0,$$

$$P(q_2, 0) = q_0,$$

$$P(q_2, 1) = q_3,$$

$$P(q_3, 0) = q_1,$$

$$P(q_3, 1) = q_2.$$

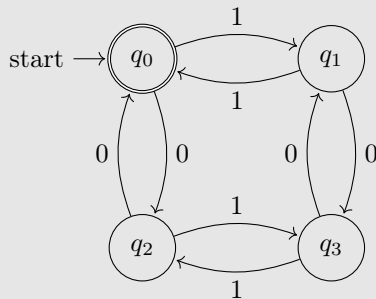
Može se pokazati da ovaj konačni automat prepoznaje reči koje se sastoje od parnog broja jedinica i parnog broja nula.

Za predstavljanje automata se koriste grafovi, pri čemu čvorovi grafa odgovaraju stanjima automata, a potezi preslikavanjima. Na Slici 3.4 predstavljen je graf koji odgovara automatu iz primera. Početno stanje automata je označeno strelicom, a krajnje stanje sa dva koncentrična kruga.

Kako se preslikavanje  $P(q_i, a) = q_j$  može tumačiti da se stanje  $q_i$  automata za ulazno slovo  $a$  preslikava u stanje  $q_j$ , svako takvo preslikavanje se na grafu predstavlja odlaznim potegom od stanja  $q_i$  do stanja  $q_j$  na kome je oznaka  $a$ .

**Primer 3.3** Graf automata

Graf automata iz primera 3.2 prikazan je na sledećoj slici.



### 3.5.1 Određivanje jezika koji se prepoznaje datim konačnim automatom

Jezik koji prepoznaje konačni automat  $K_a$  može da se odredi analitički kao skup svih reči koje početno stanje preslikavaju u krajnja stanja (kao događaji krajnjih stanja u funkciji početnog stanja). Počinje se tako što se za svako od stanja automata (svaki čvor grafa automata) pišu jednačine stanja. Za svaki čvor grafa automata, ove jednačine su određene dolaznim potezima u taj čvor. Ako postoji preslikavanje  $P(q_i, a) \rightarrow q_j$ , znači u grafu automata postoji poteg iz čvora  $q_i$  ka čvoru  $q_j$  za ulazno slovo  $a$ . To se predstavlja jednačinom stanja

$$q_j = q_i a.$$

Da bi se odredio jezik koji prepoznaje automat potrebno za svako od završnih stanja odrediti skup reči koje početno stanje prevode u to stanje, odnosno rešavanjem jednačina stanja odrediti

$$q_{\text{kraj}} = f(q_{\text{poč}}).$$

Pri rešavanju jednačina stanja za njihovo transformisanje može se koristiti sledeći model:

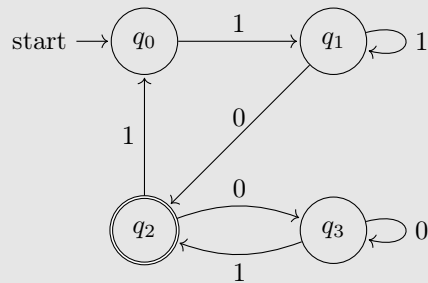
$$q_i = q_i a \cup q_j b \Rightarrow q_i = q_j b a^*$$

jer je:

$$\begin{aligned} q_i a^* \setminus q_i a a^* &= q_j b a^* \\ q_i \{\varepsilon \cup a \cup a^2 \cup a^3 \cup \dots\} \setminus q_i a \{\varepsilon \cup a \cup a^2 \cup a^3 \cup \dots\} &= q_j b a^* \\ q_i &= q_j b a^* \end{aligned}$$

#### Primer 3.4 Određivanje jezika

Zadatak je da se za automat predstavljen grafom sa sledeće slike odredi jezik koji prepoznaje.



Postavljamo jednačine stanja za svako od stanja automata:

1.  $q_0 = q_0 \varepsilon \cup q_2 1$
2.  $q_1 = q_1 1 \cup q_0 1$
3.  $q_2 = q_1 0 \cup q_3 1$
4.  $q_3 = q_2 0 \cup q_3 0$

Napomena: Kako je  $q_0$  početno stanje, njegovoj jednačini stanja se dodaje i  $q_0 \varepsilon$ .

Krajnje stanje automata je  $q_2$ , da bi odredili sve reči koje početno stanje preslikavaju u ovo stanje potrebno je da se na osnovu jednačina stanja odredi  $q_2 = f(q_0)$ :

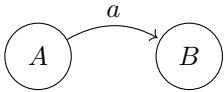
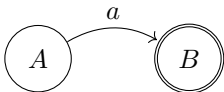
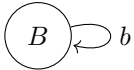
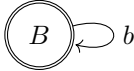
$$\begin{array}{ll} 4 & q_3 = q_2 0 0^* \\ 4' \rightarrow 3 & q_2 = q_1 0 \cup q_2 0 0^* 1 \end{array} \quad (3')$$

$2'$	$q_1 = q_0 11^*$	
$2' \rightarrow 3'$	$q_2 = q_0 11^* 0 \cup q_2 00^* 1$	$(3'')$
$1 \rightarrow 3''$	$q_2 = q_0 \varepsilon 11^* 0 \cup q_2 111^* 0 \cup q_2 00^* 1$	
	$q_2 = q_0 \varepsilon 11^* (111^* 0 \cup 00^* 1)^*$	
	$\Rightarrow L(G) = 11^* 0 (111^* 0 \cup 00^* 1)^*$	

### 3.5.2 Veza između konačnih automata i gramatika tipa tri

Pokazali smo da se jednostavno može odrediti jezik koji se prepoznaje zadanim konačnim automatom. Važi i obrnuto, za svaki jezik definisan gramatikom tipa tri može se odrediti konačni automat koji ga prepoznaje. Pravila koja se primenjuju za ova preslikavanja data su u tabeli 3.1. Svskom neterminalnom simbolu gramatike odgovara jedno stanje automata, pri čemu startnom simbolu odgovara početno stanje. Skupu terminalnih simbola gramatike odgovara skup ulaznih simbola automata a svakom pravilu gramatike odgovara preslikavanje u automatu.

TABELA 3.1: Veze između gramatika tipa tri i konačnih automata

AUTOMAT	GRAMATIKA
skup stanja $\mathbf{Q}$	skup neterminalnih simbola $\mathbf{V}_n$
ulazna azbuka $\mathbf{V}$	skup terminalnih simbola $\mathbf{V}_t$
početno stanje $q_0$	startni simbol
skup prslikavanja $\mathbf{P}$	skup pravila $\mathbf{P}$
	$a \rightarrow aB$
	$a \rightarrow aB$ i $b \rightarrow \varepsilon$
	$b \rightarrow bB$
	$b \rightarrow bB$ i $B \rightarrow \varepsilon$ ili $b \rightarrow bB$ i $B \rightarrow b$

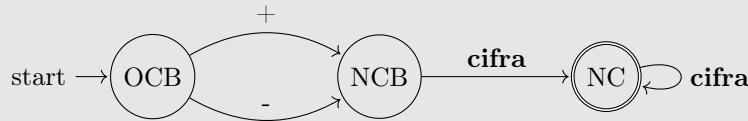
Sledeći primeri ilustruju primenu ovih pravila.

**Primer 3.5** Preslikavanje gramatike u konačni automat

Definisati konačni automat za prepoznavanje označenih celih brojeva. Označeni ceo broj definisan je sledećom gramatikom:

1.  $\langle \text{OznaceniCeoBroj} \rangle \rightarrow + \langle \text{NeoznaceniCeoBroj} \rangle$
2.  $\langle \text{OznaceniCeoBroj} \rangle \rightarrow - \langle \text{NeoznaceniCeoBroj} \rangle$
3.  $\langle \text{NeoznaceniCeoBroj} \rangle \rightarrow \text{cifra} \langle \text{NizCifara} \rangle$
4.  $\langle \text{NizCifara} \rangle \rightarrow \text{cifra} \langle \text{NizCifara} \rangle$
5.  $\langle \text{NizCifara} \rangle \rightarrow \varepsilon$

Graf automata koji prepoznaje jezik definisan gramatikom dat je na sledećoj slici.

**Primer 3.6** Preslikavanje grafa u gramatiku

Na osnovu grafa automata prikazanog u primeru 3.3 može se odrediti sledeća gramatika koja opisuje jezik koji se prepoznaje ovim automatom.

$$\mathbf{V}_n = \{Q_0, Q_1, Q_2, Q_3\}$$

$$\mathbf{V}_t = \{0, 1\}$$

$$S = Q_0$$

Skup smena **P**:

$Q_0 \rightarrow 1Q_1$	$Q_1 \rightarrow 1Q_1$
$Q_1 \rightarrow 0Q_2$	$Q_2 \rightarrow 1Q_0$
$Q_2 \rightarrow 0Q_3$	$Q_2 \rightarrow \varepsilon$
$Q_3 \rightarrow 0Q_3$	$Q_3 \rightarrow 1Q_2$

**Primer 3.7** Preslikavanje gramatike u graf automata i određivanje jezika

Odrediti kom tipu pripada i koji jezik prepoznaje gramatika zadata slede-

čim skupom smena:

$$\begin{aligned} A &\rightarrow 0B \mid 1D, & B &\rightarrow 0C \mid 1F, & C &\rightarrow 0C \mid 1F \mid \varepsilon, \\ D &\rightarrow 1E \mid 0F, & E &\rightarrow 1E \mid 0F, & F &\rightarrow 0G \mid 1G, \\ G &\rightarrow 0G \mid 1G \mid \varepsilon. \end{aligned}$$

Za svaku smenu gramatike važi da je dužina reči na levoj strani smene manja ili jednaka dužini reči na desnoj strani – zadovoljen je uslov da gramatika može biti tipa 1.

Na levoj strani svih smena je neterminalni simbol – gramatika može biti tipa 2.

Sve smene su oblika  $X \rightarrow \mathbf{a}$ , ili  $X \rightarrow \mathbf{a}Y$  ili  $X \rightarrow \varepsilon$  – gramatika je tipa 3.

Kako je data gramatika tipa 3, jezik koji je definisan gramatikom se može predstaviti u obliku regularnog izraza. Da bismo odredili regularni izraz jezika, crtamo najpre graf konačnog automata koji taj jezik prepoznaje. Ovaj graf dat je na slici na kraju primera.

Jednačine stanja automata su:

$$A = A\varepsilon \tag{1}$$

$$B = A0 \tag{2}$$

$$C = B0 + C0 \tag{3}$$

$$D = A1 \tag{4}$$

$$E = D1 + E1 \tag{5}$$

$$F = B1 + C1 + E0 \tag{6}$$

$$G = F0 + F1 + G0 + G1. \tag{7}$$

Treba izraziti sva završna stanja u funkciji ulaznog stanja.

Iz jednačina (1) i (2) sledi:

$$B = A\varepsilon 0. \tag{8}$$

Iz (3) i (8) imamo  $C = A00 + C0$ , tj.

$$C = A\varepsilon 000^*. \tag{9}$$

Iz (1) i (4) dobijamo:

$$D = A\varepsilon 1. \tag{10}$$

Iz (5) i (10) imamo  $E = A11 + E1$ , tj.

$$E = A111^*. \quad (11)$$

Zamenom (8), (9) i (11) u (6) dobija se:

$$\begin{aligned} F &= A01 + A000^*1 + A111^*0 \\ &= A(01 + 000^*1 + 111^*0). \end{aligned} \quad (12)$$

Iz (7) sledi:

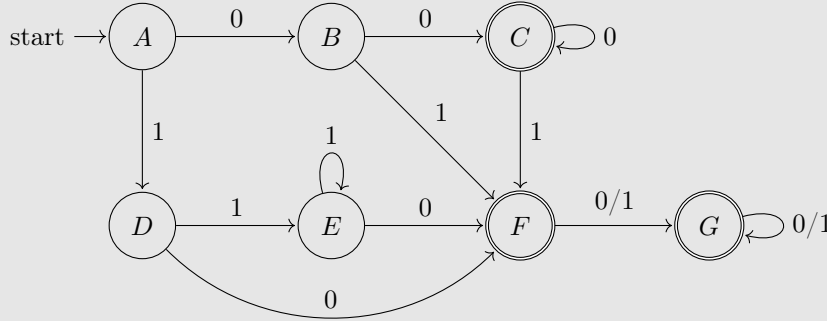
$$\begin{aligned} G &= F(0 + 1) + G(0 + 1) \\ &= F(0 + 1)(0 + 1)^*. \end{aligned} \quad (13)$$

Zamenom (12) u (13) dobijamo:

$$G = A(01 + 000^*1 + 111^*0)(0 + 1)(0 + 1)^*.$$

Konačno, jezik koji je definisan gramatikom je unija jezika koje prepoznaju stanja  $C$ ,  $F$  i  $G$ :

$$L = 000^* + (01 + 000^*1 + 111^*0) + (01 + 000^*1 + 111^*0)(0 + 1)(0 + 1)^*. \quad (3.1)$$



## 3.6 Regularni izrazi

Jezici tipa tri koji se prepoznaju konačnim automatima su regularni jezici zato što se mogu opisati regularnim izrazima. Ovde ćemo posvetiti malo više pažnje pojmu regularni izrazi. Oni se mogu posmatrati i kao meta jezici za predstavljanje formalnih jezika. Regularni izrazi u suštini definišu skup reči jezika. Sami regularni izrazi su nizovi formirani od slova neke azbuke  $\mathbf{V}$  i skupa specijalnih simbola. Regularni izraz skraćeno opisuje skup reči jezika definisanog nad azbu-

kom  $\mathbf{V}$ . Postoje posebna pravila kako se vrši tumačenje (denotacija) regularnog izraza i generisanje skupa reči jezika koji je opisan regularnim izrazom.

Regularni izraz nad azbukom  $\mathbf{V}$  definiše se sledećim skupom pravila:

1.  $\emptyset$  (prazan skup) je regularni izraz.
2.  $\varepsilon$  je regularni izraz.
3. Ako je  $a \in \mathbf{V}$ , onda je  $a$  regularni izraz.
4. Ako su  $\mathbf{r}_1$  i  $\mathbf{r}_2$  regularni izrazi, onda su i  $(\mathbf{r}_1 \mid \mathbf{r}_2)$  i  $(\mathbf{r}_1 \circ \mathbf{r}_2)$  regularni izrazi.
5. Ako je  $\mathbf{r}$  regularni izraz onda je i  $\mathbf{r}^*$  takođe regularni izraz.
6. Nema drugih regularnih izraza nad azbukom  $\mathbf{V}$ .

**Napomena:** U literaturi se često se umesto operatora  $\mid$  koristi  $+$ , a operator  $\circ$  se izostavlja.

#### Primer 3.8 Regularni izrazi

Neka je  $\mathbf{V} = \{a, b, c, \dots, x, y, z\}$ . Sledeći nizovi su regularni izrazi:

$$\emptyset, \quad a, \quad ((c \circ a) \circ t), \\ ((m \circ e) \circ (o)^* w), \quad (a \mid (e \mid (i \mid (o \mid u))))), \quad (a \mid (e \mid (i \mid (o \mid u))))^*.$$

Kada su regularni izrazi definisani, treba da se zna njihovo tumačenje, odnosno preslikavanje (denotacija) u skup reči. Za svaki regularni izraz  $\mathbf{r}$ , njegova denotacija, označena sa  $\|\mathbf{r}\|$ , je skup reči definisan sa:

1.  $\|\emptyset\|$  je prazan skup  $\emptyset$ .
2.  $\|\varepsilon\|$  je  $\{\varepsilon\}$ , skup singleton koji sadrži samo element  $\varepsilon$ .
3. Ako je  $a \in \mathbf{V}$ , onda je  $\|a\|$  skup  $\{a\}$ , skup singleton koji sadrži samo  $a$ .
4. Ako su  $\mathbf{r}_1$  i  $\mathbf{r}_2$  dva regularna izraza čija denotacija je  $\|\mathbf{r}_1\|$  i  $\|\mathbf{r}_2\|$ , respektivno, tada je  $\|(\mathbf{r}_1 \mid \mathbf{r}_2)\|$  jednako  $\|\mathbf{r}_1\| \cup \|\mathbf{r}_2\|$ , a  $\|(\mathbf{r}_1 \circ \mathbf{r}_2)\|$  jednako  $\|\mathbf{r}_1\| \circ \|\mathbf{r}_2\|$ .
5. Ako je  $\mathbf{r}$  regularni izraz čija je denotacija jednaka  $\|\mathbf{r}\|$ , onda je  $\|\mathbf{r}^*\|$  jednako  $\|\mathbf{r}\|^*$ .

#### Primer 3.9 Regularni izrazi i jezici

U sledećoj tabeli dato je tumačenje regularnih izraza iz primera 3.8.



REGULARNI IZRAZ	DENOTACIJA
$\emptyset$	$\emptyset$
$a$	$\{a\}$
$((c \circ a) \circ t)$	$\{cat\}$
$((m \circ e) \circ (o)^* \circ w)$	$\{mew, meow, meoow, meoooow, \dots\}$
$(a \mid (e \mid (i \mid (o \mid u))))$	$\{a, e, i, o, u\}$
$(a \mid (e \mid (i \mid (o \mid u))))^*$	skup svih reči koje se sastoje od samoglasnika, uključujući i $\varepsilon$

Regularni izrazi čine algebru, sa svojstvima operacija prikazanim u tabeli 3.2.

TABELA 3.2: Algebarska svojstva regularnih izraza

AKSIOM	OPIS
$\mathbf{r} \mid \mathbf{s} = \mathbf{s} \mid \mathbf{r}$	operacija $\mid$ je komutativna
$\mathbf{r}(\mathbf{s} \mid \mathbf{t}) = (\mathbf{r} \mid \mathbf{s}) \mid \mathbf{t}$	operacija $\mid$ je asocijativna
$\mathbf{s}(\mathbf{st}) = (\mathbf{rs})\mathbf{t}$	nadovezivanje je asocijativna operacija
$\mathbf{r}(\mathbf{s} \mid \mathbf{t}) = \mathbf{rs} \mid \mathbf{rt}$ $(\mathbf{s} \mid \mathbf{t}) = \mathbf{sr} \mid \mathbf{tr}$	distributivnost nadovezivanja u odnosu na $\mid$
$\mathbf{r} = \mathbf{r}$ $\mathbf{r}\varepsilon = \mathbf{r}$	$\varepsilon$ je jedinični element z operaciju nadovezivanja
$\mathbf{r}^* = (\mathbf{r} \mid \varepsilon)^*$	relacija između $*$ i $\varepsilon$
$\mathbf{r}^{**} = \mathbf{r}^*$	iteracija je idempotentna operacija

Operator  $*$  je najvišeg prioriteta. Operator nadovezivanja je sledeći po prioritetu, dok je operator  $\mid$  najnižeg prioriteta. Svi ovi operatori su levo asocijativni.

Regularni izrazi su veoma korisni za formalno i koncizno definisanje jezika. Kada je jezik konačan moguće je nabrojati sve reči koje sadrži, ali u slučaju beskonačnih jezika to postaje problem. Zbog toga se regularni izrazi i koriste u različitim disciplinama kao sredstvo za opis kompleksnih jezika. Na žalost, ne mogu se svi formalni jezici opisati regularnim izrazima. To je moguće samo u slučaju jezika definisanih gramatikama tipa tri, zbog čega se oni i nazivaju regularnim. Sledeći primer ilustruje moć regularnih izraza da definišu beskonačne i kompleksne jezike.

**Primer 3.10** Jezici definisani regularnim izrazima

Neka je data azbuka  $V$  koja sadrži sva slova engleske abecede

$$V = \{a, b, c, \dots, x, y, z\}.$$

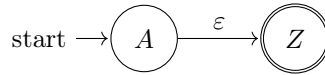
- Regularnim izrazom  $V^*$  definisan je skup svih reči koje se mogu napraviti od slova skupa  $V$ .
- Skup svih reči koje sadrže samoglasnike može se definisati regularnim izrazom  $V^* \circ (a + e + i + o + u) \circ V^*$ .
- Skup svih reči koje počinju sa *non* je:  $(non) \circ V^*$ .
- Skup svih reči koje se završavaju na *tion* ili *sion* definisan izrazom:  $V^*(t + s) \circ (ion)$ .

Regularni izrazi ne mogu da se koriste za opis izbalansiranih ili umetnutih konstrukcija. Na primer, izrazi sa zagradama u kojima se uvek zagrade javljaju u paru ne mogu da budu opisani regularnim gramatikama, ali se zato jednostavno opisuju beskontesnim gramatikama; na primer, pravilom  $E \rightarrow (E)$ .

### 3.7 Preslikavanje regularnih izraza u konačne automate

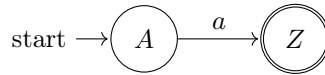
Za svaki regularni izraz se može definisati konačni automat koji ga prepoznaje. Pri tome važe sledeća pravila:

Regularni izraz  $r = \varepsilon$  se prepoznaje automatom datim na slici 3.5, gde je  $A$  početno stanje automata a  $Z$  završno stanje automata.



SLIKA 3.5: Automat za regularni izraz  $r = \varepsilon$

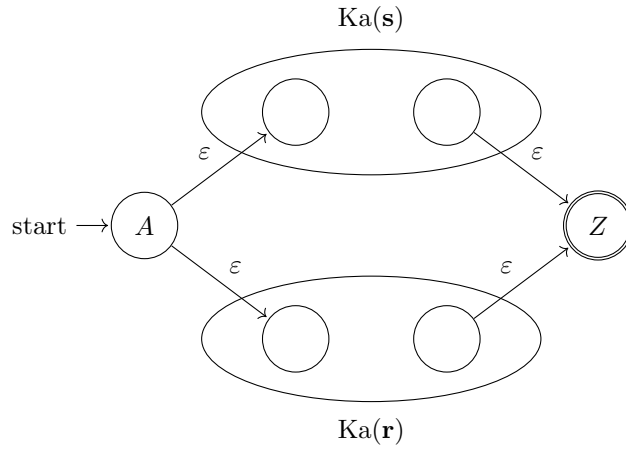
Regularnom izrazu  $r = a$  odgovara automat sa slike 3.6.



SLIKA 3.6: Automat za regularni izraz  $r = a$

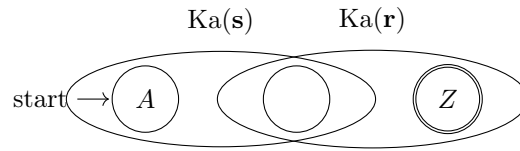
Regularni izraz  $r + s$  ili  $r \mid s$  se prepoznaje automatom koji ima dve paralelne grane, jednu u kojoj je automat za izraz  $r$  i drugu u kojoj je automat za izraz  $s$ , prikazan na slici 3.7.

### 3.7. PRESLIKAVANJE REGULARNIH IZRAZA U KONAČNE AUTOMATE 53



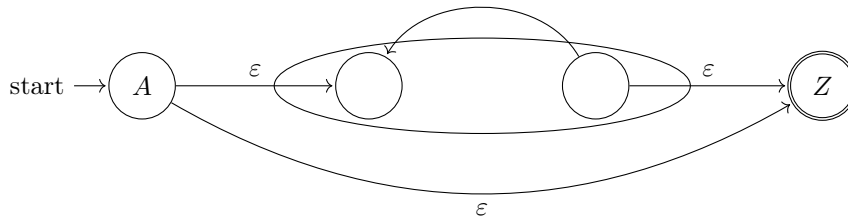
SLIKA 3.7: Automat za regularni izraz  $r \mid s$

Automat za regularni izraz  $s \circ r$ , gde su  $s$  i  $r$  regularni izrazi, realizuje se kao redna veza automata koji prepoznaje izraz  $s$  i automata koji prepoznaje izraz  $r$  (slika 3.8).



SLIKA 3.8: Automat za regularni izraz  $r \circ s$

Regularni izraz  $(s)^*$  se prepoznaje automatom sa petljom koja spaja završno i početno stanje stanje podautomata za regularni izraz  $s$ , kao što je prikazano na slici 3.9.

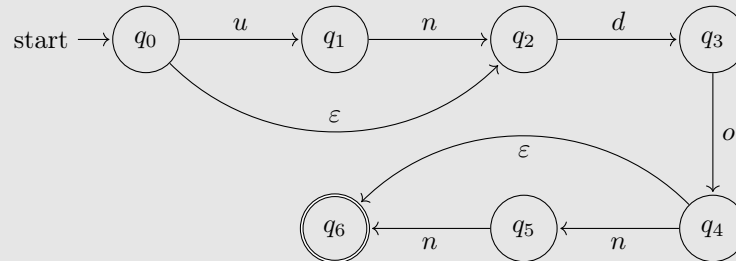


SLIKA 3.9: Automat za regularni izraz  $(s)^*$

Kako se u prethodnim primerima automata pojavljuje puno potega označenih sa  $\epsilon$ , sledeći primer ilustruje njihovo tumačenje i značaj.

**Primer 3.11**  $\varepsilon$ -grane u automatu

Automat sa sledeće slike prepoznaje jezik  $\{do, undo, undone, done\}$ .



Uočimo da i u ovom slučaju  $\varepsilon$  zamenjuje praznu reč. Na primer, kada automat prepoznaje reč *undo*, iz početnog stanja dolazi u stanje  $q_4$  i prazna reč ga iz stanja  $q_4$  prevodi u završno stanje  $q_6$ .

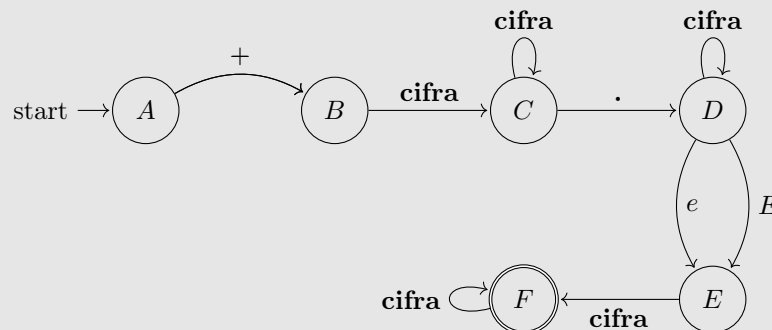
Primenom navedenih pravila može se generisati konačni automat za bilo koji regularni izraz.

**Primer 3.12** Regularni izrazi u konačni automat

Označeni realni brojevi u eksponencijalnom zapisu definisani su sledećim regularnim izrazom:

$$(+ | -) \text{ cifra cifra}^* \cdot \text{cifra}^* (E | e) (+ | -) \text{ cifra cifra}^*.$$

Za prepoznavanje ovih brojeva može se koristiti automat sa sledeće slike.

**Primer 3.13** Primeri konačnih automata

U sledećoj tabeli su dati grafovi konačnih automata koji prepoznaju regularne izraze iz primera 3.8.

REGULARNI IZRAZ	GRAF AUTOMATA
$\emptyset$	
$a$	
$((c \circ a) \circ t)$	
$((m \circ e) \circ (o)^* \circ w)$	
$(a \mid (e \mid (i \mid (o \mid u))))$	
$(a \mid (e \mid (i \mid (o \mid u))))^*$	

### 3.8 Pitanja

1. Šta su to automati? Objasniti kako teče proces prepoznavanja reči automatom.
2. Kako je definisana Tjuringova mašina?
3. Koje jezike prepoznaju automati tipa Tjuringove mašine?
4. Šta su to linearno ograničeni automati i koje jezike prepoznaju?
5. Kako je definisam magacinski automat i koje jezike prepoznaje?
6. Dati definiciju konačnih automata.
7. Objasniti vezu između konačnih automata i jezika tipa tri.
8. Dati definiciju regularnih izraza. Navesti nekoliko primera regularnih izraza.
9. Šta je to denotacija regularnih izraza? Dati primere za denotaciju.
10. Pokazati kako se osnovni regularni izrazi preslikavaju u automate.

### 3.9 Zadaci

1. Gramatika  $\mathbf{G}$  zadata je sledećim skupom smena:

$$S \rightarrow aSdd \mid aAd,$$

$$A \rightarrow bAc \mid b.$$

Odrediti tip gramatike i automat koji prepoznaje reči jezika definisanog datom gramatikom. Pokazati proces prepoznavanja reči  $\#aaabbcd\#$ .

2. Gramatika **G** zadata je sledećim skupom smena:

$$\begin{aligned} A &\rightarrow 0C \mid 1B, & B &\rightarrow 0B \mid 1C, \\ C &\rightarrow 0A \mid 1D, & D &\rightarrow 1D \mid \varepsilon. \end{aligned}$$

Definisati mašinu za prepoznavanje reči jezika ove gramatike i odrediti jezik koji je definisan ovom gramatikom.

3. Gramatika **G** zadata je sledećim skupom smena:

$$\begin{aligned} A &\rightarrow \text{cifra } B \mid +C \mid -C, & B &\rightarrow \text{cifra } B \mid .D, \\ C &\rightarrow \text{cifra } B, & D &\rightarrow \text{cifra } D \mid \varepsilon. \end{aligned}$$

- (a) Definisati mašinu za prepoznavanje reči jezika ove gramatike.  
(b) Odrediti jezik ove gramatike.

4. Zadata je gramatika tipa tri sledećim skupom smena:

$$\begin{aligned} A &\rightarrow 0B \mid 1D, & B &\rightarrow 0C \mid 1F, \\ C &\rightarrow 0C \mid 1F \mid \varepsilon, & D &\rightarrow 1E \mid 0F, \\ E &\rightarrow 1E \mid 0F \mid \varepsilon, & F &\rightarrow 0G \mid 1G, \\ G &\rightarrow 0G \mid 1G \mid \varepsilon. \end{aligned}$$

Nacrtati graf odgovarajućeg konačnog automata i odrediti regularni izraz jezika ove gramatike.

5. Gramatika **G** zadata je sledećim skupom smena:

$$\begin{aligned} A &\rightarrow 0D \mid 1A, & B &\rightarrow 0A \mid 1C, & C &\rightarrow 0A \mid 1F, \\ D &\rightarrow 0B \mid 1C, & E &\rightarrow 0B \mid 1C \mid \varepsilon, & F &\rightarrow 0E \mid 1A \mid \varepsilon. \end{aligned}$$

- (a) Definisati mašinu za prepoznavanje reči jezika određenog gramatikom **G**.  
(b) Odrediti jezik koji je definisan gramatikom **G**.  
6. Definisati regularne izraze za jezike definisane nad azbukom

$$\mathbf{V} = \{a, b, c, \dots, x, y, z, 0, 1, \dots, 9\}$$

koje čine sledeći skupovi reči:

- (a) Sve reči koje počinju na  $m$  i završavaju se na  $ov$ .  
(b) Sve reči koje počinju ciframa.

(c) Sve reči koje počinju slovom i završavaju se jednom cifrom.

7. Izvršiti denotaciju sledećih regularnih izraza definisanih nad azbukom

$$\mathbf{V} = \{0, 1\} :$$

(a)  $(0 + 1)(0)^*(1)^*$

(b)  $0(0 + 1)^*1$

(c)  $0^*10^*$ .

8. Nacrtati grafove automata za regularne izraze iz prethodnog zadatka.





## Glava 4

# Leksički analizator

Leksičkim analizatorom se realizuje faza leksičke analize u procesu prevođenja jezika. U kodu programa napisanog na programskom jeziku sa kojim se vrši prevođenje se identifikuju leksičke celine (lekseme) koje imaju neki sintaksni smisao, transformišu se u simbole (tokene) i prosleđuju se sintaksnom analizatoru.

### 4.1 Namena leksičkog analizatora

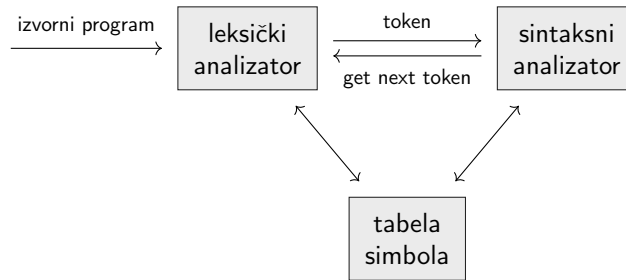
Postoji više razloga zbog kojih se leksički analizator izdvaja od sintaksnog analizatora. Možda najvažniji su:

- Jednostavnija realizacija.
- Tako se dobija mogućnost da se tehnike za sintaksnu analizu razvijaju nezavisno od jezika. Ulaz u sintaksni analizator je niz simbola definisan određenim formalnim jezikom, nije zavisna od programskog jezika koji se prevodi.
- Prenosivost kompilatora. U fazi leksičke analize eliminišu se svi mašinski zavisni elementi i generiše mašinski nezavisna sekvenca simbola koja se prosleđuje sintaksnom analizatoru.
- Povećanje efikasnosti kompilatora. Posebnim tehnikama baferisanja, koje se koriste u realizaciji leksičkog analizatora, doprinosi se njegovoj efikasnosti a time i efikasnosti kompilatora.

Sprega leksičkog i sintaksnog analizatora prikazana je na slici 4.1.

Pored navedene osnovne uloge, leksički analizator obavlja još neke zadatke:

- Izbacuje iz ulaznog koda delove koji nisu značajni za sintaksnu analizu: komentare, praznine (blanko znake), *tab* i *newline* simbole.



SLIKA 4.1: Veza između leksičkog i sintaksnog analizatora

- Usklađuje listing gršaka sa ulaznim kodom. Na primer, vodi računa o broju *newline* simbola, što se koristi kod referenciranja na grške.

Često se leksički analizator realizuje tako da su njegove funkcije razdvojene u dve faze:

- ulazna konverzija – odgovorna za transformaciju ulaznog koda, izbacivanje praznina i sl;
- leksička analiza – odgovorna za generisanje tokena.

U okviru ulazne konverzije vrši se transformisanje i pojednostavljivanje ulaznog koda i njegovo prilagođavanje ulaznom formatu leksičkog analizatora. Danas se neke funkcije ulazne konverzije vrše već u fazi pisanja koda, odnosno prenete su editoru, programu za pripremu koda. Mnogi editori transformišu kod u neki standardni format pogodan za kasnije pregledanje od strane programera, ali ga i prilagođavaju leksičkom analizatoru. Na primer, transformišu identifikatore, dodaju ili brišu neko slovo, mala slova zamenjuju velikim i obrnuto, kako bi identifikatori bili napisani uniformno.

Sam postupak leksičke analize objasnićemo na sledećem primeru. Leksički analizator vidi ulazni kod (napisan na nekom programskom jeziku) kao niz slova. Na primer, ako u programu imamo:

```

if (i == j)
    z = 0;
else
    z = 1;
  
```

leksički analizator će ovo videti kao sledeći niz slova:

```
\tif(i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

gde `\t` označava tabulator a `\n` novu liniju.

Njegov zadatak je da iz ovog niza izdvoji lekseme i predstavi ih simbolima, tokenima. Sintaksnom analizatoru prosleđuje niz tokena. Kao što su u srpskom jeziku leksičke celine *imenica*, *glagol*, *pridev*, u programskim jezicima su to *identifikator*, *ključna reč*, *konstanta*, i sl.

### 4.1.1 Tokeni

Jedan token u suštini predstavlja skup nizova (reči). Na primer:

- identifikator: niz koji počinje slovom a može da sadrži slova i cifre;
- celobrojna konstanta: neprazan niz cifara, sa ili bez znaka;
- ključna reč: **else**, **if**, **begin**, itd;

Tokenima se identifikuju leksičke celine koje su od interesa za sintaksni analizador. Za sintaksnu analizu nisu bitni razmaci, znaci za nove linije, tabulatori i komentari. U nekim jezicima čak i neke reči same naredbe nisu bitne. Na primer, u programskom jeziku COBOL naredba može da se sastoji od reči jezika koje su bitne i reči koje nisu bitne za prepoznavanje naredbe, već se koriste samo da bi zapis naredbe bio jasniji.

Za analizu, takođe, nije bitno da li se na određenom mestu nalazi identifikator koji se zove ALFA ili BETA, već je jedino to da je na određenom mestu identifikator, zbog toga se sintaksnom analizatoru preko tokena prenosi samo ta informacija.

Izbor tokena koji će se koristiti zavisi od programskog jezika na koji se odnosi kompilator, kao i od toga kako je projektovan sintaksni analizador. Na primer, u prethodno datom nizu

```
\tif (i == j)\n\t\tz = 0;\n\t\telse\n\t\t\tz = 1;
```

biće izdvojeni sledeći tokeni: *ključna reč if, otvorena zagrada, identifikator, relacija, identifikator, zatvorena zagrada, identifikator, dodela, celobrojna konstanta, graničnik, ključna reč else, identifikator, dodela, celobrojna konstanta, graničnik.*

Definisanje tokena može bitno da utiče na kvalitet samog prevođenja a time i na kvalitet jezika. To se može ilustrovati nekim karakterističnim primerima iz istorije programskih jezika.

Na primer, prilikom realizacije kompilatora za programski jezik FORTRAN uzimano je da se blanko znaci ignorišu. To znači da se nizovi **V AR1** i **VA R1** tretiraju kao isti identifikator **VAR1**. Međutim, ovo pravilo je dovelo do čuvene greške koja se potkrala u programu za upravljanje letilicom iz programa Gemini, gde je umesto naredbe

```
DO 10 I = 1, 100
```

bilo napisano

```
DO 10 I = 1. 100,
```

što je od strane leksičkog analizatora prepoznato kao da promenljiva **DO10I** dobija vrednost **1.100** umesto kao zaglavlje **DO** petlje. Ova greška je bila razlog pada letilice.

Leksički analizator se obično realizuje tako da se uvek gleda jedan ili više simbola unapred da bi se pravilno identifikovala leksema.

Primeri jezika u kojima su neke stvari bile loše postavljene pa su zbog toga otežavale analizu ima mnogo. Pomenućemo još neke:

- U programskom jeziku PL-I ključne reči nisu bile zaštićene pa su mogle da se koriste i kao identifikatori. Nije teško uočiti kakvu zabunu leksičkom analizatoru unosi sledeći deo koda:

```
IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN
```

Takođe, u primeru

```
DECLARE (ARG1, ..., ARGN)
```

nije lako sve do zatvorene zagrade zaključiti da li je `DECLARE` referenca na višedimenzionalno polje ili naredba za opis. Ovo je zahtevalo da se prilikom realizacije leksičkog analizatora koristi višestruki pogled unapred.

- Nažalost, sličnih primera ima i u savremenim jezicima. Na primer, u C++ se za definisanje templejta koristi sintaksa

```
Foo<Bar>,
```

dok je strim sintaksa:

```
cinn >{}> var.
```

Međutim, javlja se konflikt sa ugnježdenim templejtima:

```
Foo<Bar<Bazz>{}>.
```

### 4.1.2 Šabloni

Potreban nam je način da jednoznačno definišemo koji nizovi odgovaraju kom tokenu. U te svrhe se obično koriste regularni izrazi. Međutim, prilikom definisanja ovih regularnih izraza koristi se nešto izmenjena notacija u odnosu na već korišćenu, što ilustruje sledeći primer.

$$\begin{aligned} \text{cifra} &= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \quad \text{ili} \quad \text{cifra} = [0 - 9], \\ \text{int} &= \text{cifra} \text{ cifra} \quad \text{ili} \quad \text{int} = \text{cifra}^+, \\ \text{slovo} &= A \mid B \mid C \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \quad \text{ili} \quad \text{slovo} = [A - Z a - z], \\ \text{id} &= \text{slovo}(\text{slovo} \mid \text{cifra})^*, \\ \text{float} &= (+ \mid -)? \text{cifra} + (. \text{cifra})? (E(+ \mid -)? \text{cifra})? \end{aligned}$$

U ovoj notaciji važi da se simbol  $\mid$  koristi kao operator *ili*, simbol  $-$  kao oznaka za interval dok je

$$(\mathbf{r})? = \mathbf{r} \mid \varepsilon \quad \text{i} \quad \mathbf{r}^+ = \mathbf{r}^* \setminus \varepsilon.$$

Regularni izrazi su šabloni za prepoznavanje nizova i njihovo preslikavanje u tokene. To je samo jedan od načina kako se mogu definisati šabloni i u praksi se često koristi. U principu, šablon tokena može da bude definisan i na bilo koji drugi način. Mi smo na početku ove lekcije koristili tekstualni opis, pa smo npr. identifikator definisali kao niz koji počinje slovom, a može da sadrži slova i cifre.

Da zaključimo:

- **Leksema** – ulazni niz koji se prepoznaje na osnovu formalnog opisa pomoću šablona i za koji se generiše određeni token.
- **Token** – izlazni simbol koji se generiše kada je prepoznat određeni ulazni niz znakova.
- **Šablon** (*pattern*) – regularni izraz, formalni opis ulaznih nizova za koje se generiše određeni token.

#### Primer 4.1 Tokeni, leksema i šabloni

U sledećoj tabeli predstavljeni su primeri nekih tokena, leksema za koje se generiše i šablona kojima se definisani i prema kojima se generiše.

LEKSEMA	ŠABLON	TOKEN
<b>const</b>	Const	<b>const</b>
<b>if</b>	If	<b>if</b>
<, ≤, =, ≠, >, ≥	< ≤ = ≠ > ≥	<b>rel</b>
pi, C, P2	<i>slovo</i> ( <i>slovo</i>   <i>cifra</i> )*	<b>id</b>
3.1416, 0, 6.02E23	(+   -)? <i>cifra</i> + (. <i>cifra</i> +)? (E(+   -)? <i>cifra</i> +)?	<b>num</b>
“Ovo je podatak”	“(slovo   blanko   <i>cifra</i> )*”	<b>literal</b>

### 4.1.3 Tabela simbola

Pored toga što sintaksnom analizatoru predaje niz tokena leksički analizator treba da sačuva neke attribute tih tokena da bi oni bili kasnije iskorišćeni u toku generisanja koda. Na primer, u fazi sintaksne analize važna je samo informacija da je na određenom mestu u nizu identifikator, a nije važno koji je to identifikator. Međutim, kasnije u fazi semantičke analize i u fazi generisanja koda, važno je koji je to identifikator, šta on imenuje, ako imenuje promenljivu kakvog je tipa, da li je u pitanju skalar ili vektor i sl. Takođe, kada se generiše listing grešaka, bitno je koji je identifikator i u kojoj liniji se on nalazi.

Kako leksički analizator prvi dolazi u kontakt sa ulaznim kodom i kasnije ga transformiše on mora da sačuva te informacije za kasniju upotrebu. U te svrhe

leksički analizator generiše tabelu simbola u kojoj se pamte svi atributi relevantni za određeno simboličko ime. Da bi se kasnije moglo pristupiti odgovarajućem slogu u tabeli simbola, uz token ID kojim se zamenjuju simbolička imena, kao dodatni atribut se prenosi i pointer na slog u tabeli simbola koji odgovara tom simboličkom imenu. Nekada se neki atributi prenose uz sam token, kao na primer vrednosti konstanti i sl, kako je to pokazano u sledećem primeru.

#### Primer 4.2 Generisani tokeni

Uzmimo kao primer sledeću naredbu napisanu u programskom jeziku C:

$$E = M * C ** 2$$

Tokeni sa pridruženim atributima:

*$\langle id, pokazivač na tablicu simbola za E \rangle$*

*$\langle op\_dod \rangle$*

*$\langle id, pokazivač na TS za M \rangle$*

*$\langle op\_mul \rangle$*

*$\langle id, pokazivač na TS za C \rangle$*

*$\langle op\_exp \rangle$*

*$\langle num, type\ integer, val\ 2 \rangle$*

Detaljnije o organizaciji i realizaciji tabela simbola biće reči u poglavlju 9.

## 4.2 Realizacija leksičkog analizatora

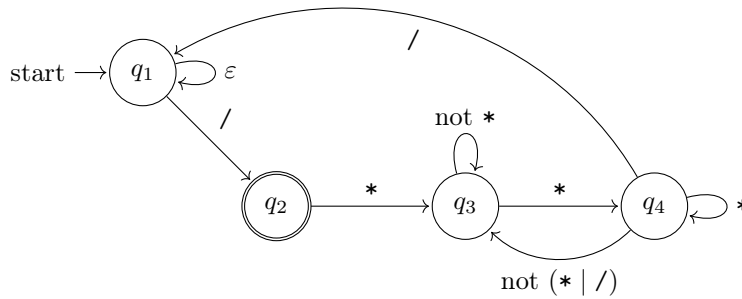
Kako je osnovni zadatak leksičkog analizatora da prepozna nizove slova koji su definisani preko regularnih izraza, za njegovu realizaciju se obično koriste konačni automati.

Na primeru jednog jednostavnog programskog jezika pokazaćemo kako može da bude generisan graf automata koji prepoznaje taj jezik. Pretpostavimo da naš jezik sadrži:

- komentare oblika: */\*Ovo je komentar \*/*
- relacione operatore: *<, <=, <>, =, >, >=*,
- aritmetičke operatore: *+, -, /, \**,
- operator dodeljivanja *:=*,
- separatore *;, (, )*,
- ključne reči (u tabeli ključnih reči),

- identifikatore (simbolička imena),
- literale (nizove oblika: 'Ovo je podatak'),
- celobrojne konstante bez znaka,
- beline (prazna slova).

Leksički analizator treba da prepozna sve navedene lekseme, da ignoriše komentare i beline i da generiše tokene za sve preostale slučajeve. Biće realizovan kao konačni automat koji polazi iz početnog stanja uvek kada započinje prepoznavanje nove lekseme i prelazi u završno stanje kada je prepoznao leksemu. U završnim stanjima automata generišu se odgovarajući tokeni.



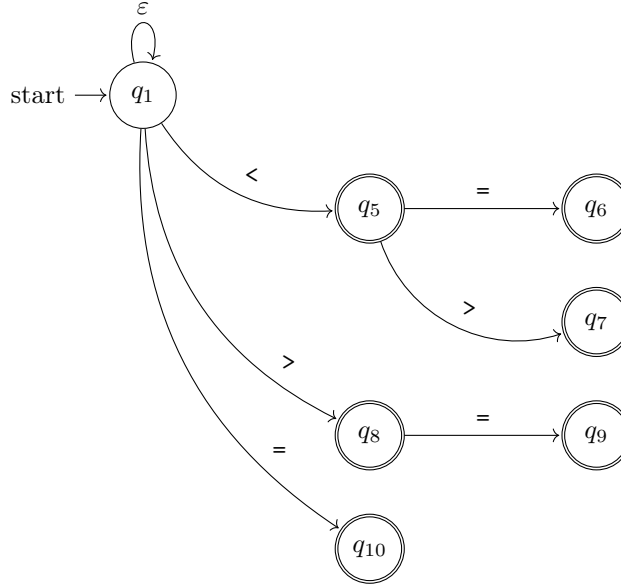
SLIKA 4.2: Deo automata kojim se izbacuju komentari

Deo ovog automata kojim se izbacuju komentari prikazan je na slici 4.2. Na ovom grafu stanje  $q_2$  automata je završno stanje i u njemu se prepoznaje token  $/$  kao operator deljenja. Kada dođe u stanje  $q_2$ , analizator će na osnovu toga koji je sledeći znak po redu odlučiti da li je to završno stanje ili će preći u novo stanje. Ako je iza kose crte znak  $*$ , analizator prelazi u sledeće stanje. U svim ostalim slučajevima stanje  $q_2$  je završno i prepoznaje se token  $/$ .

Deo automata kojim se prepoznaju relacioni operatori prikazan je na slici 4.3. Stanje  $q_5$  je završno za operator  $<$  i u njemu se generiše token **lt**, u stanju  $q_6$  se prepoznaje  $<=$  i generiše se token **le**, dok se u stanju  $q_7$  prepoznaje  $<>$  i generiše se token **ne**. Stanje  $q_8$  prepoznaje  $>$  i daje token **gt**, a stanje  $q_9$  operator  $>=$  i token **ge**, dok je stanje  $q_{10}$  završno stanje za operator  $=$  i generiše token **eq**.

Ovaj primer pokazuje kako se tretiraju dvoznaci. U ovom slučaju za niz od dva znaka koja čine jedan operator generiše se jedan token.

Na slici 4.4 predstavljen je deo automata kojim se prepoznaju aritmetički operatori i operator dodele. Svakom od aritmetičkih operatora odgovara po jedno završno stanje. Obično su oni sami za sebe tokeni. U ovom delu treba obratiti pažnju na to kako se prepoznaje operator dodele. Znak dvotačke  $(:)$  prevodi automat u stanje  $q_{16}$  ali to stanje nije završno zato što u ovom jeziku dvotačka



SLIKA 4.3: Prepoznavanje relacionih operatora

nema leksički smisao: tek sa pojavom znaka = iza dvotačke prelazi se u stanje  $q_{17}$  koje je završno i u kome se generiše odgovarajući token.

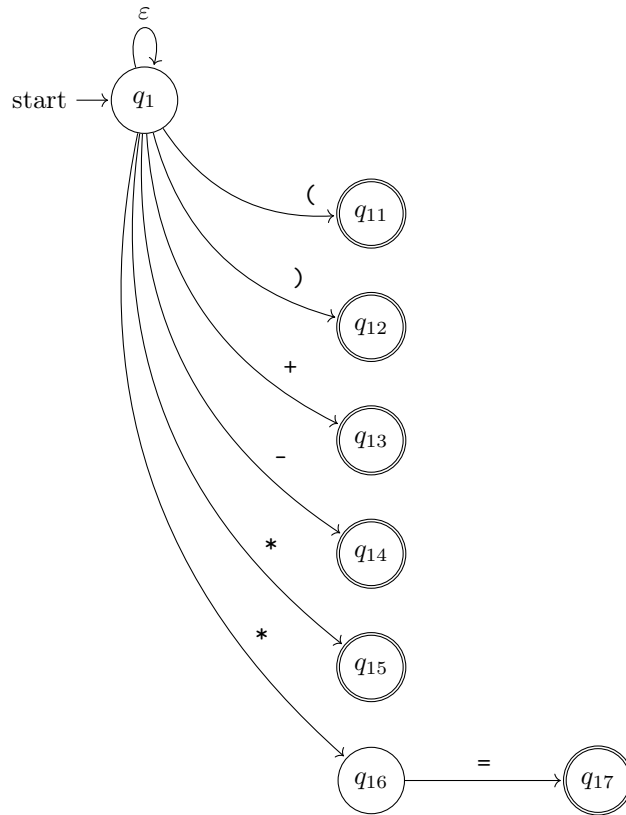
Deo automata kojim se prepoznaju identifikatori, literali i graničnik ; prikazan je na slici 4.5. Svako slovo prevodi automat u stanje  $q_{19}$ . Svako sledeće slovo ili cifra zadržavaju ga u tom stanju sve dok se ne pojavi neki drugi znak kada to stanje postaje završno. Nakon toga najpre se proverava da li je prepoznata reč ključna reč jezika. U te svrhe pretražuje se tablica ključnih reči jezika. Ukoliko se u ovoj tablici ne pronađe ta reč, generiše se token **id**, što znači da je prepoznat identifikator.

Interesantan je i deo kojim se prepoznaju literali (znakovne konstante). U deo automata koji prepoznaje ove lekseme ulazi se ako se pojavi znak apostrof (') koji prevodi automat u stanje 20. Svaki znak koji nije apostrof zadržava automat u tom stanju. Svaki sledeći apostrof prevodi automat u stanje  $q_{21}$ . Ovo stanje je završno stanje ako se string završava jednim ili neparnim brojem apostrofa. Paran broj apostrofa vraća automat u stanje  $q_{20}$  koje nije završno. U stanju  $q_{21}$  generiše se token **str** i kao njegov atribut pamti prepoznati literal.

Na slici 4.6 prikazan je deo automata kojim se prepoznaju celi brojevi bez znaka. Pojava cifre prevodi automat u stanje  $q_{22}$ . Svaka sledeća cifra zadržava ga u tom stanju. Nailaskom znaka koji se razlikuje od cifre stanje  $q_{22}$  postaje završno i generiše se token **int**.

U programskim jezicima se obično definiše više tipova konstanti i one su znatno složenije od celobrojnih konstanti bez znaka koje smo imali u našem primeru. Na





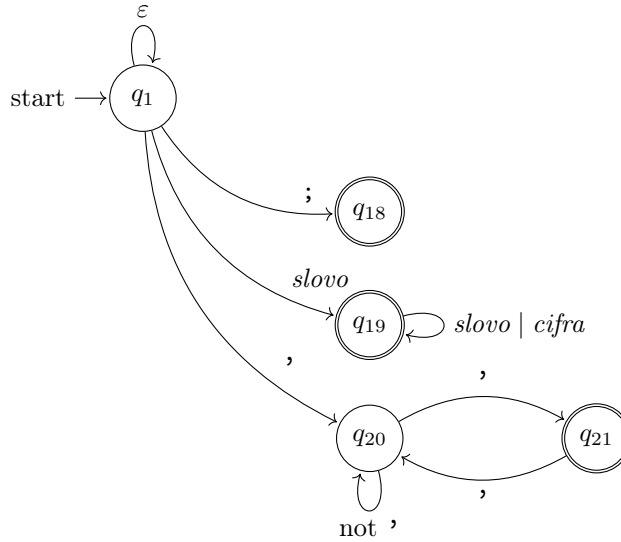
SLIKA 4.4: Prepoznavanje aritmetičkih operatora

slici 4.7 prikazan je graf automata koji prepoznaje celobrojne i realne konstante zapisane u normalnom i eksponencijalnom zapisu, prema njihovoj definiciji u programskom jeziku Pascal. Ove konstante definisane su regularnim izrazom

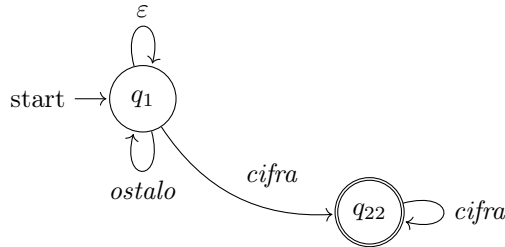
$$num = (+|-)? \text{cifra}^* (. \text{cifra}^+)? (E(+|-)? \text{cifra}^+)?.$$

### 4.3 Programska realizacija leksičkog analizatora

Programska realizacija leksičkog analizatora se svodi na pisanje programa koji realizuje automat koji je definisan. Tu mogu da se primene različite tehnike. Jedan pristup je da se napiše potprogram za svako od stanja automata pa da se pozivi potprograma povežu u skladu sa grafom automata. Drugi pristup je da se napiše program koji realizuje automat u celosti.



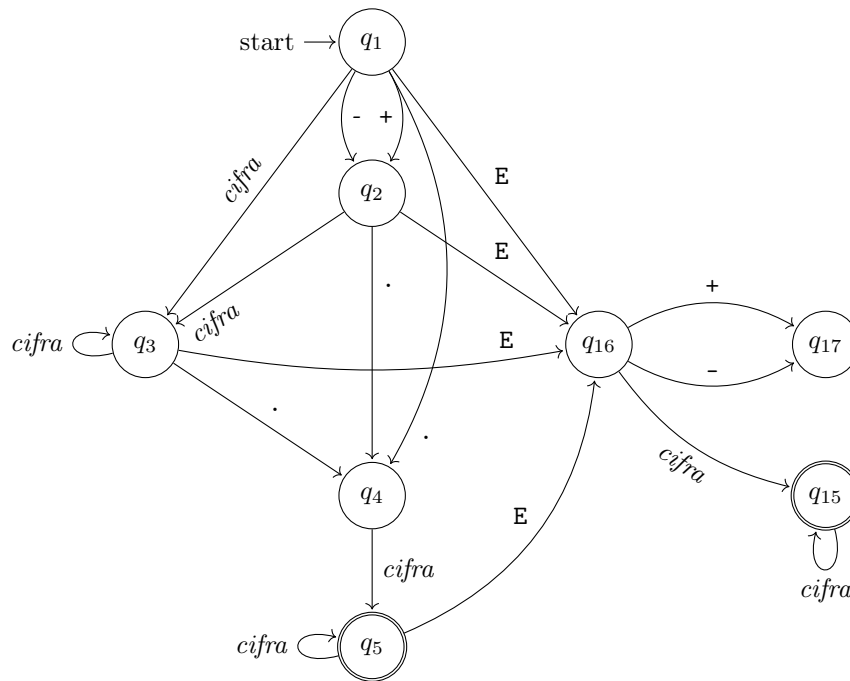
SLIKA 4.5: Prepoznavanje identifikatora i literala



SLIKA 4.6: Prepoznavanje celih brojeva bez znaka

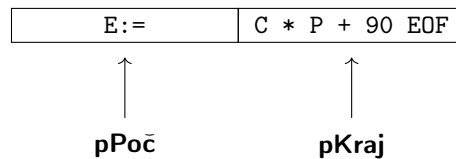
### 4.3.1 Tehnika dvostrukog bafera

Prilikom programske realizacije leksičkog nalizatora često se koriste tehnika kod koje se kao pomoćna struktura koristi dvostruki bafer (obično dužine 1024 bajtova); slika 4.8. Najpre se puni leva polovina bafera. Oba pokazivača (**pPoč** – pokazivač početka lekseme i **pKraj** – pokazivač kraja lekseme) se postavljaju na početak. Čita se znak po znak niza i **pKraj**-pokazivač kraja lekseme se pomera udesno sve dok se ne prepozna kraj lekseme. Kada se prepozna jedna leksema **pPoč** se pomera na poziciju **pKraj** i kreće se sa prepoznavanjem nove lekseme. Kada **pKraj** dođe do kraja prvog bafera puni se drugi bafer i nastavlja se sa traženjem kraja lekseme u drugom baferu. Kada se dođe do kraja drugog bafera puni se prvi bafer i nastavlja se sa traženjem lekseme od njegovog početka. Na ovaj način dobija se jedan beskonačni bafer. Jasno je da zbirna dužina oba bafera treba da bude veća od najduže lekseme koja se može pojaviti u programu.



SLIKA 4.7: Automat koji prepoznaje brojeve prema definiciji u jeziku Pascal

Zbog toga je bitno kako su definisani leksički elementi jezika. Već smo pomenuli problem sa dužinom naredbe **DECLARE** u progrskom jeziku PL-1 koji se javljao zbog toga što u ovom jeziku ključne reči nisu bile zaštićene, već su mogle da se koriste i kao identifikatori promenljivih.



SLIKA 4.8: Dvostruki bafer

Pseudokod algoritma koji se koristi pri analizi tehnikom dvostrukog bafera dat je u nastavku.

```

pKraj := pKraj + 1;
if (pKraj <> eof) then begin
  if pKraj na kraju prve polovine
then
  reload druge polovine;
  pKraj := pKraj + 1;
end

```

```

else if pKraj na kraju druge polovine then begin
    reload prve polovine;
    pomeriti pKraj na početak prve polovine
end
else Kraj lekseme
pPoč = pKraj
End

```

## 4.4 Pitanja

1. Objasniti namenu leksičkog analizatora.
2. Objasniti zašto je bolje da se leksička analiza izdvoji kao posebna faza u procesu prevođenja jezika.
3. Šta su to tokeni?
4. Objasniti vezu između tokena, leksema i šablona.
5. Objasniti vezu između šablona i regularnih izraza.
6. Šta je to tablica simbola?
7. Koje greške može da otkriva leksički analizator?
8. Objasniti tehniku dvostrukog bafera.

## 4.5 Zadaci

1. Identifikovati lekseme u sledećem delu koda napisanog na programskom jeziku C.

```

float triangle(float width, float height) {
    float area;
    area = width * height / 2.0;
    return (area);
}

```

2. Celobrojne konstante u programskom jeziku C su definisane sledećim skupom pravila.

$$\begin{aligned}
 \text{decimal-constant} &\rightarrow \text{nonzero-digit} \mid \text{decimal-constant digit} \\
 \text{octal-constant} &\rightarrow \text{octal-digit} \mid \text{octal-constant octal-digit} \\
 \text{hexadecimal-constant} &\rightarrow 0x \text{ hexadecimal-digit} \\
 &\quad \mid 0X \text{ hexadecimal-digit} \\
 &\quad \mid \text{hexadecimal-constant hexadecimal-digit}
 \end{aligned}$$

$nonzero-digit \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$   
 $octal-digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7$   
 $hexadecimal-digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$   
 $\mid a \mid b \mid c \mid d \mid e \mid f$   
 $\mid A \mid B \mid C \mid D \mid E \mid F.$

Nacrtati graf automata koji prepoznaje ovako definisane celobrojne konstante.

3. Nacrtati graf automata koji prepoznaje sve operatore dodeljivanja i relacione operatore u C++. Uključiti operatore dodeljivanja

$=, \quad *, \quad /, \quad \%, \quad +, \quad -, \quad <=, \quad >=, \quad \&, \quad ^, \quad |,$

kao i relacione operatore

$==, \quad !=, \quad <, \quad >, \quad <=, \quad >=.$



## Glava 5

# Sintaksna analiza

Sintaksni analizator prima niz simbola (tokena) od leksičkog analizatora i proverava da li taj niz pripada jeziku koji je opisan zadatom gramatikom. Cilj sintaksnog analizatora je da generiše sintaksno stablo za ulazni niz simbola.

### 5.1 Redosled primene pravila

Da bi se generisalo sintaksno stablo potrebno je odrediti koja se pravila i u kom redosledu primenjuju prilikom preslikavanja startnog simbola u analizirani niz. Generalno gledano, postoje dva osnovna principa po kojima se određuje redosled pravila. Kako se za opis jezika obično koriste beskonteksne gramatike, u svakom koraku izvođenja po jedan neterminalni simbol se preslikava u neku reč, pa redosled primene pravila može da bude:

1. sleva nadesno (levo izvođenje) – kada se zamenjuje prvi neterminalni simbol sa leve strane i
2. zdesna nalevo (desno izvođenje) – kada se zamenjuje prvi neterminalni simbol sa desne strane.

Sledeći primer ilustruje ova dva principa izvođenja.

#### **Primer 5.1** Levo i desno izvođenje

Neka je data gramatika

$$\mathbf{G}(\{\langle id \rangle, \langle slovo \rangle, \langle cifra \rangle\}, \{a, b, c, \dots, z, 0, 1, \dots, 9\}, id, \mathbf{P}),$$

gde je  $\mathbf{P}$  sledeći skup pravila:

$$\begin{aligned}\langle id \rangle &\rightarrow \langle slovo \rangle \mid \langle id \rangle \langle slovo \rangle \mid \langle id \rangle \langle cifra \rangle, \\ \langle slovo \rangle &\rightarrow a \mid b \mid c \mid \dots \mid z, \\ \langle cifra \rangle &\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9.\end{aligned}$$

Razmotrićemo generisanje reči  $a111$ .

Levo izvođenje reči daje:

$$\begin{aligned}\langle id \rangle &\rightarrow \langle id \rangle \langle cifra \rangle \\ &\rightarrow \langle id \rangle \langle cifra \rangle \langle cifra \rangle \\ &\rightarrow \langle id \rangle \langle cifra \rangle \langle cifra \rangle \langle cifra \rangle \\ &\rightarrow \langle slovo \rangle \langle cifra \rangle \langle cifra \rangle \langle cifra \rangle \\ &\rightarrow a \langle cifra \rangle \langle cifra \rangle \langle cifra \rangle \\ &\rightarrow a1 \langle cifra \rangle \langle cifra \rangle \\ &\rightarrow a11 \langle cifra \rangle \\ &\rightarrow a111.\end{aligned}$$

Desnim izvođenjem dobija se:

$$\begin{aligned}\langle id \rangle &\rightarrow \langle id \rangle \langle cifra \rangle \\ &\rightarrow \langle id \rangle 1 \\ &\rightarrow \langle id \rangle \langle cifra \rangle 1 \\ &\rightarrow \langle id \rangle 11 \\ &\rightarrow \langle id \rangle \langle cifra \rangle 11 \\ &\rightarrow \langle id \rangle 111 \\ &\rightarrow \langle slovo \rangle 111 \\ &\rightarrow a111.\end{aligned}$$

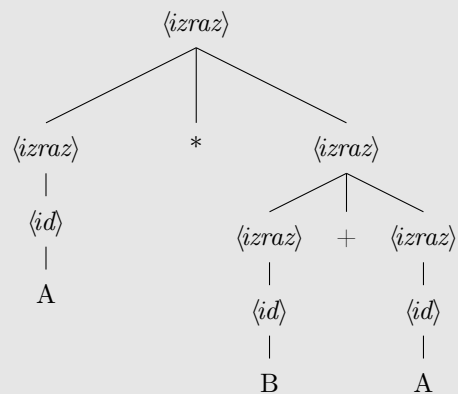
Na sledećoj slici predstavljeno je sintaksno stablo koje odgovara jednom i drugom izvođenju.





$$\begin{aligned} &\rightarrow A * B + \langle \text{izraz} \rangle \\ &\rightarrow A * B + \langle id \rangle \\ &\rightarrow A * B + A. \end{aligned}$$

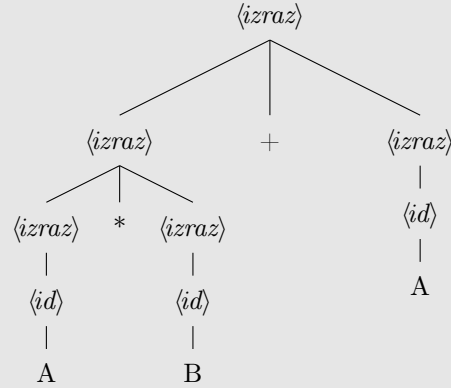
Sintaksno stablo koje odgovara ovom izvođenju dato je na sledećoj slici. U ovom slučaju bi najpre bila generisana naredba sabiranja a zatim množenja. (Pri generisanju naredbi se prolazi kroz sintaksno stablo odozdo naviše.)



Desnim izvođenjem za posmatrani niz dobija se:

$$\begin{aligned} \langle \text{izraz} \rangle &\rightarrow \langle \text{izraz} \rangle + \langle \text{izraz} \rangle \\ &\rightarrow \langle \text{izraz} \rangle + \langle id \rangle \\ &\rightarrow \langle \text{izraz} \rangle + A \\ &\rightarrow \langle \text{izraz} \rangle * \langle \text{izraz} \rangle + A \\ &\rightarrow \langle \text{izraz} \rangle * \langle id \rangle + A \\ &\rightarrow \langle \text{izraz} \rangle * B + A \\ &\rightarrow \langle id \rangle * B + A \\ &\rightarrow A * B + A. \end{aligned}$$

Ovom izvođenju odgovara sintaksno stablo prikazano na sledećoj slici. U ovom slučaju prvo bi bila generisana naredba množenja, a zatim naredba sabiranja.



Očigledno je da ova gramatika, za posmatranu reč, primenom levog i desnog izvođenja daje različita sintaksna stabla. To znači da rezultat analize nije jednoznačno definisan i bio bi generisan različit skup naredbi zavisno od toga koje se izvođenje primenjuje.

Kako bi se izbegla ova višeznačnost potrebno je bolje definisati gramatiku. Na primer, sledeća gramatika je ekvivalentna navedenoj, a pri tome je i jednoznačna.

$$\mathbf{G} = (\{ \langle \text{izraz} \rangle, \langle \text{sabirak} \rangle, \langle \text{faktor} \rangle, \langle \text{id} \rangle \}, \{ +, *, A, B \}, \langle \text{izraz} \rangle, \mathbf{P})$$

Skp smena  $\mathbf{P}$  je:

$$\begin{aligned} \langle \text{izraz} \rangle &\rightarrow \langle \text{sabirak} \rangle \mid \langle \text{izraz} \rangle + \langle \text{sabirak} \rangle, \\ \langle \text{sabirak} \rangle &\rightarrow \langle \text{faktor} \rangle \mid \langle \text{sabirak} \rangle * \langle \text{faktor} \rangle, \\ \langle \text{faktor} \rangle &\rightarrow \langle \text{id} \rangle. \end{aligned}$$

Za posmatranu reč  $A * B + A$  dobija se sledeće levo izvođenje:

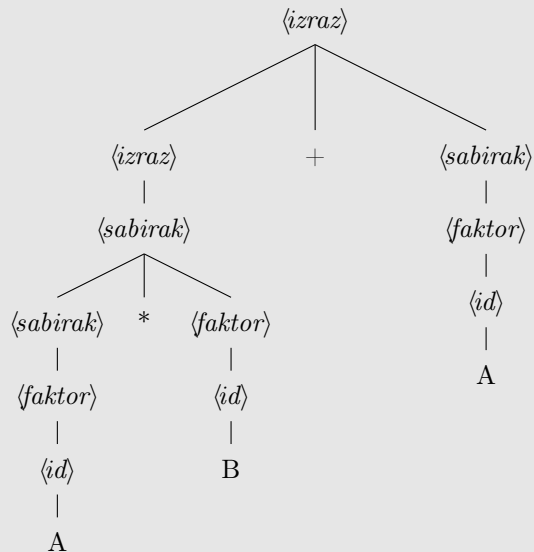
$$\begin{aligned} \langle \text{izraz} \rangle &\rightarrow \langle \text{izraz} \rangle + \langle \text{sabirak} \rangle \\ &\rightarrow \langle \text{sabirak} \rangle + \langle \text{sabirak} \rangle \\ &\rightarrow \langle \text{sabirak} \rangle * \langle \text{faktor} \rangle + \langle \text{sabirak} \rangle \\ &\rightarrow \langle \text{faktor} \rangle * \langle \text{faktor} \rangle + \langle \text{sabirak} \rangle \\ &\rightarrow \langle \text{id} \rangle * \langle \text{faktor} \rangle + \langle \text{sabirak} \rangle \\ &\rightarrow A * \langle \text{faktor} \rangle + \langle \text{sabirak} \rangle \\ &\rightarrow A * \langle \text{id} \rangle + \langle \text{sabirak} \rangle \\ &\rightarrow A * B + \langle \text{sabirak} \rangle \\ &\rightarrow A * B + \langle \text{faktor} \rangle \\ &\rightarrow A * B + \langle \text{id} \rangle \end{aligned}$$

$\rightarrow A * B + A.$

Desno izvođenje za istu reč bi bilo:

$$\begin{aligned}
 \langle \text{izraz} \rangle &\rightarrow \langle \text{izraz} \rangle + \langle \text{sabirak} \rangle \\
 &\rightarrow \langle \text{izraz} \rangle + \langle \text{faktor} \rangle \\
 &\rightarrow \langle \text{izraz} \rangle + \langle \text{id} \rangle \\
 &\rightarrow \langle \text{izraz} \rangle + A \\
 &\rightarrow \langle \text{sabirak} \rangle + A \\
 &\rightarrow \langle \text{sabirak} \rangle * \langle \text{faktor} \rangle + A \\
 &\rightarrow \langle \text{sabirak} \rangle * \langle \text{id} \rangle + A \\
 &\rightarrow \langle \text{sabirak} \rangle * B + A \\
 &\rightarrow \langle \text{faktor} \rangle * B + A \\
 &\rightarrow \langle \text{id} \rangle * B + A \\
 &\rightarrow A * B + A.
 \end{aligned}$$

Oba ova izvođenja generišu isto sintaksno stablo prikazano u nastavku.



U odnosu na to kako obavljaju sintaksnu analizu postoje dva tipa sintakasnih analizatora:

- *top-down* analizatori koji vrše analizu odozgo naniže i
- *bottom-up* analizatori koji vrše analizu odozdo naviše.

U slučaju *top-down* analize polazi se od startnog simbola i nastoji se da se odrede pravila koja treba primeniti da bi se generisala reč čija se analiza vrši. To znači da se pravila otkrivaju u redosledu u kom se i primenjuju prilikom generisanja reči, odnosno odozgo naniže ako se to posmatra na sintaksnom stablu.

U slučaju *bottom-up* analize primenjuje se postupak redukcije. Kreće se od reči čija se analiza vrši i nastoji se da se ta reč redukuje na startni simbol. Pravila se određuju u redosledu koji je suprotan redosledu njihove primene kod generisanja reči i sintaksnog stabla, odnosno odozdo naviše ako se to gleda na sintaksnom stablu.

U principu *bottom-up* analizatori su efikasniji i na njima se uglavnom zasnivaju komercijalna rešenja kompilatora.

Postoje veoma efikasni algoritmi za sintakсну analizu i u jednoj i u drugoj klasi i ovde će biti predstavljeni neki od njih. Kako bi se bolje sagledala razlika u pristupu analizi najpre će biti objašnjeni osnovni algoritmi za analizu iz obe klase analizatora.

## 5.2 Osnovni algoritam za *top-down* sintakсну analizu

Kreće se od startnog simbola i bira se prvo pravilo kojim se startni simbol preslikava u neku frazu. Posle toga u svakom koraku se nastoji da se prvi neterminalni simbol sa leve strane zameni desnom stranom prvog raspoloživog pravila. Praktično, nastoji se da se generiše analizirani niz levim izvođenjem. Ukoliko na nekom koraku ne postoji odgovarajuća smena, vraćamo se natrag do nivoa na kome je moguće primeniti alternativno pravilo. Analiza je uspešna ako se kao rezultat ovog postupka dobije niz koji se analizira. Ako se vraćanjem unazad dođe ponovo do startnog simbola gramatike, i nema novih alternativa za zamenu tog simbola, postupak analize je neuspešan.

Prilikom realizacije ovog algoritma obično se koriste dva pokazivača, **pUlaz** koji pokazuje na znak u ulaznom nizu koje se u određenom trenutku prepoznaje i **pRadni** koji pokazuje na tekući znak u nizu koji se izvodi.

Prilikom analize obično se koristi neki specijalni znak kao završni, granični znak niza. U našem slučaju će to biti simbol #.

Algoritam se formalno može formulisati na sledeći način:

1. U izvedenu sekvencu upisati niz  $S\#$ , gde je  $S$  startni simbol, a  $\#$  granični. Postaviti pokazivač **pRadni** na prvi znak u izvedenom nizu (na startni simbol) a pokazivač **pUlaz** na prvi simbol ulaznog niza.
2. Ukoliko je simbol na koji pokazuje **pRadni** u izvedenoj sekvenci neterminalni simbol, zameniti ga desnom stranom prve smene na čijoj je levoj

strani taj neterminalni simbol i postaviti **pRadni** na prvi simbol unete reči.

3. Ukoliko je simbol na koji pokazuje **pRadni** u izvedenoj sekvenci terminalni simbol i pri tome jednak ulaznom simbolu na koji pokazuje **pUlaz**, to znači da je taj simbol prepoznat, oba pokazivača se pomeraju za jedno mesto udesno i postupak se nastavlja (prelazi se na analizu sledećeg simbola).
4. Ukoliko je simbol na koji pokazuje **pRadni** u izvedenoj sekvenci terminalni simbol i različit od ulaznog simbola na koji pokazuje **pUlaz**, poništiti dejstvo poslednje primenjene smene, vratiti se na stanje pre primene te smene i pokušati na tom nivou sa primenom nove smene. Ukoliko na tom nivou nema alternativa vratiti se još jedan korak nazad poništiti poslednju primenjenu smenu i tržiti novu alternativu na tom nivou.
5. Ukoliko se vraćanjem dođe do startnog simbola i pri tome nema novih alternativa, ulazni niz nije moguće prepoznati, postupak prepoznavanja niza je neuspešan, kôd sadrži sintaksnu grešku.
6. Postupak analize je uspešan kada se oba pokazivača (**pUlaz** i **pRadni**) nađu na graničnom simbolu.

### Primer 5.3 Osnovni algoritam za *top-down* analizu

Neka je data gramatika

$$\mathbf{G} = (\{S, A, B\}, \{a, b, c, d\}, S, \mathbf{P})$$

sa skupom pravila **P**:

1.  $S \rightarrow aAd$ ,
2.  $S \rightarrow aB$ ,
3.  $A \rightarrow b$ ,
4.  $A \rightarrow c$ ,
5.  $B \rightarrow ccd$ ,
6.  $B \rightarrow ddc$ .

Primenu algoritma pokazaćemo na primeru reči  $a\ c\ c\ d\ \#$ .

	ULAZNI NIZ	IZVEDENI NIZ	AKCIJA
1	<span style="border: 1px solid black;">a</span> ccd#	<span style="border: 1px solid black;">S</span> #	primenjuje se (1) ×
2	<span style="border: 1px solid black;">a</span> ccd#	<span style="border: 1px solid black;">a</span> Ad#	prepoznaje se <i>a</i>
3	a <span style="border: 1px solid black;">c</span> cd#	a <span style="border: 1px solid black;">A</span> d#	primenjuje se (3) ×
4	a <span style="border: 1px solid black;">c</span> cd#	a <span style="border: 1px solid black;">b</span> d#	greška, vraćanje unatrag
5	a <span style="border: 1px solid black;">c</span> cd#	a <span style="border: 1px solid black;">A</span> d#	primenjuje se (4) ×
6	a <span style="border: 1px solid black;">c</span> cd#	a <span style="border: 1px solid black;">c</span> d#	prepoznaje se <i>c</i>
7	ac <span style="border: 1px solid black;">c</span> d#	ac <span style="border: 1px solid black;">d</span> #	greška, vraćanje unatrag
8	a <span style="border: 1px solid black;">c</span> cd#	a <span style="border: 1px solid black;">A</span> d#	nema novih pravila, vraćanje
9	<span style="border: 1px solid black;">a</span> ccd#	<span style="border: 1px solid black;">S</span> #	primenjuje se (2)
10	<span style="border: 1px solid black;">a</span> ccd#	<span style="border: 1px solid black;">a</span> B#	prepoznaje se <i>a</i>
11	a <span style="border: 1px solid black;">c</span> cd#	a <span style="border: 1px solid black;">B</span> #	primenjuje se (5)
12	a <span style="border: 1px solid black;">c</span> cd#	a <span style="border: 1px solid black;">c</span> cd#	prepoznaje se <i>c</i>
13	ac <span style="border: 1px solid black;">c</span> d#	ac <span style="border: 1px solid black;">c</span> d#	prepoznaje se <i>c</i>
14	acc <span style="border: 1px solid black;">d</span> #	acc <span style="border: 1px solid black;">d</span> #	prepoznaje se <i>d</i>
15	accd <span style="border: 1px solid black;">#</span>	accd <span style="border: 1px solid black;">#</span>	uspešan kraj analize

**Napomena:** Posebno su označena slova na koja pokazuju ulazni i radni pokazivač, koja se posmatraju u određenom koraku. Uz pravila koja su poništena stoji oznaka ×.

Za generisanje reči *accd#*, potrebno je primentiti smene (2) i (5). Na sledećoj slici prikazano je sintakšno stablo za analiziranu reč.

```

graph TD
    S --> a
    S --> B
    B --> c1[c]
    B --> c2[c]
    B --> d

```

### 5.2.1 Problem leve rekurzije

*Top-down* sintaksnom analizom se generiše skup smena čiji redosled primene odgovara levom izvođenju. Zbog toga ovaj postupak analize ne može da se primeni kod tzv. levo rekurzivnih gramatika. Levo rekurzivne gramatike su

gramatike kod kojih postoje levo rekurzivna pravila, odnosno pravila oblika:

$$A \rightarrow Ar, \quad \text{gde je } A \in \mathbf{V}_n, r \in \mathbf{V}^*.$$

Kod ovakvih pravila preslikavanjem neterminala  $A$  dobija se reč koja opet počinje istim neterminalom, i to se iz koraka u korak ponavlja, odnosno ulazi se u jednu beskonačnu petlju, tako da analzu nije moguće završiti.

**Primer 5.4** Levo rekurzivna gramatika

Neka je data gramatika

$$\mathbf{G} = (\{S, A\}, \{a, c\}, S, \mathbf{P})$$

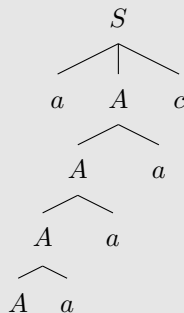
gde je  $\mathbf{P}$  sledeći skup pravila:

$$S \rightarrow aAc,$$

$$S \rightarrow Aa,$$

$$S \rightarrow \varepsilon.$$

Ova gramatika će u postupku sintaksne analize uvek ući u petlju prikazanu na sledećoj slici.



Pored ove direktne rekurzije gramatika može da ima i indirektna rekurzivna pravila koje je teško otkriti.

**Primer 5.5** Levo rekurzivna gramatika sa indirektnim rekurzivnim pravilima

Sledeći skup pravila sadrži rekurziju po dubini:

$$S \rightarrow Aa \mid b,$$

$$A \rightarrow Sd \mid c \mid \varepsilon.$$



Neterminal  $S$  generiše neterminal  $A$ , a ovaj opet neterminal  $S$ .

### 5.2.2 Eliminisanje leve rekurzije

Svaku levu rekurzivnu gramatiku moguće je transformisati u ekvivalentnu nerekurzivnu gramatiku. Navešćemo dve mogućnosti za to.

Ako gramatika sadrži skup pravila za isti neterminalni simbol, među kojima su neka levo rekurzivna, a neka ne:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m,$$

ovaj skup pravila moguće je zameniti sledećim skupom nerekurzivnih pravila:

$$\begin{aligned} A &\rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m \mid \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_m A', \\ A' &\rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n \mid \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A'. \end{aligned}$$

Moguća je i sledeća transformacija:

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_m A', \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A'. \end{aligned}$$

#### Primer 5.6 Eliminisanje leve rekurzije

Neka je data levo rekurzivna gramatika:

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

Ekvivalentna nerekurzivna gramatika bila bi:

$$\begin{aligned} E &\rightarrow T \mid TE', \\ E' &\rightarrow +T \mid +TE', \\ T &\rightarrow F \mid FT', \\ T' &\rightarrow *F \mid *FT', \\ F &\rightarrow (E) \mid a. \end{aligned}$$

Eliminisanje indirektno rekurzije moguće je sledećom transformacijom:

Sve skupove pravila oblika

$$X \rightarrow Y\alpha,$$

$$Y \rightarrow X\beta_1 \mid X\beta_2 \mid \cdots \mid X\beta_n$$

treba zameniti pravilima

$$x \rightarrow X\beta_1\alpha \mid X\beta_2\alpha \mid \cdots \mid X\beta_n\alpha.$$

Posle ove transformacije treba se osloboditi svih direktnih levo-rekurzivnih smena, ukoliko postoje.

### Primer 5.7 Eliminisanje leve rekurzije

Gramatika iz primera 5.5 može da se transformiše u nerekurzivnu gramatiku na sledeći način:

Najpre ćemo razdvojiti smene da bismo bolje videli indirektnu rekurziju:

1.  $S \rightarrow Aa$
2.  $S \rightarrow b$
3.  $A \rightarrow Sd$
4.  $A \rightarrow c$
5.  $A \rightarrow \varepsilon$

Prvo eliminišemo indirektna levo rekurzivna pravila. U datoj gramatici pravila (1) i (3) su indirektno rekurzivna. Transformacija se vrši tako što se u smeni (1) neterminal A zameni desnim stranama svih smena za taj neterminal. Kako se posle te zamene neterminal A ne pojavljuje u desnoj strani ni jedne smene, neterminal A (kao i smene za preslikavanje tog neterminalnog simbola) se može eliminisati iz gramatike. Nakon ove transformacije, gramatika će sadržati sledeći skup smena:

$$\begin{array}{ll} S \rightarrow Sda, & S \rightarrow ca, \\ S \rightarrow a, & S \rightarrow b. \end{array}$$

U sledećem koraku se eliminišu direktno levo rekurzivno pravila (1), tako da je gramatika konačno definisana skupom smena:

$$\begin{array}{ll} S \rightarrow ca, & S \rightarrow a, \\ S \rightarrow b, & S \rightarrow caS', \\ S \rightarrow aS', & S \rightarrow bS'. \end{array}$$

### Primer 5.8 Top-down analiza

Blok naredbi u jeziku  $\mu$ Pascal definisan je sledećom gramatikom:

$$\mathbf{G} = (\{Blok, NizNar, Naredba, Dodela, Izraz\}, \\ \{\mathbf{begin}, \mathbf{end}, \mathbf{ID}, \mathbf{CONST}, \mathbf{;}, \mathbf{+}, \mathbf{:=}\}, \\ Blok, \mathbf{P}),$$

gde je  $\mathbf{P}$  sledeći skup pravila:

$$\begin{aligned} Blok &\rightarrow \mathbf{begin} \ NizNar \ \mathbf{end} \\ NizNar &\rightarrow NizNar \ \mathbf{;} \ Naredba \mid Naredba \\ Naredba &\rightarrow Dodela \mid Blok \\ Dodela &\rightarrow \mathbf{ID} \ \mathbf{:=} \ Izraz \\ Izraz &\rightarrow Izraz \ \mathbf{+} \ \mathbf{CONST} \mid \mathbf{CONST} \end{aligned}$$

Proveriti da li je blok

$$\mathbf{begin} \ \mathbf{ID} \ \mathbf{:=} \ \mathbf{CONST} \ \mathbf{end}$$

sintaksno korektno napisan.

**Rešenje:** Najpre eliminišemo levo rekurzivna pravila: Najpre skup smena

$$NizNar \rightarrow NizNar \ \mathbf{;} \ Naredba \mid Naredba$$

zamenjujemo smenama:

$$\begin{aligned} NizNar &\rightarrow Naredba \ NizNar' \\ NizNar' &\rightarrow \mathbf{;} \ Naredba \ NizNar' \mid \varepsilon \end{aligned}$$

Zatim skup smena

$$Izraz \rightarrow Izraz \ \mathbf{+} \ \mathbf{CONST} \mid \mathbf{CONST}$$

zamenjujemo smenama:

$$\begin{aligned} Izraz &\rightarrow \mathbf{CONST} \ Izraz' \\ Izraz' &\rightarrow \mathbf{+} \ \mathbf{CONST} \ Izraz' \mid \varepsilon \end{aligned}$$

Transformisana gramatika sadrži sledeći skup pravila:

$$Blok \rightarrow \mathbf{begin} \ NizNar \ \mathbf{end} \quad (1)$$

$$NizNar \rightarrow Naredba \ NizNar' \quad (2)$$

$$NizNar' \rightarrow \mathbf{;} \ Naredba \ NizNar' \quad (3)$$

$$NizNar' \rightarrow \varepsilon \quad (4)$$

$$Naredba \rightarrow Dodela \quad (5)$$

$$Naredba \rightarrow Blok \quad (6)$$

$$Dodela \rightarrow ID := Izraz \quad (7)$$

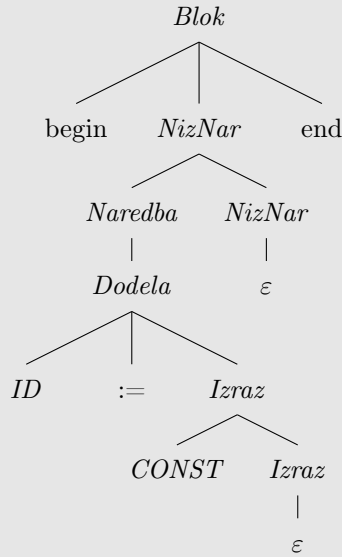
$$Izraz \rightarrow CONST Izraz' \quad (8)$$

$$Izraz' \rightarrow + CONST Izraz' \quad (9)$$

$$Izraz' \rightarrow \varepsilon \quad (10)$$

Postupak sintaksne analize zadate reči je dat u tabeli 5.1.

Za generisanje niza `begin ID := CONST end #` koristi se skup smena  $\{(1), (2), (5), (7), (8), (10), (4)\}$ . Na sledećoj slici je prikazano odgovarajuće sintaksno stablo.



### 5.3 Osnovni algoritam za *bottom-up* analizu

U slučaju *bottom-up* analize primenjuje se postupak redukcije reči koja se analizira. Osnovni algoritam za analizu se sastoji u tome što se ulazni niz analizira slovo po slovo od početka prema kraju i nastoji se da se neki njegov podniz prepozna kao desna strana nekog od raspoloživih pravila gramatike. Kada se takav podniz pronađe, vrši se njegova redukcija na simbol koji je na levoj strani primenjene smene i postupak nastavlja sve dok se niz ne redukuje na startni simbol. Kao i kod osnovnog algoritma za *top-down* analizu i ovde ima vraćanja unatrag i poništavanja primenjenih smena u slučaju kada dođe do greške i ne može da se ide dalje sa analizom.

U LAZNI NIZ	GENERISANI NIZ	AKCIJA
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{Blok} \boxed{\#}$	primenjuje se (1)
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{NizNar} \boxed{\text{end}} \boxed{\#}$	prepoznaje se begin
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{NizNar} \boxed{\text{end}} \boxed{\#}$	primenjuje se (2)
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{Naredba} \boxed{NizNar'} \boxed{\text{end}} \boxed{\#}$	(5)
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{Dodela} \boxed{NizNar'} \boxed{\text{end}} \boxed{\#}$	(7)
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{ID} := \boxed{Izraz} \boxed{NizNar'} \boxed{\text{end}} \boxed{\#}$	prepoznaje se ID
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{ID} := \boxed{Izraz} \boxed{NizNar'} \boxed{\text{end}} \boxed{\#}$	prepoznaje se :=
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{ID} := \boxed{Izraz} \boxed{NizNar'} \boxed{\text{end}} \boxed{\#}$	(8)
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{Izraz'NizNar'} \boxed{\text{end}} \boxed{\#}$	prepoznaje se CONST
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{Izraz'} \boxed{NizNar'} \boxed{\text{end}} \boxed{\#}$	(9) $\times$
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{+} \boxed{CONST} \boxed{Izraz'NizNar'} \boxed{\text{end}} \boxed{\#}$	greška
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{Izraz'} \boxed{NizNar'} \boxed{\text{end}} \boxed{\#}$	(10)
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{NizNar'} \boxed{\text{end}} \boxed{\#}$	(3) $\times$
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{;} \boxed{Naredba} \boxed{NizNar'} \boxed{\text{end}} \boxed{\#}$	greška
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{NizNar'} \boxed{\text{end}} \boxed{\#}$	(4)
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	prepoznaje se end
$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	$\boxed{\text{begin}} \boxed{ID} := \boxed{CONST} \boxed{\text{end}} \boxed{\#}$	uspešan kraj

TABELA 5.1: Postupak sintaksne analize iz primera 5.7

Pri realizaciji ovog algoritma koristi se magacin u koji se ubacuje slovo po slovo iz ulaznog niza. Pri tome se u svakom koraku ispituje da li je moguće izvršiti redukciju niza koji je u vrhu magacina. Ako redukcija nije moguća, u magacin se ubacuje novo slovo, a ako je moguća, prepoznati niz sa vrha magacina se redukuje na neterminal koji je na levoj strani primenjene smene, odnosno neterminal zamenjuje redukovani niz u magacinu. Analiza je uspešno izvršena kada se dođe do graničnog simbola, a u magacinu ostane samo startni simbol.

Ilustrovaćemo ovaj postupak na primeru reči i gramatike iz primera 5.3.

**Primer 5.9** Osnovni algoritam za *bottom-up* analizu

Neka je data gramatika

$$\mathbf{G} = (\{S, A, B\}, \{a, b, c, d\}, S, \mathbf{P})$$

sa skupom smena  $\mathbf{P}$ :

$$S \rightarrow aAd \quad (1)$$

$$S \rightarrow aB \quad (2)$$

$$A \rightarrow b \quad (3)$$

$$A \rightarrow c \quad (4)$$

$$B \rightarrow ccd. \quad (5)$$

*Bottom-up* sintaksna analiza reči  $a c c d \#$  data je u nastavku.

	ULAZNI NIZ	NIZ U MAGACINU	AKCIJA
1	$acc d \#$	$\boxed{\varepsilon}$	
2	$cc d \#$	$\boxed{a}$	Nema smene, ubacuje se novo slovo
3	$cd \#$	$a\boxed{c}$	Primenjuje se smena (4)
4	$cd \#$	$a\boxed{A}$	Nema smene, gleda se dublje
5	$cd \#$	$a\boxed{A}$	Nema smene, ubacuje se novo slovo
6	$d \#$	$aA\boxed{c}$	Primenjuje se smena (4)
7	$d \#$	$aA\boxed{A}$	Nema smene, gleda se dublje
8	$d \#$	$a\boxed{AA}$	Nema smene, gleda se dublje
9	$d \#$	$a\boxed{AA}$	Nema smene, ubacuje se novo slovo

10	#	$aAA\boxed{d}$	Nema smene, gleda se dublje
11	#	$aA\boxed{Ad}$	Nema smene, gleda se dublje
12	#	$a\boxed{AA}d$	Nema smene, gleda se dublje
13	#	$\boxed{aAA}d$	Nema smene, nema slova za ubacivanje; poništava se sve do koraka 6 i pokušava sa da se nađe nova alternativa na tom nivou
14	d#	$aA\boxed{c}$	Nema smene, gleda se dublje
15	d#	$a\boxed{Ac}$	Nema smene, gleda se dublje
16	d#	$\boxed{aAc}$	Nema smene, ubacuje se novo slovo
17	#	$aAc\boxed{d}$	Nema smene, gleda se dublje
18	#	$aA\boxed{cd}$	Nema smene, gleda se dublje
19	#	$a\boxed{Acd}$	Nema smene, gleda se dublje
20	#	$\boxed{aAcd}$	Nema smene, nema slova za ubacivanje; poništava se sve do poslednje primenjene smene (korak 3) i traži se nova alternativa na tom nivou
21	cd#	$a\boxed{c}$	Nema smene, gleda se dublje
22	cd#	$\boxed{ac}$	Nema smene, ubacuje se slovo
23	d#	$ac\boxed{c}$	Primenjuje se smena (4)
24	d#	$ac\boxed{A}$	Nema smene, gleda se dublje
25	d#	$a\boxed{cA}$	Nema smene, gleda se dublje
26	d#	$\boxed{acA}$	Nema smene, ubacuje se novo slovo
27	#	$acA\boxed{d}$	Nema smene, gleda se dublje
28	#	$ac\boxed{Ad}$	Nema smene, gleda se dublje
29	#	$a\boxed{cAd}$	Nema smene, gleda se dublje
30	#	$\boxed{acAd}\#$	Nema smene, nema slova za ubacivanje; poništava se sve do 23 i pokušava sa da se nađe nova alternativa na tom nivou
31	d#	$ac\boxed{c}$	Nema smene, gleda se dublje

32	$d\#$	$a\boxed{cc}$	Nema smene, gleda se dublje
33	$d\#$	$\boxed{acc}$	Nema smene, ubacuje se novo slovo
34	$\#$	$ac\boxed{d}$	Nema smene, gleda se dublje
35	$\#$	$a\boxed{cd}$	Nema smene, gleda se dublje
36	$\#$	$a\boxed{ccd}$	Primenjuje se smena (5)
37	$\#$	$a\boxed{B}$	Nema smene, gleda se dublje
38	$\#$	$\boxed{aB}$	Primenjuje se smena (2)
39	$\#$	$\boxed{S}$	Uspešan kraj analize

Korišćena je notacija u kojoj je uokviren niz koji se nalazi u vrhu magacina i trenutno traži kao desna strana smene.

I u ovom postupku dobija se da se analizirani niz generiše primenom smena (5) i (2) samo što su one otkrivene u redosledu koji je suprotan od njihove primene, što je i suština samog postupka. Sintaksno stablo koje odgovara reči je isto kao i u primeru 5.3.

## 5.4 Pitanja

1. Objasniti šta je zadatak sintaksnog analizatora.
2. Objasniti pojam levog i desnog izvođenja.
3. Objasniti koji se problem javlja kod nejednoznačno definisanih gramatika.
4. Koja su to dva osnovna načina sintaksne analize?
5. Opisati osnovni algoritam za *top-down* analizu.
6. Šta je to problem leve rekurzije i kada se javlja?
7. Dati transformacije kojima se eliminiše leva rekurzija.
8. Šta je to indirektna rekurzija i kako se eliminiše?
9. Objasniti osnovni algoritam za *bottom-up* analizu.



## 5.5 Zadaci

1. Primenom osnovnog algoritma za *top-down* sintaksnu analizu proveriti da li niz  $\# \mathbf{READ}(\mathbf{ID}, \mathbf{ID}, ) \#$  pripada jeziku gramatike  $G$  zadate sledećim skupom pravila:

$$\begin{aligned} \text{Ulaz} &\rightarrow \mathbf{READ}(\text{NizPromenljivih}) \\ \text{NizPromenljivih} &\rightarrow \text{NizPromenljivih}, \mathbf{ID} \mid \mathbf{ID} \end{aligned}$$

2. Primenom osnovnog algoritma za *top-down* sintaksnu analizu proveriti da li reč  $\#a + a * (a + a)\#$  pripada jeziku koji je definisan gramatikom

$$\mathbf{G} = (\{S, I, F\}, \{a, +, *\}, S, \mathbf{P}),$$

gde je  $\mathbf{P}$  sledeći skup pravila:

$$\begin{aligned} S &\rightarrow S + I \mid I & I &\rightarrow I * F \mid F \\ F &\rightarrow a(S) \mid a. \end{aligned}$$

3. Neka je gramatika  $\mathbf{G}$  zadata skupom pravila

$$S \rightarrow S + I \mid I, \quad I \rightarrow (S) \mid f(S) \mid id.$$

Primenom osnovnog algoritma za *top-down* sintaksnu analizu proveriti da li niz

$$\#id + f(id + *f(id))\#$$

pripada jeziku ove gramatike.

4. Primenom osnovnog algoritma za *bottom-up* analizu proveriti da li reč  $\#a + a * a\#$  pripada jeziku definisanom gramatikom iz zadatka 2.
5. Traženu proveru iz zadatka 3 izvršiti primenom osnovnog algoritma za *bottom-up* analizu.



## Glava 6

# Prediktivna analiza – LL-1 analizatori

Osnovni problem kod polaznih algoritama za sintaksnu analizu predstavljenih u prethodnom poglavlju je veliki broj povratnih koraka tako da je sam postupak analize dosta dug i neizvestan. U ovom poglavlju biće razmotreni mnogo efikasniji analizatori za *top-down* analizu koji u svakom koraku vrše neku vrstu predikcije na osnovu koje odlučuju koja će se pravilo primeniti. Ako među raspoloživim pravilima ne postoji odgovarajuće pravilo postupak analize se prekida kao neuspešan. Smene se otkrivaju direktno bez povratnih petlji.

Za razliku od osnovnog algoritma za *top-down* analizu kod kojeg se pravila biraju na osnovu prvog neterminalnog simbola sa leve strane u izvedenom nizu, kod ovih analizatora odluka o pravilu koje će da se primeni se donosi i na osnovu slova u ulaznom nizu koje treba da se prepozna, na kome je trenutno ulazni pokazivač. Primenjuje se pravilo koje će sigurno da generiše bar to slovo. Ovakvi analizatori su poznati kao LL-1 analizatori (*look-left 1*), gde cifra 1 ukazuje na to da se predikcija vrši na osnovu jednog slova u ulaznom nizu. Generalno gledano, mogu da se definišu i LL- $k$  analizatori kod kojih se predikcija vrši na osnovu reči dužine  $k$ , ali su LL-1 mnogo upotrebljiviji.

Da bi mogla da se vrši predikcija, gramatika kojom je opisan jezik mora da bude tako definisana da za jedan ulazni simbol i jedan neterminal postoji najviše jedna raspoloživa smena, nema više alternativa. Krenućemo od definicije proste LL-1 gramatike.

## 6.1 Proste LL-1 gramatike

Proste LL-1 gramatike su beskonteksne gramatike kod kojih sve smene za isti neterminalni simbol počinju različitim terminalnim simbolima:

$$A \rightarrow \alpha_1\alpha_1 \mid \alpha_2\alpha_2 \mid \cdots \mid \alpha_n\alpha_n, \quad \text{gde je } a_i \in \mathbf{V}_t, \alpha_i \in \mathbf{V}^*, a_i \neq a_j \text{ za } i \neq j.$$

### Primer 6.1 Prosta LL-1 gramatika

Gramatika  $\mathbf{G} = (\{S, A\}, \{a, b, c, d\}, S, \mathbf{P})$ , gde je  $\mathbf{P}$  skup pravila

$$\begin{array}{ll} S \rightarrow aS & S \rightarrow bA \\ A \rightarrow d & A \rightarrow ccA, \end{array}$$

je LL-1 gramatika. Za neterminal  $S$  postoje dva pravila kojima se  $S$  preslikava u reči koje počinju različitim početnim slovima ( $a$  i  $b$ ). Za neterminal  $A$  postoje dva pravila koja takođe imaju različita početna slova reči u koje se preslikava taj neterminal ( $d$ ,  $c$ ).

Reč  $aabccd\#$  pripada jeziku definisanom ovom gramatikom i izvodi se na sledeći način.

$$S' \rightarrow S\# \rightarrow aS\# \rightarrow aaS\# \rightarrow aabA\# \rightarrow aabccA\# \rightarrow aabccd\#$$

**Napomena:** Skup pravila je dopunjen pravilom:  $S' \rightarrow S\#$ , kako bi se uveo i granični simbol  $\#$ .

Za sintaksnu analizu jezika definisanih LL-1 gramatikama koristi se magacinski automat koji kao pomoćnu strukturu koristi sintaksnu tabelu. Ova sintaksna tabela se sastoji od onoliko kolona koliko ima terminalnih simbola i onoliko vrsta koliko ima neterminalnih i terminalnih simbola zajedno. Polja sintaksne tabele se popunjavaju tako da se u polju koje odgovara neterminalu  $A$  i terminalu  $a$ , upisuje smena koji na levoj strani ima neterminal  $A$ , a na desnoj strani reč koja počinje terminalom  $a$ , ako takva smena postoji. U preseku vrste koja je označena nekim terminalom i kolone koja je označena istim tim terminalom upisuje se vrednost  $pop$ , dok se u preseku vrste i kolone koje su označene graničnim simbolom  $\#$  upisuje znak za kraj analize ( $acc$ ). Sva ostala polja su polja greške ( $err$ ). Sintaksna tabela  $Ts(A, a)$ , gde je  $A$  oznaka vrste, a  $a$  oznaka kolone se formalno može definisati na sledeći način:

$$Ts(A, a) = \begin{cases} pop, & \text{ako je } A = a, a \in \mathbf{V}_t, \\ acc, & \text{ako je } A = \# \wedge a \in \#, \\ (a\alpha, i), & \text{ako je } A \rightarrow a\alpha \text{ } i\text{-to pravilo,} \\ err, & \text{u svim ostalim slučajevima.} \end{cases}$$

**Primer 6.2** LL-1 sintaksna tabela

Sintaksna tabela koja odgovara gramatici iz primera 6.1 data je u nastavku.

VRH MAGACINA	ULAZNI SIMBOL				
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	#
<i>S</i>	( <i>aS</i> , 1)	( <i>bA</i> , 2)			
<i>A</i>			( <i>ccA</i> , 4)	( <i>d</i> , 3)	
<i>a</i>	<i>pop</i>				
<i>b</i>		<i>pop</i>			
<i>c</i>			<i>pop</i>		
<i>d</i>				<i>pop</i>	
#					<i>acc</i>

Opisani magacinski automat prepoznaje reč koju analizira tako što se u magacin ubacuje startni niz  $S\#$  i u svakom koraku vrši preslikavanje određeno sintaksnom tabelom, odnosno:

$$M : \{V \cup \{\#\}\} \times \{V_t \cup \{\#\}\} \mapsto \{(\beta, r), pop, acc, err\}.$$

Označimo konfiguraciju automata u određenom trenutku prepoznavanja sa  $(az, A\alpha, \mathbf{p})$ , gde je:

- $az$ , deo ulaznog niza koji tek treba da bude prepoznat, pri čemu je  $a$  prvi simbol u tom nizu, tj.  $a \in \mathbf{V}_t$  i  $az \in \mathbf{V}_t^*$ ,
- $A\alpha$  generisani niz u magacinu, pri čemu je  $A$  simbol u vrhu magacina,  $A \in \mathbf{V}$  i  $\alpha \in \mathbf{V}^*$ ,
- $\mathbf{p}$  je niz smena koje su do tog trenutka primenjene za generisanje niza.

Preslikavanje koje u toku prepoznavanja vrši automat možemo da definišemo na sledeći način:

$$(az, A\alpha, \mathbf{p}) = \begin{cases} (\mathbf{z}, \alpha, \mathbf{p}), & \text{ako je } Ts(A, a) = pop, \\ kraj, & \text{ako je } Rs(A, a) = acc, \\ (az, \beta\alpha, \mathbf{p}i), & \text{ako je } Ts(A, a) = (\beta, i) - i\text{-to pravilo,} \\ greška & \text{ako je } Ts(A, a) = err. \end{cases}$$

Početna konfiguracija magacina je određena sa:  $(\mathbf{w}, S\#, \varepsilon)$ , gde je  $\mathbf{w}$  reč čije prepoznavanje treba izvršiti,  $S$  startni simbol, a  $\varepsilon$  označava prazan niz primenjenih smena. Niz je uspešno prepoznat ako se u toku analize dođe do graničnog

simbola  $i$  u ulaznom i u generisanom nizu, odnosno iz tablice sintaksne analize pročita vrednost za kraj analize ( $acc$ ).

Napomenimo da iz same definicije LL-1 gramatike proizilazi da tabela sintaksne analize mora da bude definisana jednoznačno. U svakom koraku prepoznavanja reči jednoznačno je definisano preslikavanje koje treba izvršiti: ili je moguće primeniti neku smenu i napredovati u postupku prepoznavanja, ili je došlo do greške i prepoznavanje treba prekinuti. U ovom postupku analize nema povratnih koraka, što je posledica same definicije LL-1 gramatika. Postupak analize pokazaćemo na primeru reči  $aabccd\#$  i gramatike iz primera 6. 1.

### Primer 6.3 Postupak LL-1 sintaksne analize

Ako se posmatra gramatika iz primera 6.1, u toku prepoznavanja reči  $aabccd\#$ , u automatu se događaju sledeća preslikavanja:

$$\begin{aligned}
 (aabccd\#, S\#, \varepsilon) &\rightarrow (aabccd\#, aS\#, 1) \\
 &\rightarrow (abccd\#, S\#, 1) \\
 &\rightarrow (abccd\#, aS\#, 11) \\
 &\rightarrow (bccd\#, S\#, 11) \\
 &\rightarrow (bccd\#, ba\#, 112) \\
 &\rightarrow (ccd\#, A\#, 112) \\
 &\rightarrow (ccd\#, ccA\#, 1124) \\
 &\rightarrow (cd\#, cA\#, 1124) \\
 &\rightarrow (d\#, A\#, 1124) \\
 &\rightarrow (d\#, d\#, 11243) \\
 &\rightarrow (\#, \#, 11243).
 \end{aligned}$$

Kako se posle primene nekog pravila sigurno prepoznaje jedan terminalni simbol iz ulaznog niza, gore dati postupak prepoznavanja se može napisati i skraćeno na sledeći način:

$$\begin{aligned}
 (aabccd\#, S\#, \varepsilon) &\rightarrow (abccd\#, S\#, 1) \\
 &\rightarrow (bccd\#, S\#, 11) \\
 &\rightarrow (ccd\#, A\#, 112) \\
 &\rightarrow (cd\#, cA\#, 1124) \\
 &\rightarrow (d\#, A\#, 1124) \\
 &\rightarrow (\#, \#, 11243).
 \end{aligned}$$

### 6.1.1 Leva faktORIZACIJA

Pokazali smo da se u slučaju LL-1 gramatika može vršiti prediktivna sintaksna analiza koja je mnogo efikasnija od osnovnog algoritma za sintaksnu analizu. Veoma značajno u svemu tome je da se gramatike koje ne zadovoljavaju uslov LL-1 gramatika mogu jednostavno transformisati u ovaj tip gramatika. Naime, ukoliko u gramatici postoji veći broj pravila kojima se isti neterminalni simbol preslikava u reči sa istim početnim slovom, odnosno smene oblika

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma, \quad \text{gde je } \alpha, \beta_1, \dots, \beta_n, \gamma \in \mathbf{V}^* \wedge A \in \mathbf{V}_n,$$

takva gramatika sigurno nije LL-1 gramatika, ali se lako može transformisati u LL-1 gramatiku primenom sledećih smena:

$$\begin{aligned} A &\rightarrow aA' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

Ova transformacija gramatika poznata je pod imenom leva faktORIZACIJA.

#### Primer 6.4 Leva faktORIZACIJA

Neka je gramatika zadata sledećim skupom pravila:

$$\begin{aligned} Nar &\rightarrow \text{if Izraz then } Nar \text{ else } Nar \mid \text{if Izraz then } Nar \\ Izraz &\rightarrow b \end{aligned}$$

Kako u datoj gramatici za neterminal  $Nar$  postoje dva pravila koja na desnoj imaju reči sa istim prefiksom: **if Izraz then**  $Nar$ , ova gramatika nije LL-1. Njoj ekvivalentna LL-1 gramatika je:

$$\begin{aligned} Nar &\rightarrow \text{if Izraz then } Nar \text{ else } Nar Nar' \\ Nar' &\rightarrow \text{else } Nar \mid \varepsilon \\ Izraz &\rightarrow b \end{aligned}$$

## 6.2 Pomoćne funkcije za definisanje opštih LL-1 gramatika

Datom definicijom LL-1 gramatika obuhvaćen je samo najjednostavniji slučaj ovih gramatika kada reč na desnoj strani smene počinje terminalnim simbolom. Međutim, taj uslov može i da se proširi da obuhvati i složenije slučajeve LL-1

gramatika. Na primer, razmotrimo sledeću gramatiku u kojoj imamo i pravila koja neterminale preslikavaju u reči koje započinju neterminalima:

$$S' \rightarrow S\# \quad (1)$$

$$A \rightarrow c \quad (2)$$

$$S \rightarrow Abe \quad (3)$$

$$B \rightarrow AS \quad (4)$$

$$A \rightarrow dB \quad (5)$$

$$B \rightarrow b \quad (6)$$

$$A \rightarrow aS \quad (7)$$

Očigledno je da se prilikom odlučivanja kada se mogu primeniti takva pravila moraju uzeti u obzir svi terminalni simboli koji se nalaze na početku fraza koje se generišu od početnih neterminala. U našem primeru pravilo (4) će moći da se primeni u svim slučajevima kada se u radnom magacinu nalazi neterminal  $B$ , a prepoznaje se neki od terminalnih simbola kojim započinju fraze neterminala  $A$ , koji je početni simbol desne strane pravila (4).

Još komplikovanije je odlučivanje kada koje pravilo treba primeniti ako u gramatici postoje tzv.  $\varepsilon$ -pravila, odnosno kada se neki neterminalni simbol ukida, kao u slučaju gramatike definisane sledećim skupom pravila:

$$1. A' \rightarrow A\#$$

$$2. A \rightarrow iB \leftarrow e$$

$$3. B \rightarrow SB$$

$$4. B \rightarrow \varepsilon$$

Za koji terminalni simbol će biti primenjeno takvo pravilo očigledno zavisi od toga koji se simboli nalaze iza neterminala koji se ukida. U ovom primeru pravilo  $B \rightarrow \varepsilon$  će moći da se primeni za one terminalne simbole koji mogu da se nađu iza neterminala  $B$ .

Kako bi u definiciji LL-1 gramatika obuhvatili sve ove slučajeve definišu se dve funkcije koje nalaze primenu i kod nekih drugih tipova sintaksnih analizatora. To su funkcije First i Follow.

### 6.2.1 Funkcija First

Za svaku smenu oblika  $X \rightarrow a$ , može da se definiše funkcija  $\text{First}(a)$  koja kao rezultat daje sve terminalne simbole koji mogu da se nađu na početku reči izvedenih iz reči  $a$ , tj.

$$\alpha \in \mathbf{V}^+ \Rightarrow \text{First}(\alpha) = \{s \mid \alpha \xrightarrow{*} s\beta, \text{ gde je } s \in \mathbf{V}_t, \beta \in \mathbf{V}^*\}.$$



**Primer 6.5** Funkcija First

Neka je gramatika zadata sledećim skupom pravila.

$$\begin{array}{llll} S' \rightarrow S\# & S \rightarrow Abe & S \rightarrow dB & A \rightarrow dB \\ A \rightarrow aS & A \rightarrow c & B \rightarrow AS & B \rightarrow b \end{array}$$

Tada je:

$$\begin{aligned} \text{First}(A) &= \{s \mid A \xrightarrow{*} s\beta, s \in \mathbf{V}_t, \beta \in \mathbf{V}^*\} = \{a, c, d\}, \\ \text{First}(ABe) &= \{s \mid ABe \xrightarrow{*} s\beta, s \in \mathbf{V}_t, \beta \in \mathbf{V}^*\} = \{a, c, d\}. \end{aligned}$$

**6.2.2 Funkcija Follow**

Funkcija Follow se definiše za neterminalne simbole, kao skup terminalnih simbola koji mogu u toku izvođenja da se nađu iza tog neterminalnog simbola, ili:

$$\text{Follow}(A) = \{s \in \mathbf{V}_t \mid S' \xrightarrow{*} \alpha As\gamma, A, S' \in \mathbf{V}_n, \gamma \in \mathbf{V}^*\},$$

gde je  $S'$  startni simbol gramatike.

Po definiciji, Follow funkcija startnog simbola gramatike sadrži granični simbol  $\#$ .

Za određivanje Follow funkcije neterminalnog simbola  $X$  posmatraju se desne strane pravila u kojima se pojavljuje posmatrani simbol. Pri tome simbol  $X$  na desnoj strani smene može da se nađe u jednom od sledećih konteksta:

1.  $Z \rightarrow \alpha Xx\beta \wedge x \in \mathbf{V}_t \Rightarrow x \in \text{Follow}(X)$ ,
2.  $Z \rightarrow \alpha XY\beta \wedge Y \in \mathbf{V}_n \Rightarrow \text{First}(Y) \subset \text{Follow}(X)$ ,

(**Napomena:** ukoliko postoji izvođenje  $Y \rightarrow \varepsilon$ , simbol  $Y$  se preskače i gleda se nastavak smene.)

3.  $Z \rightarrow \alpha X \Rightarrow \text{Follow}(Z) \subset \text{Follow}(X)$ .

**Primer 6.6** Funkcija Follow

Naka je data gramatika

$$\mathbf{G}(\{A, B, C, D\}, \{i, \leftarrow, e, [, ], *\}, A, \mathbf{P})$$

definisana sledećim skupom pravila.

$$A' \rightarrow A\# \quad A \rightarrow iB \quad B \rightarrow SB \leftarrow e \quad B \rightarrow \varepsilon$$

$S \rightarrow [eC]$	$S \rightarrow *i$	$C \rightarrow eC$	$C \rightarrow \varepsilon$
Važi:			
$ \begin{aligned} &A' \rightarrow A\# \\ &\rightarrow iB \leftarrow e\# \\ &\rightarrow isB \leftarrow e\# \\ &\rightarrow i[eC]B \leftarrow e\# \\ &\rightarrow i[e]B \leftarrow e\# \\ &\rightarrow i[e] \leftarrow e\# \end{aligned} $			
i	$\text{Follow}(C) = \{s \in \mathbf{V}_t \mid A' \xrightarrow{*} \alpha C s \gamma\} = \{\}\}.$		

### 6.3 LL-1 gramatike bez $\varepsilon$ -pravila

Iskoristićemo najpre funkciju First da definišemo opštije LL-1 gramatike ali ćemo pri tome isključiti postojanje  $\varepsilon$ -pravila.

Beskonteksna gramatika bez  $\varepsilon$ -pravila je LL-1 gramatika ako su za sva pravila za isti neterminalni simbol oblika  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$  skupovi

$$\text{First}(\alpha_1), \text{First}(\alpha_2), \dots, \text{First}(\alpha_n)$$

disjunktni po parovima, odnosno

$$\text{First}(\alpha_i) \cap \text{First}(\alpha_j) = \emptyset, \quad i \neq j.$$

#### Primer 6.7 LL-1 gramatika bez $\varepsilon$ -pravila

Za gramatiku datu u primeru 6.4 važi:

$$\begin{aligned}
\text{First}(dB) \cap \text{First}(aS) &= \{d\} \cap \{a\} = \emptyset, \\
\text{First}(dB) \cap \text{First}(c) &= \{d\} \cap \{c\} = \emptyset, \\
\text{First}(aS) \cap \text{First}(c) &= \{a\} \cap \{c\} = \emptyset, \\
\text{First}(AS) \cap \text{First}(b) &= \{a, c, d\} \cap \{b\} = \emptyset.
\end{aligned}$$

Ova gramatika ispunjava potrebne uslove za LL-1 gramatiku.

Sintaksna tabela za ovako definisane LL-1 gramatike definisana je sa:

$$\text{Ts}(A, a) = \begin{cases} \text{pop}, & \text{ako je } A = a, a \in \mathbf{V}_t, \\ \text{acc}, & \text{ako je } A = \# \wedge a \in \#, \\ (\beta, i), & \text{ako je } a \in \text{First}(\beta) \text{ i } A \rightarrow \beta \text{ je } i\text{-to pravilo,} \\ \text{err}, & \text{u svim ostalim slučajevima.} \end{cases}$$

Tabela se popunjava tako što se određena smena upisuje u polje na preseku vrste koja odgovara neterminalu sa leve strane smene i u sve kolone koje odgovaraju terminalnim simbolima koji ulaze u skup određen funkcijom First za reč koja je na desnoj stani smene.

**Primer 6.8** Sintaksna tabela za LL-1 gramatiku bez  $\varepsilon$ -pravila

Za gramatiku datu u primeru 6.6 sintaksna tabela je data u nastavku.

VRH MAGACINA	ULAZNI SIMBOL					
	$a$	$b$	$c$	$d$	$e$	$\#$
$S$	$(ABe, 3)$		$(ABe, 3)$	$(ABe, 3)$		
$A$	$(aS, 7)$		$(c, 2)$	$(dB, 5)$		
$B$	$(AS, 4)$	$(b, 6)$	$(AS, 4)$	$(AS, 4)$		
$a$	$\text{pop}$					
$b$		$\text{pop}$				
$c$			$\text{pop}$			
$d$				$\text{pop}$		
$e$					$\text{pop}$	
$\#$						$\text{acc}$

Uočimo da su smene (3) i (4) upisane u kolonama  $a$ ,  $b$  i  $d$  zato što je

$$\text{First}(ABe) = \text{First}(A) = \{a, c, d\} \quad \text{i} \quad \text{First}(AS) = \text{First}(A) = \{a, c, d\}.$$

Sam postupak sintaksne analize je potpuno isti kao i kod prostih LL-1 gramatika.

**Primer 6.9** Postupak sintaksne analize za LL-1 gramatiku bez  $\varepsilon$ -pravila

Za gramatiku iz primera 6.6, postupak sintaksne analize reči  $dcbebe\#$  je

sledeći.

$$\begin{aligned}
 (dccbebe\#, S\#, \varepsilon) &\rightarrow (dccbebe\#, ABe\#, 3) \\
 &\rightarrow (dccbebe\#, dBB\#, 35) \\
 &\rightarrow (ccbebe\#, BBe\#, 35) \\
 &\rightarrow (ccbebe\#, ASBe\#, 354) \\
 &\rightarrow (ccbebe\#, cSBe\#, 3542) \\
 &\rightarrow (cbebe\#, SBe\#, 3542) \\
 &\rightarrow (cbebe\#, aBeBe\#, 35423) \\
 &\rightarrow (cbebe\#, cBeBe\#, 354232) \\
 &\rightarrow (bebe\#, BeBe\#, 354232) \\
 &\rightarrow (bebe\#, beBe\#, 3542326) \\
 &\rightarrow (ebe\#, eBe\#, 35423226) \\
 &\rightarrow (be\#, Be\#, 35423226) \\
 &\rightarrow (be\#, be\#, 354232266) \\
 &\rightarrow (e\#, e\#, 3542266) \\
 &\rightarrow (\#, \#, 35423266).
 \end{aligned}$$

Sintaksna analiza je uspešno završena i kao rezultat je dobijeno da se zadata reč generiše nizom smena

$$(3), (5), (4), (2), (3), (2), (6) \text{ i } (6)$$

odnosno zadatoj reč odgovara sledeće izvođenje:

$$\begin{aligned}
 S' &\xrightarrow{1} S\# \\
 &\xrightarrow{3} ABe\# \\
 &\xrightarrow{5} dBB\# \\
 &\xrightarrow{4} dASBe\# \\
 &\xrightarrow{2} dcSBe\# \\
 &\xrightarrow{3} dcABeBe\# \\
 &\xrightarrow{2} dccBeBe\# \\
 &\xrightarrow{6} dccbeBe\# \\
 &\xrightarrow{6} dccbebe\#.
 \end{aligned}$$

## 6.4 LL-1 gramatike sa $\varepsilon$ -pravilima

Beskontekсна gramatika koja sadrži i  $\varepsilon$ -pravila je LL-1 gramatika ako i samo ako za sva pravila oblika  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$  i  $A \rightarrow \varepsilon$  važi

$$\text{First}(\alpha_i \circ \text{Follow}(A)) \cap \text{First}(\alpha_j \circ \text{Follow}(A)) = \emptyset, \quad \text{za svako } i \neq j,$$

gde je simbolom  $\circ$  označena operacija nadovezivanja (konkatenacije).

Ovaj uslov može da se iskaže i na sledeći način:

1. Za sva pravila  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$  treba da važi

$$\text{First}(\alpha_i) \cap \text{First}(\alpha_j) = \emptyset, \quad \text{za svako } i \neq j;$$

2. Ako je  $\alpha_i \xrightarrow{*} \varepsilon$ , tada mora da važi i

$$\text{First}(\alpha_j) \cap \text{Follow}(A) = \emptyset, \quad \text{za svako } j \neq i.$$

### Primer 6.10 Primer LL-1 gramatike sa $\varepsilon$ -pravilima

Neka je data gramatika

$$\mathbf{G}(\{A, B, C, D\}, \{i, \leftarrow, e, [, ], *\}, A, \mathbf{P})$$

definisana sledećim skupom smena:

$$\begin{array}{llll} A' \rightarrow A\# & A \rightarrow iB \leftarrow e & B \rightarrow SB & B \rightarrow \varepsilon \\ S \rightarrow [eC] & S \rightarrow *i & C \rightarrow eC & C \rightarrow \varepsilon. \end{array}$$

Ispitaćemo da li ova gramatika ispunjava potrebne uslove za LL-1 gramatiku. Dati uslovi treba da budu ispitani za svaki par pravila za isti neterminalni simbol.

Za pravila kojima se definiše preslikavanje neterminala  $S$  važi:

$$\text{First}([eC] \circ \text{Follow}(S)) \cap \text{First}(*i \circ \text{Follow}(S)) = \{[\} \cup \{*\} = \emptyset.$$

Kako je presek ova dva skupa prazan skup, uslov je ispunjen za ovaj neterminal.

Za pravila kojima se preslikava neterminal  $B$  se dobija:

$$\begin{aligned} \text{First}(SB \circ \text{Follow}(B)) \cap \text{First}(\varepsilon \circ \text{Follow}(B)) &= \\ \{[, *] \cap \text{Follow}(B) &= \\ \{[, *] \cap \{\leftarrow\} &= \emptyset. \end{aligned}$$

Za neterminal  $C$  važi:

$$\begin{aligned} \text{First}(eC \circ \text{Follow}(C)) \cap \text{First}(\varepsilon \circ \text{Follow}(C)) &= \\ \{e\} \cap \text{First}(\text{Follow}(C)) &= \\ \{e\} \cap \text{Follow}(C) &= \\ \{e\} \cap \{\} &= \emptyset. \end{aligned}$$

Za sve razmatrane parove dobijamo da je presek First funkcija pazan skup što znači da zadata gramatika pripada skupu LL-1. gramatika.

Sintaksna tabela kod LL-1 gramatika sa  $\varepsilon$ -pravilima je nešto modifikovana u odnosu na prethodne definicije. Naime, sada je problem gde u tablici treba smestiti  $\varepsilon$ -pravila. Zbog toga je sintaksna tabela definisana na sledeći način:

$$\text{Ts}(A, a) = \begin{cases} pop, & \text{ako je } A = a, a \in \mathbf{V}_t, \\ acc, & \text{ako je } A = \# \wedge a \in \#, \\ (\alpha, i), & \text{ako je } a \in \text{First}(\alpha) \text{ i } A \rightarrow \alpha \text{ je } i\text{-to pravilo,} \\ & \text{ili je } a \in \text{Follow}(A) \text{ i } A \rightarrow \varepsilon \text{ je } i\text{-to pravilo, za } A \in \mathbf{V}_n \\ err, & \text{u svim ostalim slučajevima.} \end{cases}$$

Prema ovoj definiciji sledi da se  $\varepsilon$ -pravila upisuju u polja određena vrstom koja odgovara neterminalnom simbolu koji se preslikava u  $\varepsilon$  i kolonama svih terminalnih simbola koji pripadaju funkciji Follow za taj neterminalni simbol.

**Primer 6.11** Sintaksna tabela za gramatiku iz primera 6.10

Za gramatiku iz primera 6.10 dobija se sintaksna tabela prikazana u nastavku.

VRH MAGACINA	ULAZNI SIMBOL					
	$i$	$\leftarrow$	$e$	[	]	$*$ #
$A$	$(iB \leftarrow e, 1)$					
$B$		$(\varepsilon, 3)$		$(SB, 2)$		$(SB, 2)$
$S$				$([eC], 4)$		$(*i, 5)$
$C$			$(eC, 6)$		$(\varepsilon, 7)$	
$i$	<i>pop</i>					
$\leftarrow$		<i>pop</i>				
$e$			<i>pop</i>			
[				<i>pop</i>		
]					<i>pop</i>	
*						<i>pop</i>
#						<i>acc</i>

Pravilo (3)  $B \rightarrow \varepsilon$  nalazi se u koloni  $\leftarrow$  zato što ovaj simbol pripada skupu  $\text{Follow}(B)$ , što se vidi iz pravila (1). Slično, pravilo (7)  $C \rightarrow \varepsilon$ , nalazi se u koloni  $]$  zato što taj simbol pripada skupu  $\text{Follow}(C)$ , prema pravilu (4).

Sintaksna tabela se koristi u algoritmu za analizu na isti način kao i kod prethodno razmatranih jednostavnijih slučajeva LL-1 gramatika (primer 6.12).

#### Primer 6.12 Sintaksna analiza za gramatiku iz primera 6.10

Za gramatiku iz primera 6.10 i reč  $i[e] \leftarrow e\#$  postupak sintaksne analize je:

$$\begin{aligned}
(i[e] \leftarrow e\#, A\#, e) &\rightarrow (i[e] \leftarrow e\#, iB \leftarrow \#, 1) \\
&\rightarrow ([e] \leftarrow e\#, B \leftarrow e\#, 1) \\
&\rightarrow ([e] \leftarrow e\#, SB \leftarrow e\#, 12) \\
&\rightarrow ([e] \leftarrow e\#, [eC]B \leftarrow e\#, 124) \\
&\rightarrow [e] \leftarrow e\#, eC]B \leftarrow e\#, 124) \\
&\rightarrow [] \leftarrow e\#, C]B \leftarrow e\#, 124) \\
&\rightarrow [] \leftarrow e\#, ]B \leftarrow e\#, 1247) \\
&\rightarrow (\leftarrow e\#, B \leftarrow e\#, 1247) \\
&\rightarrow (\leftarrow e\#, \leftarrow e\#, 12473) \\
&\rightarrow (e\#, e\#, 12473) \\
&\rightarrow (\#, \#, 12473).
\end{aligned}$$

Reč je prepoznata i za njeno izvođenje koristi se niz smena (1), (2), (4), (7) i (3).

Da zaključimo – ukoliko gramatika kojom je opisan jezik ima svojstva LL-1 gramatike, moguće je vršiti *top-down* analizu bez povratnih koraka, odnosno mnogo efikasnije u odnosu na osnovni *top-down* algoritam.

## 6.5 Rešeni zadaci

**Zadatak 1** Ispitati da li gramatika

$$\begin{aligned} G = (\{ & Blok, NizNar, Naredba, Dodela, Izraz \}, \\ & \{ \mathbf{begin}, \mathbf{end}, \mathbf{ID}, \mathbf{CONST}, ;, +, := \}, \\ & Blok, P), \end{aligned}$$

iz primera 5.8, koja je posle eliminisanja leve rekurzije zadata sledećim skupom smena,

$$Blok \rightarrow \mathbf{begin} \ NizNar \ \mathbf{end} \quad (1)$$

$$NizNar \rightarrow Naredba \ NizNar' \quad (2)$$

$$NizNar' \rightarrow ; \ Naredba \ NizNar' \quad (3)$$

$$NizNar' \rightarrow \varepsilon \quad (4)$$

$$Naredba \rightarrow Dodela \quad (5)$$

$$Naredba \rightarrow Blok \quad (6)$$

$$Dodela \rightarrow \mathbf{id} := Izraz \quad (7)$$

$$Izraz \rightarrow \mathbf{const} \ Izraz' \quad (8)$$

$$Izraz' \rightarrow + \mathbf{const} \ Izraz' \quad (9)$$

$$Izraz' \rightarrow \varepsilon \quad (10)$$

zadovoljava uslove LL-1 gramatike.

*Rešenje.* Vršimo analizu skupova smena koje na levoj strani imaju isti neterminalni simbol.

Prvi par smena koje treba ispitati je par (3) i (4).

$$\text{First}(\ ; \ Naredba \ NizNar') = \{ ; \}$$

$$\text{First}(\varepsilon) = \{ \varepsilon \}$$

$$\text{First}(\ ; \ Naredba \ NizNar') \cap \text{First}(\varepsilon) = \emptyset$$

Samo jedna od navedenih smena se preslikava u prazan skup.

$$\text{Follow}(NizNar') = \text{Follow}(NizNar) = \{ \mathbf{end} \}$$



$$\text{First}(\text{; Naredba NizNar}') \cap \text{Follow}(\text{NizNar}) = \emptyset$$

Drugi par smena je par (5) i (6).

$$\begin{aligned}\text{First}(\text{Dodela}) &= \{\text{id}\} \\ \text{First}(\text{Blok}) &= \{\text{begin}\} \\ \text{First}(\text{Dodela}) \cap \text{First}(\text{Blok}) &= \emptyset\end{aligned}$$

Nijedna od navedenih smena se ne preslikava u prazan skup.

Treći par smena je par (9) i (10).

$$\begin{aligned}\text{First}(+\text{const Izraz}') &= \{+\} \\ \text{First}(\varepsilon) &= \{\varepsilon\} \\ \text{First}(+\text{const Izraz}') \cap \text{First}(\varepsilon) &= \emptyset\end{aligned}$$

Samo jedna od navedenih smena se preslikava u prazan skup.

$$\begin{aligned}\text{Follow}(\text{Izraz}') &= \{\text{;, end}\} \\ \text{First}(+\text{const Izraz}') \cap \text{Follow}(\text{Izraz}') &= \emptyset\end{aligned}$$

Gramatika **G** ispunjava uslove LL-1 gramatike.

**Zadatak 2** Odrediti tablicu sintaksne analize za LL-1 gramatiku zadatu sledećim skupom pravila:

$$\begin{array}{ll} S \rightarrow aT & T \rightarrow SA \mid A \\ A \rightarrow bB \mid c & B \rightarrow d \mid \varepsilon \end{array}$$

*Rešenje.* Sintaksna tabela za zadatu gramatiku data je u tabeli 6.1.

## 6.6 Pitanja

1. Dati definiciju proste LL-1 gramatike.
2. Kako je definisana sintaksna tabela proste LL-1 gramatike?
3. Objasniti algoritam za sintaksnu analizu proste LL-1 gramatike.
4. Objasniti pojam leve faktorizacije.
5. Dati definiciju funkcije First.
6. Dati definiciju funkcije Follow.
7. Dati definiciju LL-1 gramatika bez  $\varepsilon$  pravila.
8. Kako se popunjava sintaksna tabela kod LL-1 gramatika bez  $\varepsilon$  pravila?

TABELA 6.1

VRH MAGACINA	ULAZNI SIMBOL				
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	#
<i>S</i>	$(aT, 1)$				
<i>T</i>	$(SA, 2)$	$(A, 3)$	$(A, 3)$		
<i>A</i>		$(bB, 4)$	$(c, 5)$		
<i>B</i>		$(\varepsilon, 7)$	$(\varepsilon, 7)$	$(d, 6)$	$(\varepsilon, 7)$
<i>a</i>	<i>pop</i>				
<i>b</i>		<i>pop</i>			
<i>c</i>			<i>pop</i>		
<i>d</i>				<i>pop</i>	
#					<i>acc</i>

9. Dati definiciju LL-1 gramatika sa  $\varepsilon$  pravilima.
10. Kako se popunjava sintaksna tabela kod LL-1 gramatika sa  $\varepsilon$  pravilima?

## 6.7 Zadaci

1. Za LL-1 gramatiku zadatu skupom smena

$$E \rightarrow (E) T \mid i T \qquad T \rightarrow +E \mid *E \mid \varepsilon$$

generisati tablicu sintaksne analize.

2. Gramatika **G** zadata je skupom smena

$$S \rightarrow aaSd \mid aAd \qquad A \rightarrow bAc \mid b.$$

Transformisati gramatiku **G** u LL-1 gramatiku, odrediti tablicu sintaksne analize i korišćenjem formirane tabele proveriti da li niz #aaababcd# pripada jeziku koji je određen ovom gramatikom.

3. Utvrditi da li je gramatika zadata skupom smena

$$A \rightarrow aBC \mid bB \qquad B \rightarrow CbA \mid c \mid \varepsilon \qquad C \rightarrow aBa \mid cBaC \mid \varepsilon$$

LL-1 gramatika. Ako jeste, formirati odgovarajuću tablicu sintaksne analize.

4. Generisati LL-1 sintaksnu tabelu gramatike koja je definisana skupom smena

$$E \rightarrow (E) T \mid \mathbf{const} T \qquad T \rightarrow +E \mid *E \mid \varepsilon.$$

Korišćenjem generisane tabele proveriti da li izraz

$$\# \mathbf{const} * (\mathbf{const} + \mathbf{const}) \#$$

pripada jeziku date gramatike.

5. Transformisati gramatiku **G** zadatu skupom smena

$$Ulaz \rightarrow \mathbf{READ}(NizPromenljivih)$$

$$NizPromenljivih \rightarrow NizPromenljivih, \mathbf{ID} \mid \mathbf{ID}$$

u gramatiku tipa LL-1 i kreirati odgovarajuću tablicu sintaksne analize.

Korišćenjem kreirane tabele proveriti da li izraz

$$\# \mathbf{READ}(\mathbf{ID}, \mathbf{ID}, ) \#$$

pripada jeziku date gramatike.

6. Proveriti da li gramatika zadata sledećim skupom smena **P** zadovoljava uslove LL-1 gramatika i, ako zadovoljava, kreirati njenu sintaksnu tabelu.

$$\begin{array}{ll} S \rightarrow aT & T \rightarrow SA \mid A \\ A \rightarrow bB \mid c & B \rightarrow d \mid \varepsilon \end{array}$$

7. Proveriti da li je gramatika zadata skupom smena

$$\begin{array}{ll} S \rightarrow aT & T \rightarrow SA \mid A \\ A \rightarrow bB \mid c & B \rightarrow d \mid \varepsilon \end{array}$$

tipa LL-1. Ako jeste, kreirati njenu sintaksnu tabelu.

8. LL-2 gramatiku koja je zadata skupom smena

$$S \rightarrow aaSd \mid abAd \qquad S \rightarrow baAc \mid bc$$

transformisati u LL-1 gramatiku, odrediti sintaksnu tabelu i korišćenjem formirane tabele proveriti da li niz

$$\#aaabbcd\#$$

pripada jeziku koji je određen ovom gramatikom.

9. Formirati LL-1 sintaksnu tabelu gramatike zadate skupom smena

$$\begin{array}{l} S \rightarrow \mathbf{if} b \mathbf{then} S E \mid a \\ E \rightarrow \mathbf{else} S \mid \varepsilon. \end{array}$$

Korišćenjem formirane tabele proveriti da li niz

$$\# \mathbf{if} b \mathbf{then} a \mathbf{else if} b \mathbf{then} a \#$$

pripada jeziku koji je definisan ovom gramatikom.



## Glava 7

# Analiza zasnovana na relacijama prvenstva

U ovom poglavlju biće opisan poseban tip beskonteksnih gramatika koje omogućavaju *bottom-up* sintaksnu analizu bez vraćanja unatrag koja se zasniva na relacijama prioriteta. Ove gramatike nalaze primenu kod definisanja aritmetičkih izraza, relacija i sl. koji kao delovi naredbi dodeljivanja čine veliki deo programskog jezika.

### 7.1 Relacije prvenstva

**Definicija:** Operatorske gramatike su beskonteksne gramatike kod kojih ne postoje pravila u kojima se u reči na desnoj strani javljaju dva neterminalna simbola jedan do drugog, odnosno ne postoje pravila oblika

$$C \rightarrow \alpha AB\beta \quad \text{gde je} \quad A, B, C \in \mathbf{V}_n \text{ i } \alpha, \beta \in \mathbf{V}^*.$$

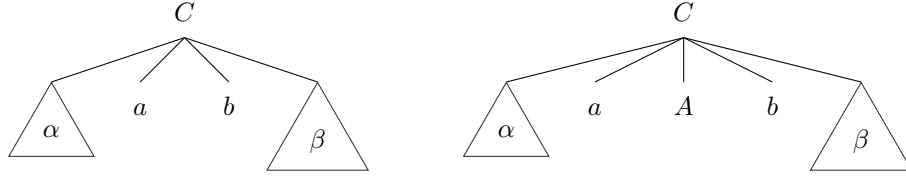
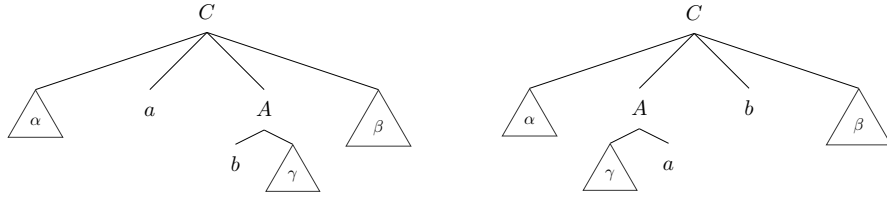
Da bi mogao uspešno da se sprovede postupak sintaksne analize, između terminalnih simbola operatorske gramatike definišu se relacije prvenstva (prioriteta).

Postoje tri osnovne relacije prioriteta:  $<$ ,  $>$  i  $\doteq$ .

#### 7.1.1 Relacija istog prioriteta

Terminalni simboli  $a$  i  $b$  su u relaciji istog prioriteta, u oznaci  $a \doteq b$ , ako i samo ako postoji smena oblika

$$C \rightarrow \alpha ab\beta \text{ ili } C \rightarrow \alpha aAb\beta, \quad \text{gde je} \quad C, A \in \mathbf{V}_n, \quad a, b \in \mathbf{V}_t, \quad \alpha, \beta \in \mathbf{V}^*.$$

SLIKA 7.1: Relacija istog prioriteta ( $a \doteq b$ )SLIKA 7.2: Relacija nižeg ( $a < b$ ) i višeg ( $a > b$ ) prioriteta

Ako se to predstavi na sintaksnom stablu, onda znači da se terminalni simboli  $a$  i  $b$  pojavljuju na istom nivou sintaksnog stabla (slika 7.1).

### 7.1.2 Relacija nižeg prioriteta

Terminalni simbol  $a$  je nižeg prioriteta u odnosu na terminalni simbol  $b$ , u oznaci  $a < b$ , ako i samo ako postoji smena oblika:

$$C \rightarrow \alpha a A \beta \text{ i } A \xrightarrow{*} b \gamma \text{ ili } A \xrightarrow{*} D b \gamma$$

gde je  $C, A, D \in \mathbf{V}_n$ ,  $a, b \in \mathbf{V}_t$ ,  $\alpha, \beta, \gamma \in \mathbf{V}^*$ . Predstavljeno na sintaksnom stablu to znači da se terminalni simbol  $a$  pojavljuje levo i na višem nivou u odnosu na terminalni simbol  $b$  (slika 7.2, levo).

### 7.1.3 Relacija višeg prioriteta

Terminalni simbol  $a$  je višeg prioriteta u odnosu na terminalni simbol  $b$ , u oznaci  $a > b$ , ako i samo ako postoji smena oblika:

$$C \rightarrow \alpha A b \beta \text{ i } A \xrightarrow{*} \gamma a \text{ ili } A \xrightarrow{*} \gamma a D,$$

gde je  $C, A, D \in \mathbf{V}_n$ ,  $a, b \in \mathbf{V}_t$ ,  $\alpha, \beta, \gamma \in \mathbf{V}^*$ . Posmatrano na sintaksnom stablu to znači da se terminalni simbol  $a$  pojavljuje levo od terminalnog simbola  $b$  ali na nekom nižem nivou (slika 7.2, desno).

## 7.2 Operatorska gramatika prvenstva

Operatorska gramatika u kojoj za svaka dva terminalna simbola važi najviše jedna relacija prvenstva naziva se operatorska gramatika prvenstva. Kod ovih gramatika moguće je primeniti algoritam za *bottom-up* analizu u kome neće biti povratnih petlji.

U postupaku sintaksne analize operatorskih gramatika prvenstva koristi se pomoćna sintaksna tabela (operatorska tabela prvenstva). To je tabela sa onoliko vrsta i kolona koliko je terminalnih simbola azbuke (uključujući i granični simbol). Elementi operatorske tabele prvenstva su relacije prvenstva, pri čemu se simbol sa leve strane relacije uzima kao oznaka vrste, a simbol sa desne strane relacije kao oznaka kolone.

### Primer 7.1 Relacije prvenstva

Za operatorsku gramatiku prvenstva definisanu skupom smena

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow \text{id}$$

odrediti tablicu prvenstva.

**Određivanje relacija prvenstva.** Da bi se odredile relacije prvenstva između graničnih simbola i ostalih simbola gramatike uvodi se pomoćna smena oblika

$$S' \rightarrow \#S\#$$

gde je  $S$  startni simbol gramatike.

Kako je  $E$  startni simbol gramatike iz našeg primera, uvodi se smena

$$E' \rightarrow \#E\#.$$

Kako bismo uočili relacije prvenstva, polazimo od startnog simbola gramatike i generišemo fraze tog simbola, odnosno reči koje se izvode od tog simbola. Pri tome uočavamo relacije između terminalnih simbola. Taj postupak se nastavlja daljim uvođenjem smena sve dok se ne uoče sve relacije između terminalnih simbola (tabela 7.1). Napomenimo da su u toj tabeli prikazane smene koje su primenjene, niz koji se generiše, deo sintaksnog stabla i relacija koja se izvodi. U ovom postupku se primenjuje levo izvođenje.

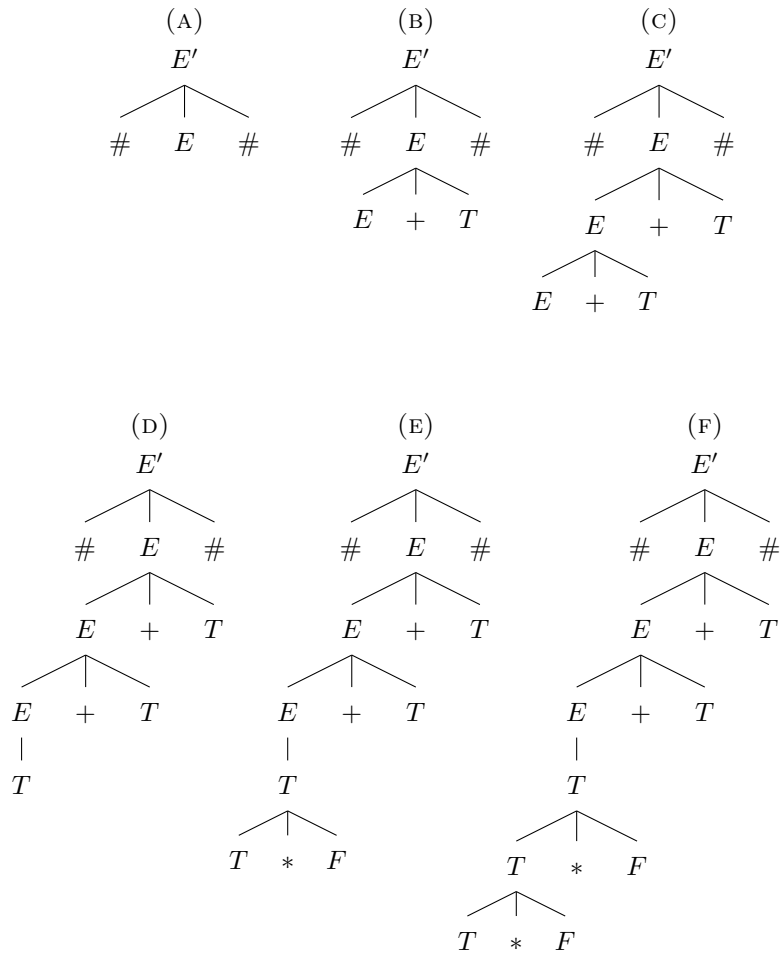
Odgovarajuća operatorska tabela prvenstva je data u tabeli 7.2.

TABELA 7.1.: Otkrivanje relacije prvenstva uz primer 7.1

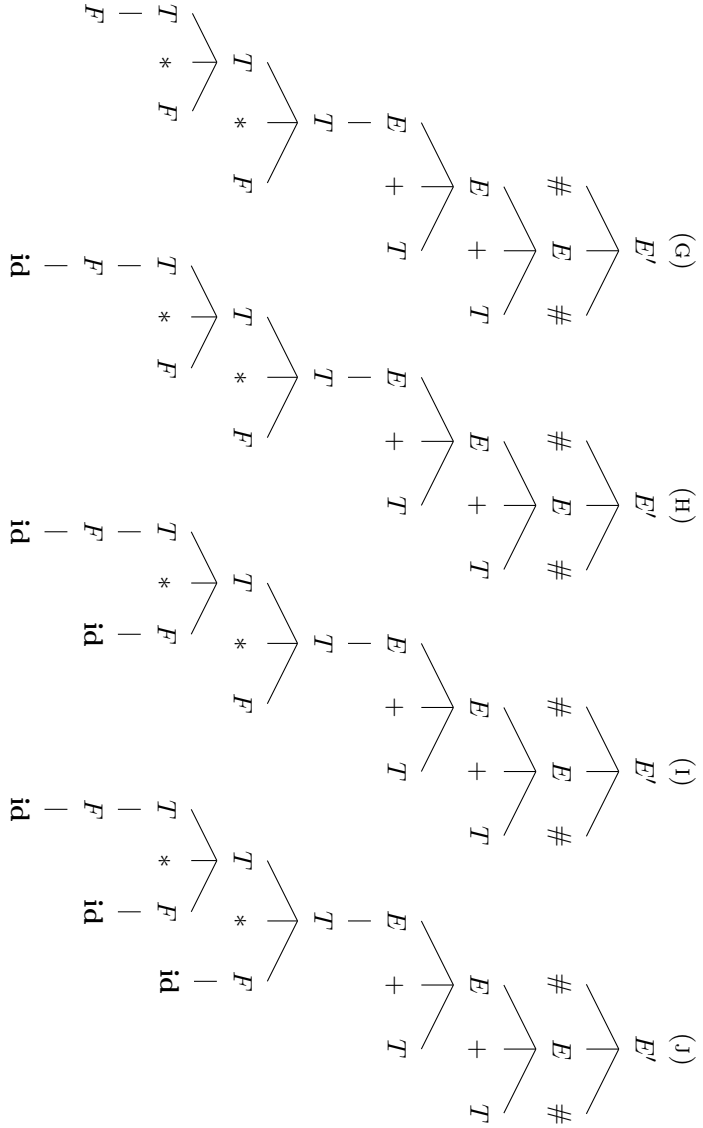
PRIMENJENA SMENA	IZVEDENI NIZ	STABLO	UOČENE RELACIJE
$E' \rightarrow \#E\#$	$\#E\#$	slika 7.3 (A)	$\# \doteq \#^a$
$E \rightarrow E + T$	$\#E + T\#$	slika 7.3 (B)	$\# < +, + > \#$
$E \rightarrow E + T$	$\#E + T + T\#$	slika 7.3 (C)	$+ > +$
$E \rightarrow T^b$	$\#T + T + T\#$	slika 7.3 (D)	nema
$T \rightarrow T * F$	$\#T * F + T + T\#$	slika 7.3 (E)	$\# < *, * > +$
$T \rightarrow T * F$	$\#T * F * F + T + T\#$	slika 7.3 (F)	$* > *$
$T \rightarrow F$	$\#F * F * F + T + T\#$	slika 7.4 (G)	nema
$F \rightarrow \text{id}$	$\#\text{id} * F * F + T + T\#$	slika 7.4 (H)	$\# < \text{id}, \text{id} > *$
$F \rightarrow \text{id}$	$\#\text{id} * \text{id} * F + T + T\#$	slika 7.4 (I)	$* < \text{id}, \text{id} > *$
$F \rightarrow \text{id}$	$\#\text{id} * \text{id} * \text{id} + T + T\#$	slika 7.4 (J)	$i > +$
$T \rightarrow T * F$	$\#\text{id} * \text{id} * \text{id} + T * F + T\#$	slika 7.5 (K)	$+ < *, * < +$
$T \rightarrow F$	$\#\text{id} * \text{id} * \text{id} + F * F + T\#$	slika 7.5 (L)	nema
$F \rightarrow \text{id}$	$\#\text{id} * \text{id} * \text{id} + \text{id} * F + T\#$	slika 7.5 (M)	$+ < \text{id}$
$F \rightarrow \text{id}$	$\#\text{id} * \text{id} * \text{id} + \text{id} * \text{id} + T\#$	slika 7.6 (N)	$\text{id} > +$
$T \rightarrow T * F$	$\#\text{id} * \text{id} * \text{id} + \text{id} * \text{id} + T * F\#$	slika 7.6 (O)	$* > \#$
$T \rightarrow F, F \rightarrow \text{id}, F \rightarrow \text{id}$	$\#\text{id} * \text{id} * \text{id} + \text{id} * \text{id} + \text{id} * \text{id}\#$	slika 7.6 (P)	$i > \#$

<sup>a</sup>ova relacija ne utiče na postupak analize<sup>b</sup>Ako se još jednom primeni prva smena za preslikavanje simbola  $\#E\#$ , ne bi se dobile nove relacije; zato primenjujemo  $E \rightarrow T$

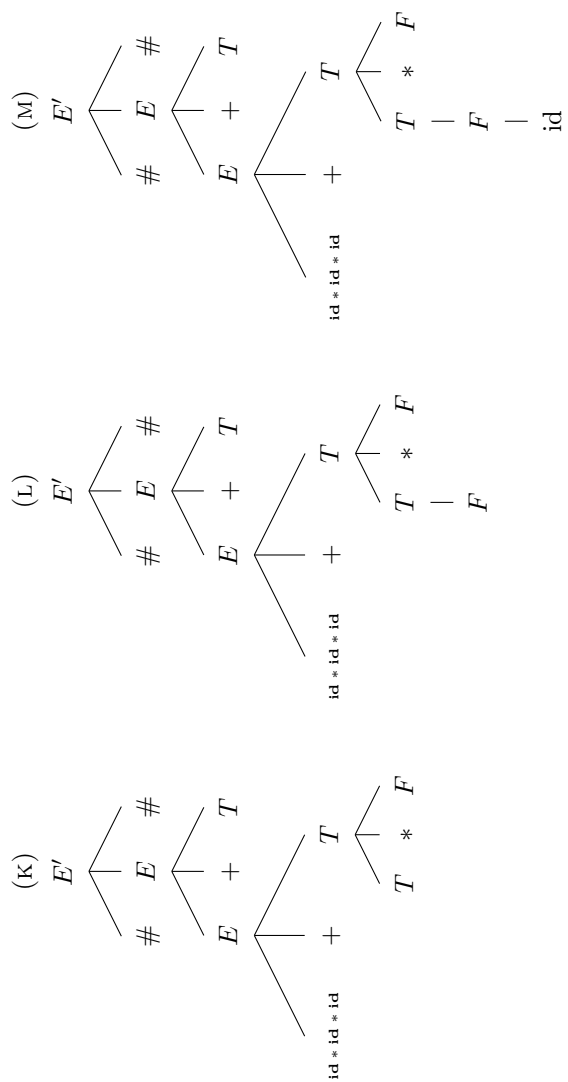




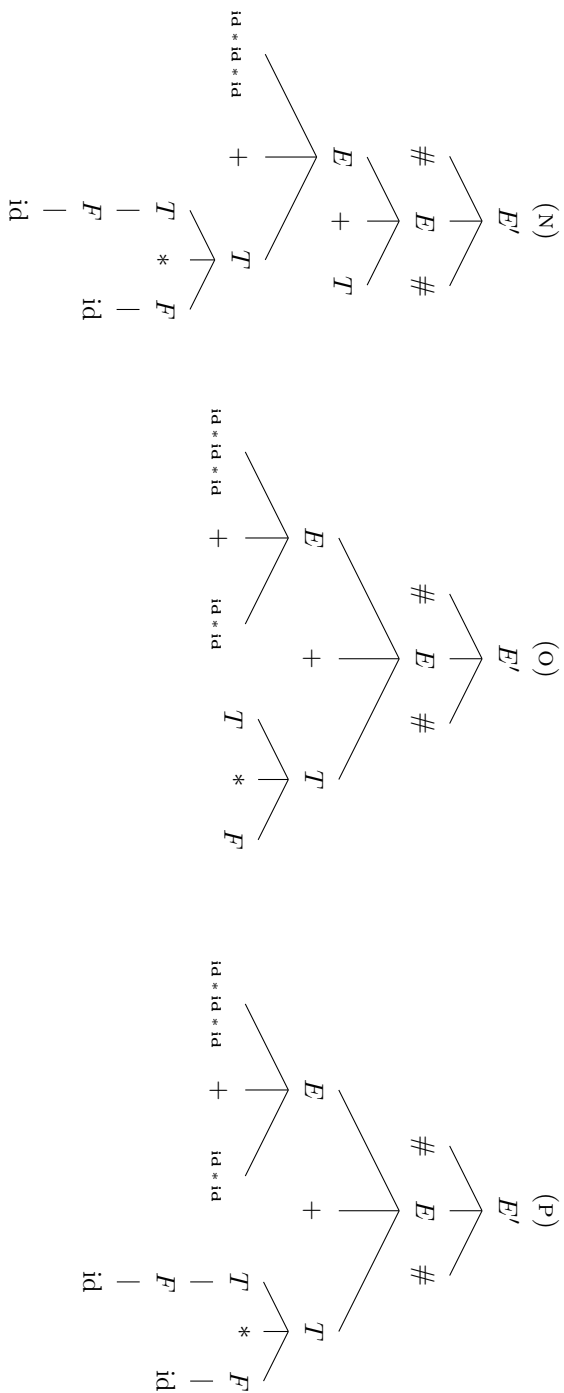
SLIKA 7.3: Sintaksna stabla uz tabelu 7.2 (prvi deo)



Slika 7.4: Sintakсна stabla uz tabelu 7.2 (drugi deo)



SLIKA 7.5: Sintaksna stabla uz tabelu 7.2 (treći deo)



Slika 7.6: Sintakсна stabla uz tabelu 7.2 (četvrti deo)

TABELA 7.2: Operatorska tabela iz primera 7.1

	id	+	*	#
id		>	>	>
+	<	>	<	>
*	<	>	>	>
#	<	<	<	=

### 7.2.1 Postupak sintaksne analize

Kada je jezik definisan operatorskom gramatikom prvenstva moguće je definisati *bottom-up* postupak sintaksne analize bez povratnih koraka. U postupku sintaksne analize, prati se stanje ulaznog niza (**t**) i pomoćnog magacina ( $\alpha$ ) i izvršavaju se sledeći koraci:

Ulazni niz (**t**) dopuni se graničnim simbolom (**#**) sa desne strane i u pomoćni stek ( $\alpha$ ) takođe se upiše granični simbol (**#**). Zatim se na osnovu relacije prvenstva koja važi između poslednjeg terminalnog simbola u steku i sledećeg simbola u ulaznom nizu određuje akcija koja će se izvršiti.

1. Ako važi relacija  $<$  ili relacija  $=$ , tekući terminalni simbol iz ulaznog niza se upisuje u stek  $\alpha$  i čita se novi simbol iz ulaznog niza. Uz terminalni simbol upisuje se i oznaka relacije prvenstva koju ima sa prethodnim simbolom.
2. Ako važi relacija  $>$  treba ići dublje u stek i tražiti prvi terminal u steku koji je  $<$  od svog sledbenika u steku i proveriti da li fraza sa vrha steka (od uočenog simbola do vrha, fraza između simbola  $<$  i  $>$ ) predstavlja desnu stranu neke smene. Ako takva smena postoji, treba izvršiti redukciju, tj. umesto uočene fraze u stek ubaciti uniformnu oznaku neterminalnog simbola ( $P$ ). U suprotnom, ako takva fraza ne postoji kao desna strana smene, u zapisu postoji greška i dalju analizu treba prekinuti. Kada se redukcija uspešno izvede treba ispitati da li je sledeći ulazni simbol **#**, a sadržaj steka  $\#P$ . U tom slučaju ceo ulazni niz redukovan, tj. prepoznat.
3. Ako ne postoji ni jedna od navedenih relacija znači da u zapisu postoji greška i dalju analizu treba prekinuti.

Osnovna ideja kod ovog postupka sinteze je da se identifikuju fraze koje su višeg prioriteta u odnosu na okruženje i da se one prve redukuju. Ukoliko je fraza niže u sintaksnom stablu ona je višeg prioriteta redukuje se pre simbola iz okruženja.

Kompletni sintaksni analizator dat je u vidu pseudokoda (algoritam 1).

```

 $t \leftarrow t + \#;$ 
 $\alpha_0 \leftarrow \#;$ 
 $SP \leftarrow 0;$  /* SP je pokazivač na vrh magacina  $\alpha$  */
 $prepoznat \leftarrow 0;$ 
 $greška \leftarrow 0;$ 
 $next \leftarrow nextlex(t);$  /* poziv leksičkog analizatora */
repeat
  if  $\alpha_{SP} \in V_t \cup \{\#\}$  then
    |  $i \leftarrow SP$ 
  end
  else
    |  $i \leftarrow SP - 1;$  /*  $i$  pokazuje na poslednji terminalni simbol u
    |   magacinu */
  end
  if  $\alpha_i < next \vee \alpha_i = next$  then
    |  $SP \leftarrow SP + 1;$ 
    |  $\alpha_{SP} = next;$ 
    |  $next \leftarrow nextlex(t);$ 
  end
  else if  $\alpha_i > next$  then
    repeat
      |  $temp \leftarrow \alpha_i;$ 
      |  $i \leftarrow i - 1;$ 
      | if  $\alpha_i = P$  then
      |   |  $i \leftarrow i - 1;$ 
      | end
    until  $\alpha_i < temp;$ 
    if fraza  $\alpha_{i+1}, \dots, \alpha_{SP}$  postoji u listi desnih strana smena then
      |  $SP \leftarrow i + 1;$ 
      |  $\alpha_{SP} \leftarrow P;$ 
      | if  $SP = 2 \wedge next = \#$  then
      |   |  $prepoznat \leftarrow 1$ 
      | end
    end
    else
      |  $greška \leftarrow 1$ 
    end
  end
  else
    |  $greška \leftarrow 1$ 
  end
until  $prepoznat \wedge greška;$ 
return  $prepoznat;$ 

```

**Algorithm 1:** Pseudokod sintaksnog analizatora operatorskih gramatika prvenstva.

**Primer 7.2** Postupak sintaksne analize

Za operatorsku gramatiku prvenstva razmatranu u primeru 7.1, postupak sintaksne analize izraza  $\#id + id * id\#$  prikazan je u sledećoj tabeli.

RADNI MAGACIN $\alpha$	ULAZNI NIZ	RELACIJA PRVENSTVA
$\#$	$\boxed{id} + id * id \#$	$\# < id$
$\# < \boxed{id}$	$id \boxed{+} id * id \#$	$id > +$
$\boxed{\#} < P$	$id \boxed{+} id * id \#$	$\# < +$
$\# < P \boxed{+}$	$id + \boxed{id} * id \#$	$+ < id$
$\# < P + < \boxed{id}$	$id + id \boxed{*} id \#$	$id > *$
$\# < P \boxed{+} P$	$id + id \boxed{*} id \#$	$+ < *$
$\# < P + < P \boxed{*}$	$id + id * \boxed{id} \#$	$* < id$
$\# < P + < P * < \boxed{id}$	$id + id * id \boxed{\#}$	$id > \#$
$\# < P + < P \boxed{*} < P$	$id + id * id \boxed{\#}$	$* > \#$
$\# < P \boxed{+} P$	$id + id * id \boxed{\#}$	$+ > \#$
$\# P$	$id + id * id \#$	izraz je prepoznat

**7.2.2 Funkcije prvenstva**

Memorija potrebna za pamćenje tabela prvenstva je  $(n + 1) \cdot (n + 1)$  (gde je  $n$  broj terminalnih simbola gramatike). Kod gramatika sa mnogo terminalnih simbola ova tabela može da bude veoma velika i da zahteva mnogo memorijskog prostora. Zato se umesto tabele prvenstva često relacije pamte preko funkcija prvenstva. Za svaki terminalni simbol gramatike definišu se po dve celobrojne funkcije  $f$  i  $g$  koje zadovoljavaju sledeće uslove:

$$a < b \Rightarrow f(a) < g(b),$$

$$a = b \Rightarrow f(a) = g(b),$$

$$a > b \Rightarrow f(a) > g(b).$$

Uočimo da je memorijski prostor potreban za pamćenje funkcija prvenstva  $2 \cdot (n + 1)$ .

Za određivanje funkcije prvenstva formira se orijentisani graf koji ima  $2n$  čvorova. Potezi u grafu se određuju na sledeći način:

- Ako važi relacija  $a > b$ , postoji poteg od čvora  $f(a)$  prema čvoru  $g(b)$ .

- Ako važi relacija  $a < b$ , postoji poteg od čvora  $g(b)$  prema čvoru  $f(a)$ .
- Ako važi relacija  $a = b$ , postoje potezi od čvora  $f(a)$  prema čvoru  $g(b)$  i od čvora  $g(b)$  prema čvoru  $f(a)$ .

Vrednost funkcije prvenstva se određuje kao dužina najdužeg puta koji polazi iz čvora koji odgovara toj funkciji. Pod dužinom puta podrazumeva se broj čvorova kroz koje put prolazi uključujući i sam taj čvor.

### Primer 7.3 Funkcije prvenstva

Graf za određivanje vrednosti funkcija prvenstva za gramatiku razmatranu u primeru 7.1 predstavljen je na slici 7.7.

Na primer, najduži put koji polazi iz čvora  $g(i)$  je put

$$g(i) \rightarrow f(*) \rightarrow g(*) \rightarrow f(+) \rightarrow g(+) \rightarrow f(\#),$$

pa je vrednost funkcije  $g(i) = 6$ .

	$i$	$+$	$*$	$\#$
$F$	5	3	5	0
$G$	6	2	4	0

## 7.3 Opšta pravila prioriteta

Kako su operatorske gramatike prvenstva pogodne za opis izraza u programskim jezicima, gramatike prvenstva treba da budu tako definisane da zadržavaju prioritete operatora koji važe u samom programskom jeziku. Neka opšta pravila prikazana su u tabeli 7.3, pri čemu važe sledeće relacije:

1. Ako je operator  $Q_1$  većeg prioriteta u odnosu na operator  $Q_2$ , tada relacije prioriteta treba da budu postavljene tako da je

$$Q_1 > Q_2 \quad \text{i} \quad Q_2 < Q_1.$$

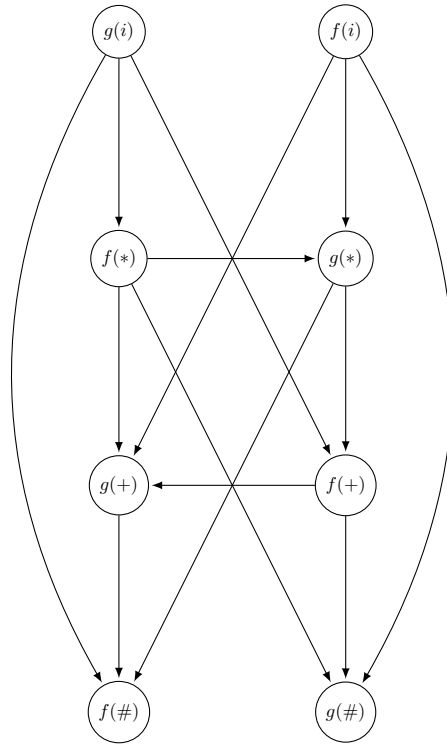
Na primer, za operatore  $*$  i  $+$  treba da važi  $* > +$  i  $+ < *$ .

Time se postiže da se u rečeničnim formama  $E + E * E + E$  prvo redukuje fraza  $E * E$ .

2. Ako su  $Q_1$  i  $Q_2$  operatori istog prioriteta (to važi i u slučaju kada su u pitanju isti operatori), tada relacija prioriteta treba da bude postavljena tako da je

$$Q_1 > Q_2 \quad \text{i} \quad Q_2 > Q_1, \quad \text{ako su operatori levo asocijativni i}$$





SLIKA 7.7: Graf za određivanje funkcija prvenstva uz primer 7.3

$Q_1 < Q_2$  i  $Q_2 < Q_1$ , ako su operatori desno asocijativni.

Primeri:

$+ > +$        $+ > -$        $- > -$        $- > +$        $** < **$

3. Levi granični simbol je nižeg prioriteta od svih drugih simbola, dok svi ostali simboli višeg prioriteta u odnosu na desni granični simbol, tj.

$$\# < Q \quad \text{i} \quad Q > \#.$$

4. Otvorena zagrada je nižeg prioriteta od ostalih simbola, dok je zatvorena zagrada višeg prioriteta od svih simbola. Takodje, svi ostali operatori su nižeg prioriteta od otvorene zagrade, a višeg prioriteta od zatvorene. Ovim se postiže da se izraz u zagradi redukuje pre konteksta u kome se nalazi, što je i namena zagrada.

$$Q < ( \quad ( < Q \quad ) > Q \quad Q > )$$

**Napomena:** Otvorena i zatvorena zagrada su istog prioriteta jedna u odnosu na drugu dok preioritet između zatvorene i otvorene zagrade ne treba

da postoji, zato što u programskim jezicima između zatvorene zagrade i sledeće otvorene zagrade mora da stoji operator.

5. Identifikator je uvek višeg prioriteta u odnosu na susedne simbole, dok su svu svi ostali simboli nižeg prioriteta od identifikatora.

$$Q < \text{id} \quad \text{id} > Q \quad \# < \text{id} \quad \text{id} > \# \quad ( < \text{id} \quad \text{id} > )$$

TABELA 7.3: Opšte relacije prioriteta

	+	-	*	/	**	id	(	)	#
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
**	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	=	
)	>	>	>	>	>			>	>
#	<	<	<	<	<	<	<		=

## 7.4 Gramatike prvenstva

Relacije prvenstva moguće je definisati tako da se odnose i na neterminalne simbole. Pri tome će važiti:

- $s \dot{=} t$  ako postoji smena oblika  $A \rightarrow \alpha st\beta$  gde su  $s, t \in \mathbf{V}$ ,  $\alpha, \beta \in \mathbf{V}^*$  i  $A \in \mathbf{V}_n$ .
- $s \dot{<} t$  ako postoji smena oblika  $A \rightarrow \alpha sB\beta$  i  $B \xrightarrow{*} t\gamma$  gde su  $s, t \in \mathbf{V}$ ,  $\alpha, \beta, \gamma \in \mathbf{V}^*$  i  $A, B \in \mathbf{V}_n$ .
- $s \dot{>} t$  ako postoji smena oblika  $A \rightarrow \alpha Bt\beta$  i  $B \xrightarrow{*} \gamma s$  ili ako postoji smena  $A \rightarrow \alpha XZ\beta$  i  $X \xrightarrow{*} \gamma_1 s$  i  $y \xrightarrow{*} t\gamma_2$ , gde su  $s, t \in \mathbf{V}$ ,  $\alpha, \beta, \gamma, \gamma_1, \gamma_2 \in \mathbf{V}^*$  i  $A, B, X, Z \in \mathbf{V}_n$ .

Beskontekсна gramatika kod koje se između bilo koja dva simbola može postaviti najviše jedna relacija prvenstva naziva se gramatika prvenstva.

Za svaku gramatiku prvenstva može se generisati jednoznačno popunjena tablica prvenstva u koju se upisuju relacije prvenstva. Ova tablica ima onoliko vrsta

i kolona koliko gramatika ima ukupno simbola i terminalnih i neterminalnih. Tablica prvenstva koristi se u postupku sintaksne analize na isti način kao i kod operatorskih gramatika prvenstva.

#### Primer 7.4 Gramatika prvenstva

Gramatiku zadatu sledećim skupom smena

$$\begin{aligned} \text{Program} &\rightarrow \text{start NizNaredbi end} \\ \text{NizNaredbi} &\rightarrow \text{NizNaredbi ; Naredba} \mid \text{Naredba} \\ \text{Naredba} &\rightarrow \text{ulaz} \mid \text{izlaz} \mid \text{dodela} \end{aligned}$$

transformisaćemo u gramatiku prvenstva i kreirati njenu tablicu prvenstva.

Da bi gramatika bila gramatika prvenstva između svaka dva simbola može da važi najviše jedna relacija prvenstva. Najpre proveravamo ovaj uslov.

Na osnovu prve smene gramatike može se zaključiti da postoji relacija:

$$\text{start} \dot{=} \text{NizNaredbi}.$$

Ukoliko se *NizNaredbi* preslika po prvoj smeni za taj neterminalni simbol, dobija se niz oblika:

$$\text{start NizNaredbi ; Naredba end},$$

odakle se može zaključiti da važi relacija:

$$\text{start} < \text{NizNaredbi}.$$

To znači da relacija prvenstva između simbola **start** i *NizNaredbi* nije jednoznačno definisana. Ovakva situacija će se javiti uvek kada postoji smena oblika  $X \rightarrow \beta A \gamma$  i uz nju postoji levo rekurzivna smena  $A \rightarrow A \alpha$ , gde  $\beta$  nije prazan niz; ili postoji smena  $X \rightarrow \beta A \gamma$ , gde  $\gamma$  nije prazan niz i uz nju desno rekurzivna smena  $A \rightarrow \alpha A$ . U tom slučaju potrebno je izvršiti sledeće transformacije. Skup smena

$$X \rightarrow \beta A \gamma \qquad A \rightarrow A \alpha$$

se zamenjuje smenama

$$X \rightarrow \beta A \gamma \qquad A \rightarrow A' \qquad A' \rightarrow A' \alpha$$

Odnosno, smene

$$X \rightarrow \beta A \gamma \qquad A \rightarrow \alpha A$$

zamenjuju se sa

$$X \rightarrow \beta A \gamma \qquad A \rightarrow A' \qquad A' \rightarrow \alpha A$$

Transformisana gramatika iz našeg primera biće:

$$\begin{aligned} \textit{Program} &\rightarrow \mathbf{start} \textit{NizNaredbi} \mathbf{end} \\ \textit{NizNaredbi} &\rightarrow \textit{NizNaredbi}' \\ \textit{NizNaredbi} &\rightarrow \textit{NizNaredbi}' ; \textit{NizNaredbi} \mid \textit{NizNaredbi} \\ \textit{Naredba} &\rightarrow \mathbf{ulaz} \mid \mathbf{izlaz} \mid \mathbf{dodela} \end{aligned}$$

Nastavljamo sa određivanjem relacija prvenstva.

Zbog kraćeg pisanja uvešćemo sledeće oznake neterminalnih simbola:

$$\begin{aligned} \textit{Program} &- P \\ \textit{NizNaredbi} &- N \\ \textit{NizNaredbi}' &- NN' \\ \textit{Naredba} &- N \end{aligned}$$

Da bi se odredile relacije prvenstva između graničnih simbola i ostalih simbola gramatike uvodimo i pomoćnu smenu oblika

$$P' \rightarrow \# P \#.$$

U tabeli 7.4 je predstavljen postupak otkrivanja relacija prvenstva između svih simbola.

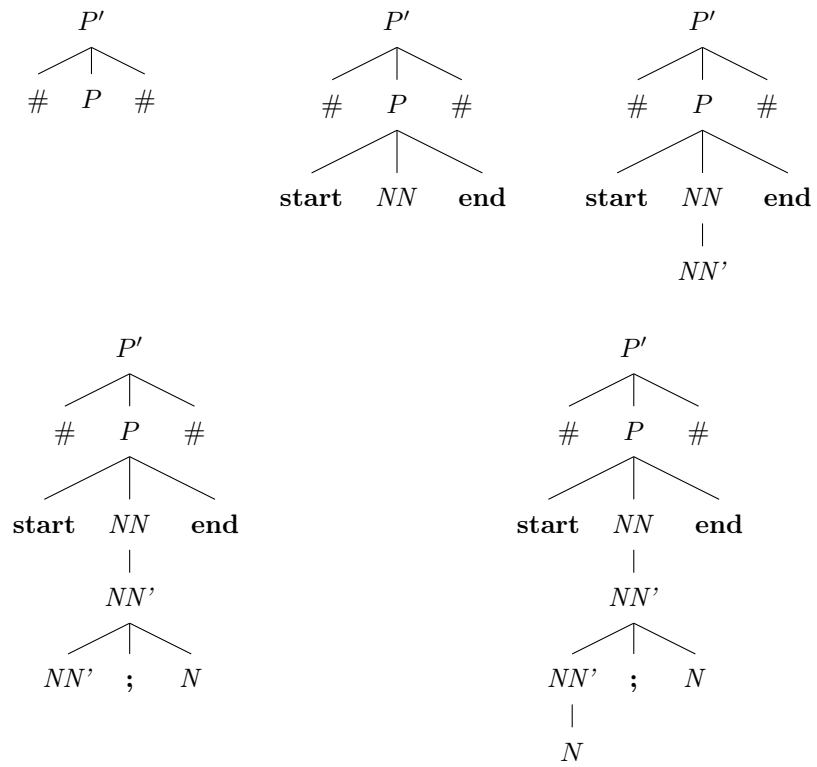
Kao rezultat ove analize dobija se tablica prvenstva dat u tabeli 7.5.

### 7.4.1 Postupak sintaksne analize

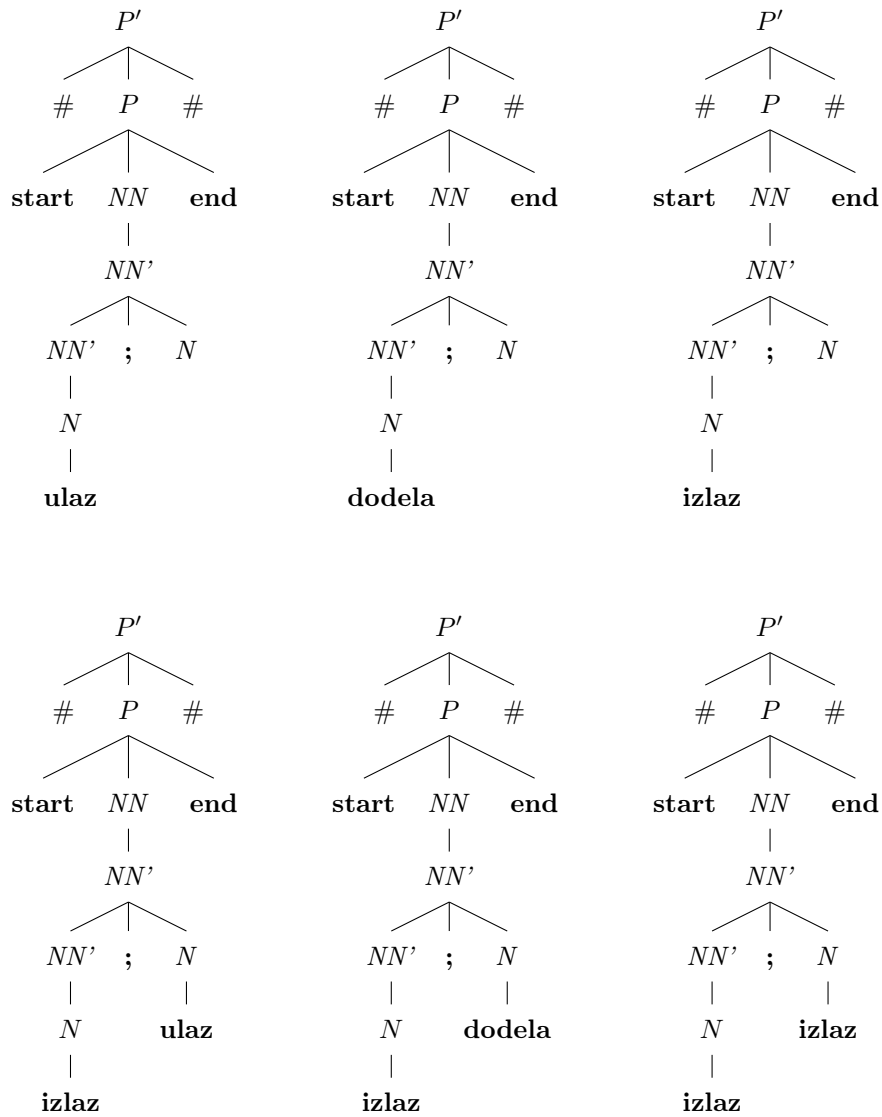
*Bottom-up* postupak sintaksne analize gramatika prvenstva je veoma sličan sintaksoj analizi operatorskih gramatika prvenstva. I u ovom slučaju prati se stanje ulaznog niza ( $\mathbf{t}$ ) i pomoćnog steka ( $\alpha$ ). Ulazni niz ( $\mathbf{t}$ ) dopuni se graničnim simbolom ( $\#$ ) sa desne strane. Isti granični simbol smešta se i u pomoćni stek ( $\alpha$ ). Zatim se, na osnovu relacije prvenstva koja važi između poslednjeg simbola u steku i sledećeg simbola u ulaznom nizu, određuje akcija koja će se

PRIMENJENA SMENA	IZVEDENI NIZ	STABLO	RELACIJE PRVENSTVA
$P' \rightarrow \#P\#$	$\#P\#$	slika 7.8	$\# \doteq P, \quad P \doteq \#$
$P \rightarrow \text{start } NN \text{ end}$	$\#\text{start } NN \text{ end } \#$		$\# < \text{start}, \quad \text{start} \doteq NN, \quad NN \doteq \text{eq}, \quad \text{end} > \#$
$NN \rightarrow NN'$	$\#\text{start } NN' \text{ end } \#$		$\text{start} < NN', \quad NN' > \text{end}$
$NN' \rightarrow NN'; N$	$\#\text{start } NN'; N \text{ end } \#$		$NN' \doteq ;, \quad ; \doteq N, \quad N > \text{end}$
$NN' \rightarrow N$	$\#\text{start } N; N \text{ end } \#$		$\text{start} < N, \quad N > ;$
$N \rightarrow \text{ulaz}$	$\#\text{start } \text{ulaz}; N \text{ end } \#$	slika 7.9	$\text{start} < \text{ulaz}, \quad \text{ulaz} > ;$
ili: $N \rightarrow \text{dodela}$	$\#\text{start } \text{dodela}; N \text{ end } \#$		$\text{start} < \text{dodela}, \quad \text{dodela} > ;$
ili: $N \rightarrow \text{izlaz}$	$\#\text{start } \text{izlaz}; N \text{ end } \#$		$\text{start} < \text{izlaz}, \quad \text{ulaz} > ;$
$N \rightarrow \text{ulaz}$	$\#\text{start } \text{izlaz}; \text{ulaz} \#$		$; < \text{ulaz}, \quad \text{ulaz} > \text{end}$
ili: $N \rightarrow \text{dodela}$	$\#\text{start } \text{izlaz}; \text{dodela} \#$		$; < \text{dodela}, \quad \text{dodela} > \text{end}$
ili: $N \rightarrow \text{izlaz}$	$\#\text{start } \text{izlaz}; \text{izlaz} \#$		$; < \text{izlaz}, \quad \text{izlaz} > \text{end}$

TABELA 7.4: Otkrivanje relacija prvenstva za gramatiku prvenstva iz primera 7.4



SLIKA 7.8: Sintaksna stabla uz tabelu 7.4 (prvi deo)



SLIKA 7.9: Sintaksna stabla uz tabelu 7.4 (drugi deo)

TABELA 7.5: Tabela prvenstva razmatrane gramatike prvenstva iz primera 7.4

	<i>P</i>	<i>NN</i>	<i>NN'</i>	<i>N</i>	<b>start</b>	<b>end</b>	<b>ulaz</b>	<b>izlaz</b>	<b>dodela</b>	<b>;</b>	<b>#</b>
<i>P</i>											$\dot{=}$
<i>NN</i>						$\dot{=}$					
<i>NN'</i>						$>$				$\dot{=}$	
<i>N</i>		$\dot{=}$	$<$	$<$			$<$	$<$	$<$		
<b>start</b>											$>$
<b>ulaz</b>						$>$				$>$	
<b>izlaz</b>						$>$				$>$	
<b>dodela</b>						$>$				$>$	
<b>;</b>				$\dot{=}$			$<$	$<$	$<$		
<b>#</b>	$\dot{=}$				$<$						

izvršiti:

1. Ako važi relacija  $<$  ili relacija  $\dot{=}$ , tekući simbol iz ulaznog niza smestiti u stek  $\alpha$  i preći na analizu sledećeg simbola.
2. Ako važi relacija  $>$  treba ići dublje u stek i tražiti prvi simbol u steku koji je  $<$  od svog sledbenika i proveriti da li fraza sa vrha steka (sadržaj magacina od uočenog simbola do vrha) predstavlja desnu stranu neke smene. Ako jeste, umesto cele fraze u stek treba ubaciti simbol sa leve strane te smene (tj. izvršiti redukciju), u suprotnom, u zapisu postoji greška i analizu treba prekinuti. Kada se redukcija uspešno izvede treba ispitati da li je sledeći ulazni simbol  $\#$ , a sadržaj steka  $\#S$  (gde je  $S$  startni simbol gramatike). Ako su ti uslovi zadovoljeni, znači da je ceo ulazni niz redukovan na startni simbol gramatike, tj. niz je prepoznat.
3. Ako ne postoji ni jedna od navedenih relacija znači da u zapisu postoji greška i dalju analizu treba prekinuti.

Kompletan *bottom-up* sintaksni analizator gramatika prvenstva dat je u vidu pseudokoda kao algoritam 2

#### Primer 7.5 Postupak sintaksne analize gramatike prvenstva

Za gramatiku prvenstva razmatranu u Primeru 7.4 izvršićemo sintaksnu analizu niza

**start ulaz ; dodela ; izlaz end.**



```

 $t \leftarrow t + \#;$ 
 $\alpha_0 \leftarrow \#;$ 
 $SP \leftarrow 0;$  /* SP je pokazivač na vrh magacina  $\alpha$  */
 $prepoznat \leftarrow 0;$ 
 $greška \leftarrow 0;$ 
 $next \leftarrow nextlex(t);$  /* poziv leksičkog analizatora */
repeat
  if  $\alpha_{SP} < next \vee \alpha_{SP} = next$  then
     $\alpha_{SP} \leftarrow next;$ 
     $SP \leftarrow SP - 1;$ 
     $next \leftarrow nextlex(t);$ 
  end
  else if  $\alpha_i > next$  then
     $i \leftarrow SP;$ 
    repeat
       $temp \leftarrow \alpha_i;$ 
       $i \leftarrow i - 1;$ 
    until  $\alpha_i < temp;$ 
    if postoji smena oblika  $\alpha_{i+1}, \dots, \alpha_{SP}$  then
       $SP \leftarrow i + 1;$ 
       $\alpha_{SP} \leftarrow P;$ 
      if  $SP = 2 \wedge \alpha_{SP} = s \wedge next = \#$  then
         $prepoznat \leftarrow 1$ 
      end
    end
    else
       $greška \leftarrow 1$ 
    end
  end
  else
     $greška \leftarrow 1$ 
  end
until  $prepoznat \wedge greška;$ 
return  $prepoznat;$ 

```

**Algorithm 2:** Pseudokod algoritma za sintaksnu analizu gramatika prvenstva.

U postupku analize datog niza koristiće se tablica prvenstva predstavljena u tabeli 7.6.

TABELA 7.6: Analiza izraza za gramatika iz primera 7.5. U poslednjoj koloni upisana je relacija prvenstva između simbola u vrhu magacina i simbola na ulazu. ✓ u poslednjem redu iznačava da je analiza uspešno završena.

RADNI MAGACIN ( $\alpha$ )	next	RELACIJA PRVENSTVA
#	start	<
# start	ulaz	<
# start ulaz	;	>
# start N	;	>
# start NN'	;	=
# start NN' ;	dodela	<
# start NN' ; dodela	;	>
# start NN' ; N	;	>
# start NN'	;	=
# start NN' ;	izlaz	<
# start NN' ; izlaz	end	>
# start NN' ; N	end	>
# start NN'	end	>
# start NN	end	=
# start NN end	#	>
# P	#	✓

Isto kao kod operatorskih gramatika prvenstva, i u slučaju gramatika prvenstva umesto tablice prioriteta mogu se definisati funkcije prvenstva. Ovde to ima još više smisla zato što ima mnogo više simbola između kojih se posmatraju relacije tako da su tablice većih dimenzija ali retko popunjene.

## 7.5 Pitanja

1. Dati definiciju operatorskih gramatika.
2. Definisati osnovne relacije prvenstva kod operatorskih gramatika prvenstva.

3. Dati definiciju operatorske gramatike prvenstva.
4. Šta je to operatorska tablica prvenstva? Dati primer.
5. Šta su to funkcije prvenstva i kako se određuju?
6. Opisati algoritam za sintaksnu analizu operatorskih gramatika prvenstva.
7. Dati definiciju relacija prvenstva kod gramatika prvenstva.
8. Dati definiciju gramatika prvenstva.
9. Šta je to tablica prvenstva i kako se popunjava.
10. Opisati algoritam za sintaksnu analizu gramatika prvenstva.

## 7.6 Zadaci

1. Proveriti da li je gramatika iz primera 7.1 gramatika prvenstva. Ako nije, dodefinisati je tako da zadovoljava uslove gramatike prvenstva i kreirati njenu tablicu prvenstva.
2. Kreirati operatorsku tablicu prvenstva gramatike koja je zadata sledećim skupom smena

$$\begin{aligned} S &\rightarrow S + I \mid I \\ I &\rightarrow ( S ) \mid f ( S ) \mid id \end{aligned}$$

Korišćenjem kreirane tabele proveriti da li niz

$$\# id + f ( id + f ( id ) ) \#$$

pripada jeziku koji je definisan ovom gramatikom.

3. Formirati operatorsku tablicu prvenstva za gramatiku **G** zadatu sledećim skupom smena:

$$\begin{aligned} S &\rightarrow P \\ P &\rightarrow R \textbf{ or } Q \mid Q \\ Q &\rightarrow T \\ T &\rightarrow T \textbf{ and } R \mid R \\ R &\rightarrow V \mid \textbf{ not } V \\ V &\rightarrow ( S ) \mid \textbf{ true } \mid \textbf{ false}. \end{aligned}$$

4. Formirati tablicu prvenstva za gramatiku **G** iz prethodnog zadatka.
5. Kreirati operatorsku tablicu prvenstva za gramatiku zadatu skupom smena:

$$Ulaz \rightarrow \textbf{ READ } ( NizPromenljivih )$$

$$NizPromenljivih \rightarrow NizPromenljivih, id \mid id.$$

Korišćenjem kreirane tablice proveriti da li izraz

$$\# \text{ READ } (id, id, ) \#$$

pripada jeziku date gramatike.

6. Gramatika **G** za definisanje tipa **enum** u programskom jeziku C zadata je sledećim skupom smena:

$$\begin{aligned} EnumType &\rightarrow \text{enum id } \{ ConstantList \} ; \\ ConstantList &\rightarrow ConstantList, ConstantDefinition \\ ConstantList &\rightarrow ConstantDefinition \\ ConstantDefinition &\rightarrow id \\ ConstantDefinition &\rightarrow id = \text{const} \end{aligned}$$

Transformisati datu gramatiku u gramatiku prvenstva, kreirati tablicu prvenstava i proveriti da li zapis

$$\text{enum id } \{ id, id = \text{const} \} ;$$

pripada jeziku ove gramatike.

7. Operatorska gramatika prvenstva zadata je sledećim skupom smena.

$$\begin{aligned} S &\rightarrow S + I \mid I \\ I &\rightarrow (S) \mid a(S) \mid a \end{aligned}$$

- (a) Formirati operatorsku tablicu prvenstva zadate gramatike.
- (b) Proveriti da li niz  $\# a(a + a) \#$  pripada jeziku date gramatike.

## Glava 8

# LR analizatori

U ovom poglavlju biće reči o karakteristikama i projektovanju LR analizatora koji se najčešće sreću u komercijalnim rešenjima kompilatora. Radi se o efikasnim *bottom-up* analizatorima. U opštem slučaju to mogu da budu LR( $k$ ) analizatori gde ova skraćenica ukazuje na njihove opšte karakteristike:

- L – (*left*) ulazni niz se analizira sleva udesno;
- R – (*right*) dobijaju se pravila koja odgovaraju desnom izvođenju;
- $k$  – broj slova na osnovu kojih se vrši predikcija (ako  $k$  nije navedeno, podrazumeva se jedno slovo).

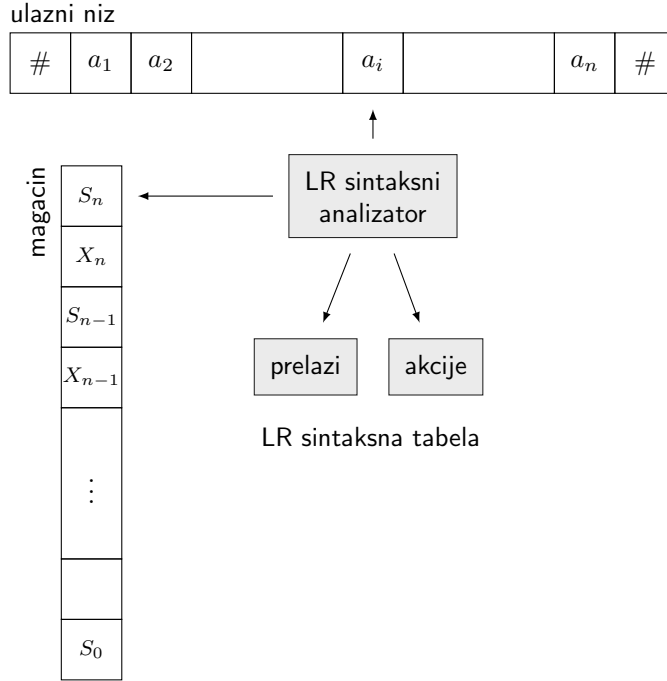
LR analizatorima mogu da se prepoznaju sve programske konstrukcije koje se mogu opisati beskonteksnim gramatikama. To je najopštiji metod za *bottom-up* analizu bez povratnih petlji koji se lako implementira. Klasa gramatika koje se prepoznaju LR analizatorima je nadskup klase gramatika kod kojih je moguća LL analiza. Takođe, omogućavaju najbrže moguće otkrivanje grešaka, onoliko brzo koliko je to uopšte moguće kod analizatora koji prolaze kroz ulazni niz sleva udesno.

### 8.1 Opis analizatora

Šema LR analizatora prikazana je na slici 8.1. U suštini to je automat koji kao radnu memoriju koristi magacin.

Simbol na vrhu radnog magacina je trenutno stanje LR sintaksnog analizatora i on sumira informacije smeštene u stek pre njega.

Postupak sintaksne analize je upravljan LR sintaksnom tabelom. Ona se sastoji od: akcija i prelaza. Ova sintaksna tabela ima onoliko vrsta koliko je mogućih stanja sintaksnog analizatora. U delu za akcije postoji onoliko kolona koliko



SLIKA 8.1: LR sintaksni analizator

je terminalnih simbola gramatike (uključujući i granični simbol), a u delu za prelaze broj kolona jednak je broju neterminalnih simbola gramatike. U delu za akcije definisani su prelazi koji će se izvršiti zavisno od stanja koje se nalazi na vrhu radnog magacina i tekućeg simbola u ulaznom nizu.

Akcija može biti:

- ***sp*** – (*shift p*) – znači da u magacin treba smestiti tekući simbol iz ulaznog niza i naredno stanje ( $p$ ) i preći na analizu sledećeg simbola iz ulaznog niza;
- ***rk*** – (*reduce k*) – znači da treba izvršiti redukciju primenom smene ( $k$ ). Ako je, na primer, smena ( $k$ )  $A \rightarrow \beta$ , iz magacina treba izbaciti  $2 \cdot \text{length}(\beta)$  elemenata, u magacin smestiti neterminalni simbol  $A$  (simbol sa leve strane smene ( $k$ )), a novo stanje pročitati iz dela *prelazi* u sintaksoj tabeli, kao prelaz iz prethodnog stanja u magacinu pod dejstvom ubačenog neterminalnog simbola ( $A$ ).
- ***acc*** (*accept*) – znači da je niz prepoznat i dalju analizu treba prekinuti;
- ***err*** (*error*) – znači da u ulaznom nizu postoji greška i treba prekinuti dalju analizu.

### 8.1.1 *Shift-reduce* algoritam analize

LR analizatori su poznati i kao Shift-reduce analizatori, što znači da se kroz ulazni niz prolazi sleva udesno i nastoji se da se izvrši redukcija (*reduce*), ako redukcija nije moguća prelazi se na sledeći znak u nizu (*shift*). Za razliku od osnovnog algoritma za *bottom-up* analizu koji je praktično *brute-force* algoritam, kod ovih analizatora se analizom smena gramatike generiše LR sintaksna tabela koja određuje promenu stanja u automatu u zavisnosti od tekućeg ulaznog simbola i nema povratnih koraka.

Postaviti ulazni pokazivač *ip* na početak niza **w** # koji se analizira;

```
while true do
  Neka je q znak u vrhu magacina;
  Neka je a znak na koji pokazuje ulazni pokazivač ip;
  if action(q, a) = shift then
    Ubaciti u magacin a, a zatim i p;
    Pomeriti ulazni pokazivač ip za jedno mesto udesno;
  end
  else if action(s, a) = reduce k, pri čemu je k-ta smena gramatike  $A \rightarrow \beta$ 
    then
      Izbaciti  $2 \cdot |\beta|$  simbola iz steka;
      Ubaciti A u stek;
      Ubaciti u stek oznaku stanja koja se dobija na osnovu goto(s', A), gde
        je s' oznaka stanja u koje smo se vratili posle redukcije;
    end
  else if action(s, #) = accept then return;
  else error;
end
```

**Algorithm 3:** LR Sintaksni analizator

Ako konfiguraciju automata označimo sa

$$(s_0, X_1, s_1, X_2, s_2, \dots, X_m, s_m, \quad a_i, a_{i+1}, \dots, a_n, \#),$$

gde je  $s_0, X_1, s_1, X_2, s_2, \dots, X_m, s_m$  niz u magacinu, pri čemu je  $s_m$  oznaka stanja koje je u vrhu magacina,  $a_i, a_{i+1}, \dots, a_n, \#$  – ulazni niz i  $a_i$  – slovo koje se u datom trenutku prepoznaje, preslikavanja u automatu se mogu predstaviti i na sledeći način.

Ako je action( $s_m, a_i$ ) = **reduce** *k* i *k*-ta smena  $A \rightarrow \beta$ ,  
 onda  $(s_0, X_1, s_1, X_2, s_2, \dots, X_{m-r}, s_{m-r}, A, s, \quad a_i, a_{i+1}, \dots, a_n, \#)$   
 $s = \text{goto}(s_{m-r}, A), \quad r = |\beta|$

Ako je action( $s_m, a_i$ ) = **shift** *p*,  
 onda  $(s_0, X_1, s_1, X_2, s_2, \dots, X_m, s_m, a_i, p, \quad a_{i+1}, \dots, a_n, \#)$

**Primer 8.1** LR alanizator

Gramatika  $\mathbf{G}$  je definisana sledećim skupom pravila.

$$E \rightarrow E + T \quad (1)$$

$$E \rightarrow T \quad (2)$$

$$E \rightarrow E * F \quad (3)$$

$$T \rightarrow F \quad (4)$$

$$F \rightarrow (F) \quad (5)$$

$$F \rightarrow id \quad (6)$$

LR sintaksni analizator je određen sledećom LR sintaksnom tabelom.

	id	+	*	(	)	#	$E$	$T$	$F$
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Postupak LR analize za reč  $\# id * id + id \#$  prikazan je u sledećoj tabeli.



NIZ U MAGACINU	ULAZNI NIZ	AKCIJA AUTOMATA
[0]	$id * id + id \#$	shift 5
0 $id$ [5]	$* id + id \#$	reduce( $F \rightarrow id$ )
0 $F$ [3]	$* id + id \#$	reduce( $T \rightarrow F$ )
0 $T$ [2]	$* id + id \#$	shift 7
0 $T$ 2 * [7]	$id + id \#$	shift 5
0 $T$ 2 * 7 $id$ [5]	$+ id \#$	reduce( $F \rightarrow id$ )
0 $T$ 2 * 7 $F$ [10]	$+ id \#$	reduce( $T \rightarrow T * F$ )
0 $T$ [2]	$+ id \#$	reduce( $E \rightarrow T$ )
0 $E$ [1]	$+ id \#$	shift 6
0 $E$ 1 + [6]	$id \#$	shift 5
0 $E$ 1 + 6 $id$ [5]	$\#$	reduce( $F \rightarrow id$ )
0 $E$ 1 + 6 $F$ [3]	$\#$	reduce( $T \rightarrow F$ )
0 $E$ 1 + 6 $T$ [9]	$\#$	reduce( $E \rightarrow E + T$ )
0 $E$ [1]	$\#$	acc

Analiza reči je uspešno završena.

## 8.2 Realizacija LR analizatora

LR analizator se realizuje kao automat koji prepoznaje sve prefikse koji mogu da nastanu u procesu generisanja reči jezika koji je opisan gramatikom za koju se vrši analiza.

### 8.2.1 Vidljivi prefiksi

U rečeničnoj formi  $\varphi\beta t$ , gde je  $\beta$  fraza koja se generiše, vidljivi prefiksi niza  $\beta$  su svi prefiksi niza  $\varphi\beta$ , uključujući i sam niz  $\varphi\beta$ .

Za  $\varphi\beta = u_1 u_2 \dots u_r$ , prefiksi su oblika  $u_1 u_2 \dots u_i$  ( $1 \leq i \leq r$ ).

#### Primer 8.2 Vidljivi prefiksi

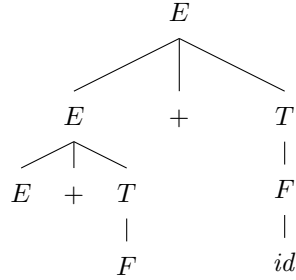
Za gramatiku  $\mathbf{G}$  iz primera 8.1, jedno od mogućih izvođenja je

$$\begin{aligned} E &\rightarrow E + T \\ &\rightarrow E + F \\ &\rightarrow E + id \\ &\rightarrow E + T + id \\ &\rightarrow E + F + id. \end{aligned}$$

Na osnovu ovog izvođenja, možemo da odredimo vidljive prefikse za frazu  $F$ :

$$E, \quad E + \quad i \quad E + F.$$

Izvođenje iz prethodnog primera predstavljeno je sledećim sintaksnim stablom.



Ako se stanje prikazano na sintaksnom stablu posmatra kao jedan trenutak u postupku *bottom-up* analize, može se uočiti da se svi vidljivi prefiksi posmatrane fraze u sintaksnom stablu nalaze levo od nje, dok se desno od nje nalazi niz terminalnih simbola koje tek treba otkriti.

### 8.2.2 $\text{LR}(k)$ gramatike

Posmatrajmo jedan korak u desnom izvođenju nekog niza:  $\varphi A \mathbf{t} \rightarrow \varphi \beta \mathbf{t}$ , gde je  $A$  prvi neterminal sa desne strane koji se preslikava u niz  $\beta$  primenom pravila  $A \rightarrow \beta$ . Očigledno je niz  $\mathbf{t}$  sastavljen od terminalnih simbola.

Gramatika  $\mathbf{G}$  je  $\text{LR}(k)$  gramatika ako se za bilo koji ulazni niz, u svakom koraku izvođenja fraza (podniz)  $\beta$  može detektovati na osnovu prefiksa  $\varphi \beta$  i skaniranjem najviše  $k$  znakova niza  $\mathbf{t}$ .

### 8.2.3 Tipovi LR analizatora

LR analizatori se realizuju kao automati koji prepoznaju vidljive prefikse u postupku generisanja reči jezika koji je opisan gramatikom. U zavisnosti od

toga koliko je kompleksan i kompletan postupak identifikacije vidljivih prefiksa razlikuje se više vrsta LR analizatora:

- SLR (*Simple LR*) – jednostavan za konstrukciju ali može da se dogodi da se za neke gramatike ne može sastaviti tablica sintaksne analize;
- kanonički LR – najmoćniji ali i najskuplji i najsloženiji postupak projektovanja;
- LALR (*Look-Ahead Left Right*) – između prva dva i po ceni i po performansama.

### 8.3 Realizacija SLR analizatora

Realizacija SLR analizatora se zasniva na definiciji tzv. LR(0) članova i primeni funkcije closure za zatvaranje LR(0) članova i funkcije goto za inicijalizaciju novog skupa LR(0) članova.

#### 8.3.1 LR(0) članovi

LR(0) članovi se generišu na osnovu pravila gramatike i svako pravilo gramatike daje nekoliko LR(0) članova u zavisnosti od dužine reči na desnoj strani pravila.

Izvođenje (na osnovu nekog pravila) sa tačkom u bilo kojoj poziciji u nizu se naziva LR(0) član.

##### **Primer 8.3** LR(0) članovi

Za gramatiku iz primera 8.1 na osnovu pravila mogu se generisati sledeći LR(0) članovi.

$$E \rightarrow \bullet E + T$$

$$E \rightarrow E \bullet + T$$

$$E \rightarrow E + \bullet T$$

$$E \rightarrow E + T \bullet$$

LR(0) članovi će se u postupku sinteze LR analizatora koristiti da se prate vidljivi prefiksi koji su uzeti u obzir. Naime, tačka praktično razdvaja vidljive prefikse od neprepoznatog dela reči.

### 8.3.2 Funkcija zatvaranja (closure)

Ako je  $\mathbf{I}$  skup LR(0) članova gramatike  $\mathbf{G}$ , tada je  $\text{closure}(\mathbf{I})$  skup LR(0) članova koji se dobijaju primenom sledećih pravila

1. Inicijalno svaki LR(0) član skupa  $\mathbf{I}$  uključuje se u skup  $\text{closure}(\mathbf{I})$ .
2. Ako  $A \rightarrow \alpha \bullet B \beta$  pripada skupu  $\text{closure}(\mathbf{I})$  i postoji pravilo gramatike oblika  $B \rightarrow \gamma$ , tada se skupu  $\text{closure}(\mathbf{I})$  pridodaje i član  $B \rightarrow \bullet \gamma$ , ako već ne postoji u tom skupu. Ovo pravilo se primenjuje sve dok je to moguće.

### 8.3.3 Funkcija iniciranja novih skupova LR(0) članova (goto)

Funkcija pomoću koje se inicira stvaranje novog skupa LR(0) članova definiše se kao  $\text{goto}(\mathbf{I}, X)$  gde je  $\mathbf{I}$  skup LR(0) članova, a  $X$  slovo azbuke.

Na osnovu svakog LR(0) člana oblika  $A \rightarrow \alpha \bullet X \beta$  iz skupa  $\mathbf{I}$  definiše se novi skup LR(0) članova  $\text{goto}(\mathbf{I}, X)$  kao zatvaranje svih članova  $A \rightarrow \alpha X \bullet \beta$ .

Ako je  $\mathbf{I}$  skup članova koji su značajni za prefiks  $\gamma$ , tada je  $\text{goto}(\mathbf{I}, X)$  skup svih LR(0) članova koji su značajni za prefiks  $\gamma X$ .

### 8.3.4 Postupak sinteze LR analizatora

Sinteza LR analizatora se svodi na generisanje LR sintaksne tabele, pri čemu se razlikuju sledeće tri faze.

1. Određivanje kanoničkog skupa LR članova koji obuhvata sve vidljive prefikse reči jezika koji je opisan gramatikom. Određivanje kanoničkog skupa kreće od LR(0) člana  $S' \rightarrow \bullet S$  koji se formira na osnovu fiktivne smene  $S' \rightarrow S$ , gde je  $S$  startni simbol gramatike. Nakon formiranja ovog početnog LR(0) člana primenjuju se funkcije closure i goto sve dok je to moguće. Rezultat ovog koraka je kanonički skup zatvaranja LR(0) članova  $K = \{\mathbf{I}_0, \mathbf{I}_1, \dots, \mathbf{I}_k\}$ .
2. Crtanje grafa prelaza konačnog automata za prepoznavanje vidljivih prefiksa pri čemu važe sledeća pravila:
  - (a) Svakom zatvaranju iz kanoničkog skupa pravila  $K = \{\mathbf{I}_0, \mathbf{I}_1, \dots, \mathbf{I}_k\}$  dodeljuje se jedno stanje u grafu automata.
  - (b) Potezi u grafu određeni su goto funkcijama iz kanoničkog skupa LR(0) pri čemu, ako je  $\text{goto}(\mathbf{I}_i, X) = \mathbf{I}_j$ , onda postoji odlazni poteg iz stanja  $\mathbf{I}_i$  do stanja  $\mathbf{I}_j$  za slovo  $X$ , gde je  $X \in \mathbf{V}$ .
  - (c) Sva stanja automata koja odgovaraju zatvaranjima LR(0) članova kojima pripadaju članovi oblika  $A \rightarrow \alpha \bullet$  su završna stanja (stanja redukcije za pravila  $A \rightarrow \alpha \bullet$ ).

- (d) Stanje kome pripada LR(0) član  $S' \rightarrow \bullet S$ , nastao na osnovu fiktivne smene je početno stanje automata.
3. Popunjavanje LR sintaksne tabele koje se vrši primenom sledećih pravila:
- (a) Ako  $[A \rightarrow \alpha \bullet a \beta] \in \mathbf{I}_i$  i  $\text{goto}(\mathbf{I}_i, a) = \mathbf{I}_j$ , tada je  $\text{action}(i, a) = \text{shift } j$ ,  $a \in \mathbf{V}_t$ .
- (b) Ako  $[A \rightarrow \alpha \bullet] \in \mathbf{I}_i$ , tada je  $\text{action}(i, a) = \text{reduce}(A \rightarrow \alpha)$  za  $a \in \text{Follow}(A)$ ,  $A \neq S'$ .
- (c) Ako  $[S' \rightarrow S \bullet] \in \mathbf{I}_i$ , tada je  $\text{action}(i, \#) = \text{accept}$ .

#### Primer 8.4 Sinteza LR analizatora

Izvršićemo sintezu LR analizatora za gramatiku  $\mathbf{G}$ , definisanu sledećim pravilima.

$$E \rightarrow E + T \quad (1)$$

$$E \rightarrow T \quad (2)$$

$$T \rightarrow T * F \quad (3)$$

$$T \rightarrow F \quad (4)$$

$$F \rightarrow (E) \quad (5)$$

$$F \rightarrow id \quad (6)$$

Startni simbol ove gramatike je  $E$  zbog čega skupu smena dodajemo i fiktivnu smenu  $E' \rightarrow E$ .

Na osnovu ove fiktivne smene formiramo inicijalni LR(0) član  $E' \rightarrow \bullet E$  od kojeg počinjemo formiranje kanoničke kolekcije LR(0) članova.

Uključujemo inicijalni član  $E' \rightarrow \bullet E$  u skup  $\mathbf{I}_0$  i vršimo zatvaranje ovog skupa primenom funkcije closure. Kako se tačka nalazi ispred neterminalnog simbola  $E$ , skupu se pridružuju svi inicijalni LR(0) članovi koji nastaju od pravila za neterminal  $E$ , a to su  $E \rightarrow \bullet E + T$  i  $E \rightarrow \bullet T$ .

Zatvaranje nastavljamo dalje. Kako skupu  $\mathbf{I}_0$  sada pripada član kod kojeg se tačka nalazi ispred neterminala  $T$ , skupu treba pridružiti inicijalne članove koji nastaju na osnovu pravila za neterminal  $T$ , odnosno  $T \rightarrow \bullet T * F$  i  $T \rightarrow \bullet F$ .

Sada skupu pripada član  $T \rightarrow \bullet F$ , kod kojeg je tačka ispred neterminala  $F$ , tako da skupu treba priključiti i članove koji nastaju na osnovu pravila za  $F$ , odnosno  $F \rightarrow \bullet (E)$  i  $F \rightarrow \bullet id$ .

Zatvaranje skupa  $\mathbf{I}_0$  je završeno i kao rezultat se dobija:

$$\begin{aligned} E' &\rightarrow \bullet E \\ E &\rightarrow \bullet E + T \\ E &\rightarrow \bullet T \\ T &\rightarrow \bullet T * F \\ T &\rightarrow \bullet F \\ F &\rightarrow \bullet (E) \\ F &\rightarrow \bullet id \end{aligned}$$

Na osnovu LR(0) članova iz skupa  $\mathbf{I}_0$ , primenom funkcije goto formiraju se novi skupovi. Tako na osnovu toga što skupu  $\mathbf{I}_0$  pripadaju članovi  $E' \rightarrow \bullet E$  i  $E \rightarrow \bullet E + T$ , kod kojih se tačka nalazi ispred neterminala  $E$ , moguće je primeniti funkciju  $\text{goto}(\mathbf{I}_0, E)$ , koja kao rezultat daje skup  $\mathbf{I}_1$  kome inicijalno priključujemo članove  $E' \rightarrow E \bullet$  i  $E \rightarrow E \bullet + T$ . Dalje bi trebalo izvršiti zatvaranje skupa  $\mathbf{I}_1$ . Međutim, on je zatvoren sa ova dva člana zato što se u jednom članu tačka nalazi na kraju, a u drugom je tačka ispred terminalnog simbola, a ne ispred neterminalnog.

Znači, dobili smo:

$$\begin{aligned} \mathbf{I}_1 &= \text{goto}(\mathbf{I}_0, E) \\ E' &\rightarrow E \bullet \\ E &\rightarrow E \bullet + T \end{aligned}$$

Na sličan način, primenom goto funkcije nad skupom  $\mathbf{I}_0$ , nastaju i sledeći skupovi:

$$\begin{aligned} \mathbf{I}_2 &= \text{goto}(\mathbf{I}_0, T) \\ E &\rightarrow T \bullet \\ T &\rightarrow T \bullet * F \end{aligned}$$

$$\begin{aligned} \mathbf{I}_3 &= \text{goto}(\mathbf{I}_0, F) \\ T &\rightarrow F \bullet \end{aligned}$$

$$\begin{aligned} \mathbf{I}_4 &= \text{goto}(\mathbf{I}_0, () \\ F &\rightarrow (\bullet E) \\ E &\rightarrow \bullet E + T \\ E &\rightarrow \bullet T \\ T &\rightarrow \bullet T * F \end{aligned}$$

$$\begin{aligned} T &\rightarrow \bullet F \\ F &\rightarrow \bullet(E) \\ F &\rightarrow \bullet id \end{aligned}$$

$$\begin{aligned} \mathbf{I}_5 &= \text{goto}(\mathbf{I}_0, id) \\ F &\rightarrow id\bullet \end{aligned}$$

Nakon što smo iscrpili mogućnosti primene funkcije goto nad skupom  $\mathbf{I}_0$ , nastavljamo sa primenom ove funkcije nad novonastalim skupovima. Uz to paralelno vršimo i zatvaranje svakog novonastalog skupa.

Kao rezultat primene funkcije goto nad skupom  $\mathbf{I}_1$  nastaje:

$$\begin{aligned} \mathbf{I}_6 &= \text{goto}(\mathbf{I}_1, +) \\ E &\rightarrow E + \bullet T \\ T &\rightarrow \bullet * F \\ T &\rightarrow \bullet F \\ F &\rightarrow \bullet(E) \\ F &\rightarrow \bullet id \end{aligned}$$

Funkcijom goto nad skupom  $\mathbf{I}_2$  dobija se

$$\begin{aligned} \mathbf{I}_7 &= \text{goto}(\mathbf{I}_2, *) \\ T &\rightarrow T * \bullet F \\ F &\rightarrow \bullet(E) \\ F &\rightarrow \bullet id \end{aligned}$$

Kako skup  $\mathbf{I}_3$  sadrži samo jedan član kod kojeg je tačka na kraju, nad njim se ne može primeniti funkcija goto i formirati novi skup.

Primenom goto funkcije nad skupom  $\mathbf{I}_4$ , dobija se jedan novi skup

$$\begin{aligned} \mathbf{I}_8 &= \text{goto}(\mathbf{I}_4, E) \\ F &\rightarrow (E\bullet) \\ E &\rightarrow E \bullet + T \end{aligned}$$

Međutim premena goto funkcije nad skupom  $\mathbf{I}_4$  gneriše i neke skupove koji već postoje, pa zato te skupove ne uzimamo kao nove. Naime, važi sledeće.

$$\text{goto}(\mathbf{I}_4, T) = \mathbf{I}_2$$

$$\text{goto}(\mathbf{I}_4, F) = \mathbf{I}_3$$

$$\text{goto}(\mathbf{I}_4, () = \mathbf{I}_4$$

$$\text{goto}(\mathbf{I}_4, id) = \mathbf{I}_5$$

Primena goto funkcije nad skupom  $\mathbf{I}_5$  takođe nije moguća zato što, kao i skup  $\mathbf{I}_3$ , sadrži samo jedan član sa tačkom na kraju.

Od skupa  $\mathbf{I}_6$  se generiše jedan novi član

$$\mathbf{I}_9 = \text{goto}(\mathbf{I}_6, T)$$

$$E \rightarrow E + T \bullet$$

$$T \rightarrow T \bullet * F$$

i nekoliko koji već postoje:

$$\text{goto}(\mathbf{I}_6, F) = \mathbf{I}_3$$

$$\text{goto}(\mathbf{I}_6, () = \mathbf{I}_4$$

$$\text{goto}(\mathbf{I}_6, id) = \mathbf{I}_5$$

Ovaj postupak se nastavlja sve dok je moguće. Skup  $\mathbf{I}_7$  daje

$$\mathbf{I}_{10} = \text{goto}(\mathbf{I}_7, F)$$

$$T \rightarrow T * F \bullet$$

kao i

$$\text{goto}(\mathbf{I}_7, () = \mathbf{I}_4,$$

$$\text{goto}(\mathbf{I}_7, id) = \mathbf{I}_5.$$

Skup  $\mathbf{I}_8$  daje

$$\mathbf{I}_{11} = \text{goto}(\mathbf{I}_8, )$$

$$E \rightarrow (E) \bullet$$

kao i

$$\text{goto}(\mathbf{I}_8, +) = \mathbf{I}_6.$$



Skup  $I_9$  daje

$$\text{goto}(I_9, *) = I_7$$

Dalja primena funkcije goto nije moguća. Ovim postupkom je određena kanonička kolekcija LR(0) članova

$$C = \{I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}, I_{11}\},$$

na osnovu koje se generiše LR sintaksna tabela i graf automata koji prepoznaje sve vidljive prefikse gramatike.

Kao rezultat se dobija LR sintaksna tabela predstavljena uz primer 8.1, dok je graf automata dat na slici 8.2.

## 8.4 Rešeni zadaci

U ovom odeljku biće data rešenja još nekih zadataka ovog tipa.

**Zadatak 3** Za gramatiku zadatu skupom smena

$$\begin{aligned} \text{Program} &\rightarrow \text{start NizNaredbi end} \\ \text{NizNaredbi} &\rightarrow \text{NizNaredbi ; Naredba} \mid \text{Naredba} \\ \text{Naredba} &\rightarrow \text{ulaz} \mid \text{izlaz} \mid \text{dodela} \end{aligned}$$

kreirati LR sintaksnu tabelu i, korišćenjem kreirane tabele, proveriti da li je program

**start ulaz ; dodela ; izlaz end**

ispravno napisan.

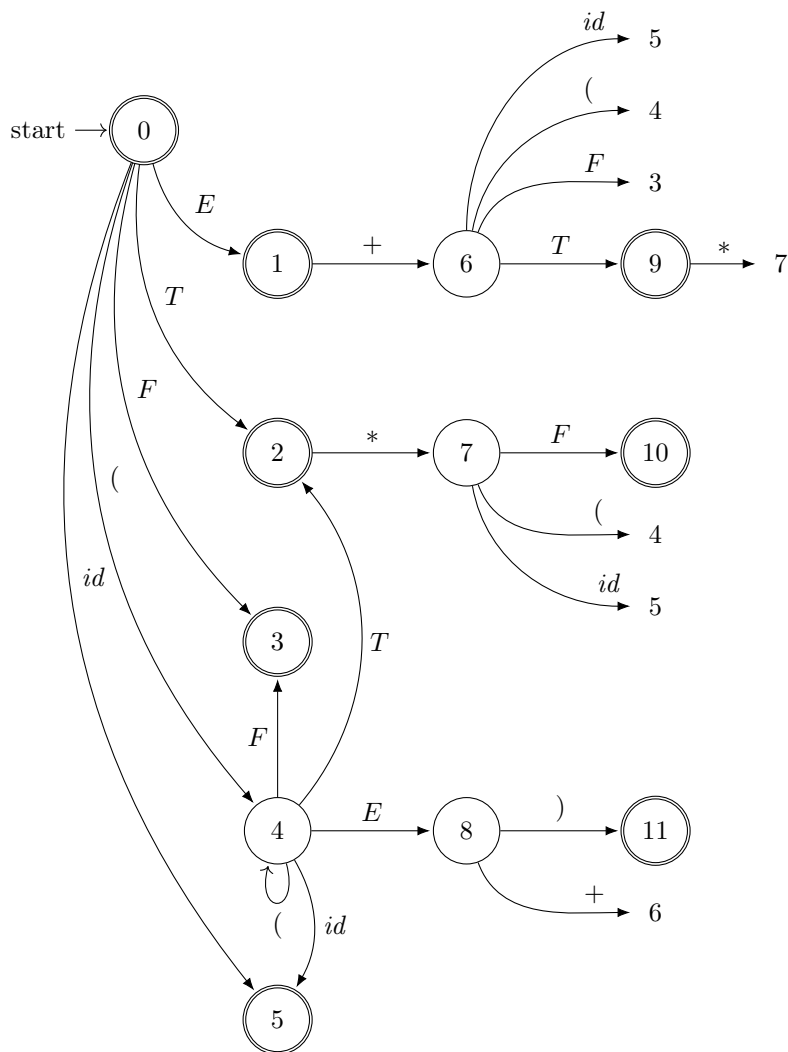
*Rešenje.* **Kreiranje kanoničkog skupa LR(0) članova.** Uvodimo smenu  $P' \rightarrow P$  i definišemo prvo zatvaranje skupa LR pavila:

$$I_0 : P' \rightarrow \bullet P.$$

Tačka koja razdvaja vidljivi prefiks od neprepoznatog dela pravila se nalazi ispred neterminalnog simbola što znači da u nastavku analize treba prepoznati neterminalni simbol. Neterminalni simboli se ne pojavljuju u ulaznom kodu pa prepoznavanje neterminalnog simbola podrazumeva, u stvari, prepoznavanje desne strane neke od smena na čijoj se levoj strani nalazi taj simbol. Zbog toga se ovom zatvaranju dodaje skup pravila izvedenih iz smena za preslikavanje tog neterminalnog simbola sa tačkom na početku. U ovom slučaju dodajemo član

$$P \rightarrow \bullet \text{start NN end}.$$

Kako smo rekli da tačka razdvaja prepoznati deo (vidljivi prefiks) od neprepoznatog, to znači da, kada se analizator nađe u posmatranom stanju, u nastavku



SLIKA 8.2: Graf automata LR analizaora iz primera 8.4

analize očekujemo pojavu simbola koji se nalaze iza tačaka u pravilima koja pripadaju tekućem zatvaranju. U konkretnom primeru, simboli koji se očekuju u stanju  $\mathbf{I}_0$  su  $P$  i **start**. To znači da treba definisati dva prelaza iz zatvaranja  $\mathbf{I}_0$  – kada se pojavi simbol  $P$  i kada se pojavi simbol **start**.

$$\mathbf{I}_1 = \text{goto}(\mathbf{I}_0, P)$$

Prepoznat je simbol  $P$ , vidljivi prefiks označen u pravilu  $(0, 1)$  se povećava – tačka se pomera iza prepoznatog simbola pa ovom zatvaranju pripada pravilo

$$P' \rightarrow P \bullet.$$

Pošto se tačka našla na kraju pravila, ovo stanje je redukciono stanje za smenu iz koje je pravilo izvedeno (tj. za smenu  $P' \rightarrow P$ ). Pošto je ovo pomoćna (fiktivna) smena, ova redukcija se ne vrši, već kad analizator dođe do ovog stanja, ukoliko je ceo ulazni kod analiziran, ulazni kod se prihvata (tj. korektan je).

$$\mathbf{I}_2 = \text{goto}(\mathbf{I}_0, \text{start} :$$

Prepoznat je simbol **start**, vidljivi prefiks se povećava – tačka se pomera iza prepoznatog simbola pa zatvaranju  $\mathbf{I}_2$  pripada pravilo

$$P \rightarrow \text{start} \bullet NN \text{end.}$$

Tačka se nalazi ispred neterminalnog simbola  $NN$ , pa ovom zatvaranju pripadaju i pravila

$$NN \rightarrow \bullet NN ; N$$

$$NN \rightarrow \bullet N$$

Pošto ovom zatvaranju pripada i član u kojem je tačka ispred neterminalnog simbola  $N$ , ovom zatvaranju treba dodati i članove nastale na osnovu smena koje na levoj strani imaju neterminal  $N$ .

$$N \rightarrow \bullet \text{ulaz}$$

$$N \rightarrow \bullet \text{izlaz}$$

$$N \rightarrow \bullet \text{dodela}$$

$$\mathbf{I}_3 = \text{goto}(\mathbf{I}_2, NN :$$

Skupu  $\mathbf{I}_2$  pripadaju dva člana u kojima je tačka ispred simbola  $NN$  pa će zatvaranju  $\mathbf{I}_3$  pripadati članovi

$$\mathbf{I}_3 = \text{goto}(\mathbf{I}_2, NN)$$

$$P \rightarrow \text{start} NN \bullet \text{end}$$

$$NN \rightarrow NN \bullet ; N$$

$$\mathbf{I}_4 = \text{goto}(\mathbf{I}_3, \text{end})$$

$$P \rightarrow \text{start } NN \text{ end } \bullet$$

Zatvaranju  $\mathbf{I}_4$  pripada član sa tačkom na kraju. Taj član je izveden iz prve smene gramatike. To znači da će stanje koje odgovara ovom zatvaranju biti redukciono stanje za smenu (1).

$$\mathbf{I}_5 = \text{goto}(\mathbf{I}_3, ;)$$

$$NN \rightarrow NN ; \bullet N$$

$$N \rightarrow \bullet \text{ulaz}$$

$$N \rightarrow \bullet \text{izlaz}$$

$$N \rightarrow \bullet \text{dodela}$$

$$\mathbf{I}_6 = \text{goto}(\mathbf{I}_5, N)$$

$$NN \rightarrow NN ; N \bullet$$

$$\mathbf{I}_7 = \text{goto}(\mathbf{I}_2, N)$$

$$NN \rightarrow N \bullet$$

$$\mathbf{I}_8 = \text{goto}(\mathbf{I}_2, \text{ulaz})$$

$$N \rightarrow \text{ulaz } \bullet$$

$$\mathbf{I}_8 = \text{goto}(\mathbf{I}_2, \text{izlaz})$$

$$N \rightarrow \text{izlaz } \bullet$$

$$\mathbf{I}_8 = \text{goto}(\mathbf{I}_2, \text{dodela})$$

$$N \rightarrow \text{dodela } \bullet$$

$$\text{goto}(\mathbf{I}_5, \text{ulaz}) = \mathbf{I}_8$$

$$\text{goto}(\mathbf{I}_5, \text{izlaz}) = \mathbf{I}_9$$

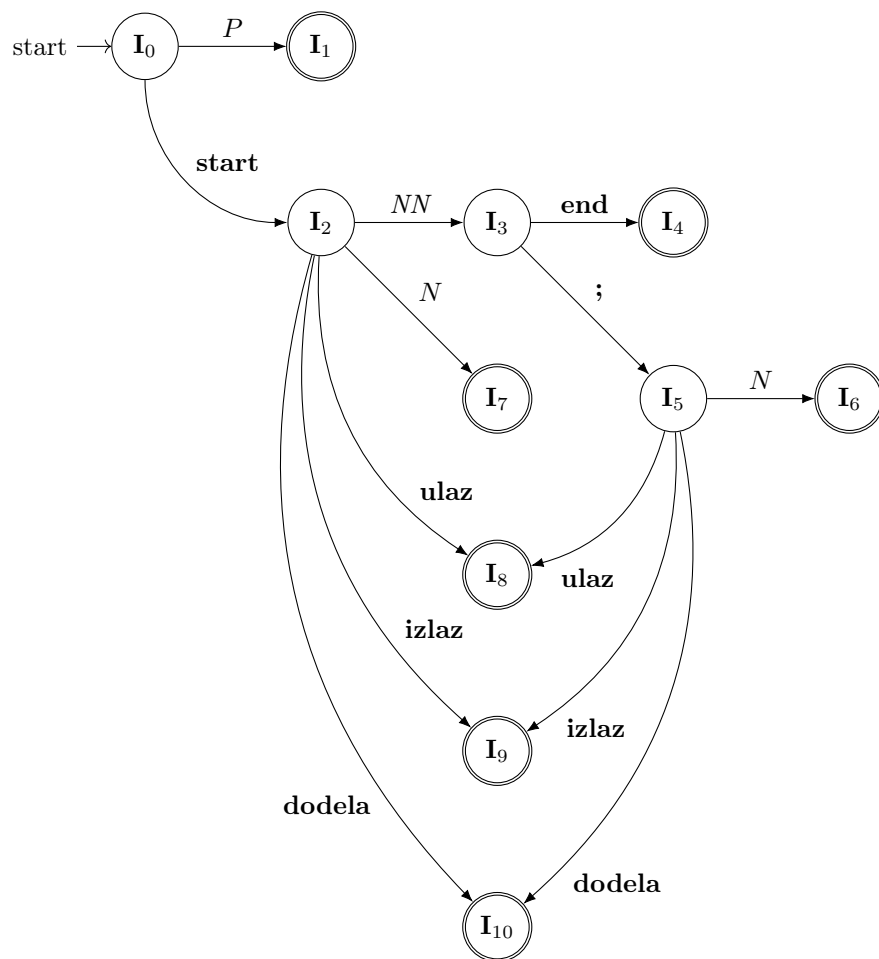
$$\text{goto}(\mathbf{I}_5, \text{dodela}) = \mathbf{I}_{10}$$

**Crtanje grafa prelaza konačnog automata za prepoznavanje vidljivih prefiksa.** Za svako zatvaranje skupa LR članova crtamo po jedan čvor u grafu. Potezi u grafu su određeni prelazima koje smo definisali prilikom kreiranja ovog skupa LR članova. Na primer, zatvaranje  $\mathbf{I}_1$  smo definisali kao prelaz iz stanja  $\mathbf{I}_0$  kad se pojavi simbol  $P$ , pa u grafu postoji grana između stanja  $\mathbf{I}_0$  i  $\mathbf{I}_1$  označena simbolom  $P$ . Redukciona stanja se obeležavaju kao završna stanja u automatu. Graf generisanog automata predstavljen je na slici 8.3.

**Popunjavanje LR sintaksne tabele.** *Upis shift akcija.* Svakom potezu koji je označen terminalnim simbolom odgovara po jedna shift akcija. Na primer, stanja  $\mathbf{I}_0$  i  $\mathbf{I}_2$  su povezana potegom koji je obeležen terminalnim simbolom **start**. To znači da će  $\text{akcija}(0, \text{start})$  biti s2. Stanja  $\mathbf{I}_3$  i  $\mathbf{I}_4$  su povezana potegom koji je obeležen terminalnim simbolom **end**, pa je  $\text{akcija}(3, \text{end}) = \text{s4}$ , itd.

*Popunjavanje prelaza u LR tabeli.* Svakom potezu koji je označen neterminalnim simbolom odgovara po jedan prelaz u LR sintaksoj tabeli. Na primer, stanja  $\mathbf{I}_0$  i  $\mathbf{I}_1$  su povezana potegom koji je obeležen neterminalnim simbolom  $P$ . To znači da je  $\text{prelaz}(0, P) = 1$ , itd.

*Upis reduce akcija.* reduce akcije se pišu u vrstama koje odgovaraju završnim stanjima automata. Na primer, završno stanje  $\mathbf{I}_4$  sadrži redukciono pravilo



SLIKA 8.3: Graf automata LR analizatora iz zadatka 3

$P \rightarrow \text{start } NN \text{ end}$  izvedeno iz smene (1).<sup>1</sup> To znači da se u stanju  $I_4$  vrši redukcija po smeni (1) kada se na ulazu pojavi bilo koji simbol koji pripada Follow funkciji za neterminalni simbol koji se nalazi na levoj strani smene, tj. za simbol  $P$ . Kako je  $\text{Follow}(P) = \{\#\}$ , dobija se  $\text{akcija}(4, \#) = r1$ .

Slično se definišu i ostale reduce akcije. Follow funkcije za ostale neterminalne simbole su

$$\text{Follow}(NN) = \{\text{end}, ;\}$$

$$\text{Follow}(N) = \text{Follow}(NN) = \{\text{end}, ;\}.$$

Jedino završno stanje u kojem se ne vrši redukcija je stanje  $I_1$ . Rečeno je već da se redukcija po novouvedenoj pomoćnoj smeni ne vrši, već ako je cela ulazna datoteka pročitana (tj. sledeći simbol je  $\#$ ), kod je kompletno ispravan pa je  $\text{akcija}(1, \#) = \text{acc}$ .

Generisana LR sintaksna tabela data je u tabeli 8.1.

TABELA 8.1: LR sintaksna tabela dobijena u zadatku 3

	start	end	;	ulaz	izlaz	dodela	#	$P$	$NN$	$N$
0	s2							1		
1							acc			
2				s8	s9	s10			3	7
3		s4	s5							
4							r1			
5				s8	s9	s10				6
6		r2	r2							
7		r3	r3							
8		r4	r4							
9		r5	r5							
10		r6	r6							

**Analiza ulaznog niza.** Postupak analize ulaznog niza

# start    ulaz ;    dodela ;    izlaz    end #

prikazan je u tabeli 8.2.

<sup>1</sup>Redukciono pravilo je pravilo koje odgovara članu sa tačkom na kraju.

NIZ U MAGACINU	next	AKCIJA AUTOMATA
<span>[0]</span>	<span>start</span>	s2
0 start <span>[2]</span>	<span>ulaz</span>	s8
0 start 2 ulaz <span>[8]</span>	<span>;</span>	r4 (smena 4: $N \rightarrow \text{ulaz}$ , $\text{prelaz}(2, N) = 7$ )
0 start 2 $N$ <span>[7]</span>	<span>;</span>	r3 (smena 3: $NN \rightarrow N$ , $\text{prelaz}(2, NN) = 3$ )
0 start 2 $NN$ <span>[3]</span>	<span>;</span>	s5
0 start 2 $NN$ ; <span>[5]</span>	<span>dodela</span>	s10
0 start 2 $NN$ 3; 5 <span>dodela</span> <span>[10]</span>	<span>;</span>	r6 (smena 6: $N \rightarrow \text{dodela}$ , $\text{prelaz}(5, N) = 6$ )
0 start 2 $NN$ 3; 5 $N$ <span>[6]</span>	<span>;</span>	r2 (smena 2: $NN \rightarrow NN$ ; $N$ , $\text{prelaz}(2, NN) = 3$ )
0 start 2 $NN$ <span>[3]</span>	<span>;</span>	s5
0 start 2 $NN$ 3; <span>[5]</span>	<span>izlaz</span>	s9
0 start 2 $NN$ 3; 5 <span>izlaz</span> <span>[9]</span>	<span>end</span>	r5 (smena 5: $N \rightarrow \text{izlaz}$ , $\text{prelaz}(5, N) = 6$ )
0 start 2 $NN$ 3; 5 $N$ <span>[6]</span>	<span>end</span>	r2 (smena 2: $NN \rightarrow NN$ ; $N$ , $\text{prelaz}(2, NN) = 3$ )
0 start 2 $NN$ <span>[3]</span>	<span>end</span>	s4
0 start 2 $NN$ 3 <span>end</span> <span>[4]</span>	<span>#</span>	r1 (smena 1: $P \rightarrow \text{start } NN \text{ end}$ , $\text{prelaz}(0, P) = 1$ )
0 $P$ <span>[1]</span>	<span>#</span>	acc – niz je prepoznat

TABELA 8.2: Postupak analize zadatkaog niza iz zadatka 3. Radi uštede prostora prikazan je samo simbol ulaznog niza koji je na redu za prepoznavanje, a ne i ostatak niza.

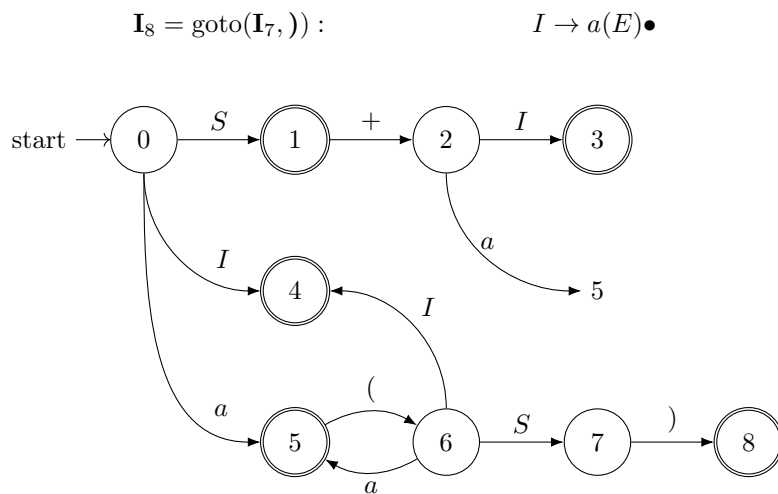
**Zadatak 4** Formirati LR sintaksnu tabelu gramatike koja je zadata skupom smena

$$\begin{aligned} E &\rightarrow E + I \mid I \\ I &\rightarrow a ( E ) \mid a \end{aligned}$$

*Rešenje.* Kanonički skup članova ove gramatike dat je u nastavku, graf prelaza konačnog automata za prepoznavanje vidljivih prefiksa na slici 8.4, a sama LR sintaksna tabela u tabli 8.3.

$\mathbf{I}_0 :$	$E' \rightarrow \bullet E$ $E \rightarrow \bullet E + I$ $E \rightarrow \bullet I$ $I \rightarrow \bullet a(E)$ $I \rightarrow \bullet a$
$\mathbf{I}_1 = \text{goto}(\mathbf{I}_0, E) :$	$E' \rightarrow E \bullet$ $E \rightarrow E \bullet + I$
$\mathbf{I}_2 = \text{goto}(\mathbf{I}_1, +) :$	$E \rightarrow E + \bullet I$ $I \rightarrow \bullet a(E)$ $I \rightarrow \bullet a$
$\mathbf{I}_3 = \text{goto}(\mathbf{I}_2, I) :$	$E \rightarrow E + I \bullet$
$\mathbf{I}_4 = \text{goto}(\mathbf{I}_0, I) :$	$E \rightarrow I \bullet$
$\mathbf{I}_5 = \text{goto}(\mathbf{I}_0, a) :$	$I \rightarrow a \bullet (E)$ $I \rightarrow a \bullet$
$\mathbf{I}_6 = \text{goto}(\mathbf{I}_5, ( ) :$	$I \rightarrow a(\bullet E)$ $E \rightarrow \bullet E + I$ $E \rightarrow \bullet I$ $I \rightarrow \bullet a(E)$ $I \rightarrow \bullet a$
$\mathbf{I}_7 = \text{goto}(\mathbf{I}_6, E) :$	$I \rightarrow a(E \bullet)$ $E \rightarrow E \bullet + I$





SLIKA 8.4: Graf automata za prepoznavanje vidljivih prefiksa uz zadatak 4

TABELA 8.3: LR sintaksna tabela dobijena u zadatku 4

	<i>a</i>	+	(	)	#	<i>S</i>	<i>I</i>
0	s5					1	4
1		s2			acc		
2	s5						3
3		r1		r1	r1		
4		r2		r2	r2		
5		r4	s6	r4	r4		
6	s5					7	4
7		s2		s8			
8		r3		r3	r3		

## 8.5 Pitanja

1. Objasniti strukturu LR analizatora.
2. Objasniti strukturu LR tablice sintaksne analize.
3. Objasniti postupak LR analize.

4. Kako se definišu vidljivi prefiksi?
5. Kako se definišu LR(0) članovi?
6. Definirati kanoničku kolekciju LR(0) članova?
7. Kako se kanonička kolekcija LR(0) članova preslikava u graf automata?
8. Kako se određuju završna stanja automata?
9. Kako je određeno početno stanje automata?
10. Kako je preslikava kanonička kolekcija LR(0) članova u LR sintaksnu tabelu?
11. Kako su određena shift polja u LR sintakсноj tabeli.
12. Kako su određena reduce polja u LR sintakсноj tabeli.
13. Kako je određen deo sa prelazima u LR sintakсноj tabeli.

## 8.6 Zadaci

1. Kreirati LR sintakсну tabelu za gramatiku zadatu skupom smena

$$\begin{aligned}
 CaseList &\rightarrow CaseList CaseItem \mid CaseItem \\
 CaseItem &\rightarrow \mathbf{case} \ CONST : Assignment \ \mathbf{break} ; \\
 Assignment &\rightarrow ID = Expression ; \\
 Expression &\rightarrow ID \mid CONST.
 \end{aligned}$$

Korišćenjem kreirane tabele proveriti da li jeziku koji je definisan ovom gramatikom pripada niz

$$\begin{aligned}
 &\mathbf{case} \ 1 : ID = CONST ; \\
 &\mathbf{case} \ 2 : ID = ID ; \mathbf{break} ;
 \end{aligned}$$

2. Kreirati LR sintakсну tabelu gramatike koja je definisana skupom smena

$$\begin{aligned}
 E &\rightarrow ( E ) T \mid \mathbf{const} \ T \\
 T &\rightarrow + E \mid * E \mid \varepsilon
 \end{aligned}$$

Korišćenjem kreirane tabele proveriti da li izraz

$$\# \mathbf{const} * (\mathbf{const} + \mathbf{const}) \#$$

pripada jeziku date gramatike.

3. Za gramatiku **G** zadatu skupom smena

$$\begin{aligned} \text{Ulaz} &\rightarrow \mathbf{read} ( \text{NizPromenljivih} ) \\ \text{NizPromenljivih} &\rightarrow \text{NizPromenljivih} , \mathbf{id} \mid \mathbf{id} \end{aligned}$$

kreirati odgovarajuću LR sintaksnu tabelu. Korišćenjem kreirane tabele proveriti da li niz

$$\mathbf{read} ( \mathbf{id} , \mathbf{id} , )$$

pripada jeziku date gramatike.

4. Kreirati LR sintaksnu tabelu gramatike koja je zadata sledećim skupom smena:

$$\begin{aligned} S &\rightarrow S + I \mid I \\ I &\rightarrow ( S ) \mid \mathbf{f} ( S ) \mid \mathbf{id} \end{aligned}$$

Korišćenjem kreirane tabele proveriti da li niz

$$\# \mathbf{id} + \mathbf{f} ( \mathbf{id} + * \mathbf{f} ( \mathbf{id} ) ) \#$$

pripada jeziku koji je definisan ovom gramatikom.

5. Kreirati LR sintaksnu tabelu za gramatiku definisanu skupom smena

$$E \rightarrow E + E \mid E - E \mid E + + \mid E - - \mid \mathbf{id}.$$

6. Kreirati LR sintaksnu tabelu za gramatiku zadatu sledećim skupom smena:

$$\begin{aligned} S &\rightarrow S A T \mid T \\ T &\rightarrow T M F \mid F \\ A &\rightarrow + \mid - \mid \mathbf{or} \\ M &\rightarrow * \mid / \mid \mathbf{mod} \mid \mathbf{div} \mid \mathbf{and} \\ F &\rightarrow \mathbf{id} \end{aligned}$$

7. Kreirati LR sintaksnu tabelu gramatike koja je zadata skupom smena

$$\begin{aligned} S &\rightarrow S + I \mid I \\ I &\rightarrow ( S ) \mid F ( S ) \mid \mathbf{id} \\ F &\rightarrow \mathbf{sin} \mid \mathbf{cos} \mid \mathbf{abs} \mid \mathbf{sqr} \end{aligned}$$

8. Kreirati LR sintaksnu tabelu za gramatiku zadatu skupom smena

$$\begin{aligned} \langle \text{block} \rangle &::= \mathbf{begin} \langle \text{decs} \rangle ; \langle \text{stmts} \rangle \mathbf{end} \\ \langle \text{decs} \rangle &::= d \mid \langle \text{decs} \rangle ; d \\ \langle \text{stmts} \rangle &::= s \mid \langle \text{stmts} \rangle ; s \end{aligned}$$

Korišćenjem kreirane sintaksne tabele proveriti da li niz

**begin  $d$  ;  $d$  ;  $s$  ;  $s$  end**

pripada jeziku koji je definisan datom gramatikom.

9. Za gramatiku zadatu skupom smena

$$E \rightarrow E \ O \ E \mid ( \ E \ ) \mid \mathbf{num}$$

$$O \rightarrow + \mid - \mid * \mid /$$

kreirati LR sintaksnu tabelu.

## Glava 9

# Tabele simbola

Tabele simbola su strukture podataka koje u programskim prevodiocima čuvaju informacije o simboličkim imenima koja se koriste u programu. Kako se simbolička imena mogu koristiti za imenovanje različitih tipove entiteta u programu (promenljivih, konstanti, funkcija, korisničkih tipova podataka i sl), jasno je da su informacije koje se pamte o simboličkim imenima veoma raznorodne.

Na primer,

- za promenljive je potrebno zapamtiti
  - ime,
  - tip (zbog provere slaganja tipova u izrazima u kojima učestvuje),
  - tačku u kodu u kojoj je definisana (zbog prijave greške ukoliko se koristi van oblasti njenog važenja),
  - tačku u kodu u kojoj je prvi put inicijalizovana (zbog prijave greške ukoliko se vrednost promenljive koristi pre njene inicijalizacije),
  - tačku u kodu u kojoj je poslednji put korišćena (zbog prijave upozorenja ukoliko u programu postoje promenljive koje se nigde ne koriste), itd;
- za funkcije je potrebno znati
  - ime (pri pozivu funkcije treba proveriti da li takva funkcija uopšte postoji),
  - tip (zbog provere slaganja tipova u izrazima u kojima se koristi),
  - broj i tipove atrumenata (zbog provere da li se lista stvarnih argumenata slaže sa listom fiktivnih argumenata), itd.

Već iz ovog kratkog primera može se zaključiti da se tabele simbola koriste u semantičkoj analizi u dve svrhe:

- za proveru tipova podataka koji učestvuju u izrazima (*type checking*) i
- za proveru opsega važenja imena (*scope checking*).

## 9.1 Predstavljanje simbola u tabeli simbola

Za sve navedene tipove entiteta koji se imenuju, zajednička karakteristike su ime i tip kojem pripadaju. Kako i informacije o tipu mogu biti vrlo složene (postoje elementarni tipovi podataka, polja, strukture, klase, itd) simboli se predstavljaju složenim tipovima podataka koje imaju strukturu grafa. U tim grafovima koriste se dva osnovna tipa čvorova:

- čvorovi kojima se predstavljaju tipovi (*type nodes* ili *structure nodes*) i
- čvorovi kojima se predstavljaju imenovani entiteti u programu (*object nodes*).

S obzirom na raznorodnost podataka koji se o entitetima i tipovima pamte u tabeli simbola, postoje dva osnovna pristupa za implementaciju ova dva tipa čvorova:

- *flat* pristup kod koga jedinstvena struktura podataka sadrži sve podatke koji se mogu pamtit o svim mogućim vrstama entiteta (odnosno tipova) i
- objektno-orijentisani pristup kod koga se u apstraktnoj klasi pamti ono što je zajedničko za sve vrste entiteta (tipova), a u konkretizovanim izvedenim klasama ono što je karakteristično za konkretne vrste entiteta/tipova.

## 9.2 Predstavljanje simbola korišćenjeme *flat* pristupa

Strukture podataka koje se mogu koristiti za predstavljanje tipova i imenovanih entiteta u hipotetičkom programskom jeziku korišćenjem *flat* pristupa implementacije simbola prikazane su u listingu 1. Pretpostavljeno je da jezik sadrži četiri osnovna tipa podataka: `int`, `char`, `float` i `double`, kao i složene tipove: polja i strukture i da entiteti koji imaju svoje simboličko ime mogu biti: promenljve, konstante, korisnički definisani tipovi podataka, članovi struktura i metode.

### Primer 9.1 Predstavljanje simbola *flat* metodom

Na slici 9.1 data je struktura podataka (graf) kojom je u tabeli simbola

## 9.2. PREDSTAVLJANJE SIMBOLA KORIŠĆENJEME FLAT PRISTUPA161

```
struct TypeNode {
    static const Int = 0, Char = 1, Float = 2,
                Double = 3, Struct = 4, Arr = 5;

    int kind; // Int, Char, Float, Double, Class ili Arr

    // atribut koji se koristi ako je kind = Arr
    TypeNode* elementType; // tip elemenata

    // atributi koji se koriste ako je kind = Struct
    int fieldsNo;           // broj članova
    ObjectNode* fields;     // članovi strukture
}

struct ObjectNode {
    // konstante kojima se definiše vrsta entiteta
    static const int Con = 0, Var = 1, Type = 2,
                Fld = 3, Meth = 4;

    String name;
    int kind;           // Con, Var, Type, Fld, Meth
    TypeNode* type;
    ObjectNode* next;

    // adresa promenljive, metode ili člana strukture
    int adr;

    // atributi koji se koristi za metode
    int parametersNo;    // broj parametara
    ObjectNode* parameters; // parametri funkcije

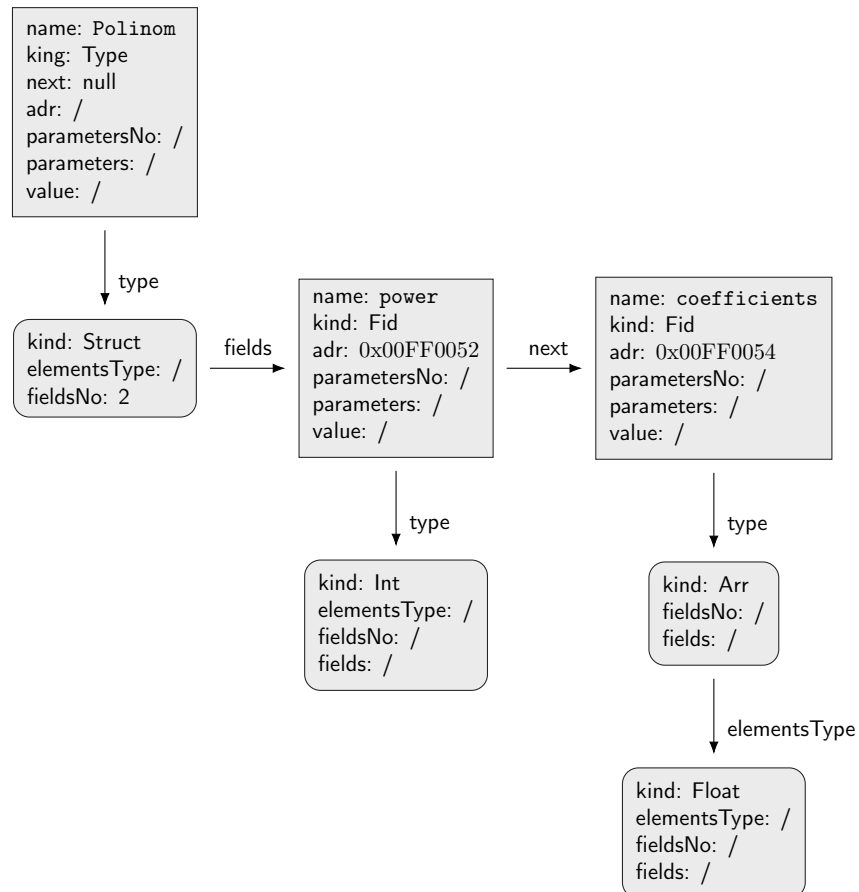
    // atribut koji se koristi za konstante
    void* value;         // vrednost konstante
};
```

LISTING 1: Predstavljanje simbola korišćenjem *flat* pristupa

predstavljena struktura `Polinom` definisana na sledeći način:

```
struct Polinom {
    int power;
    float* coefficients;
};
```

Zbog jasnoće prikaza, čvorovi kojima su predstavljeni objekti (imenovani entiteti) prikazani su pravougaonicima sa oštrim uglovima, a čvorovi kojima su predstavljeni tipovi pravougaonicima sa zaobljenim uglovima.



SLIKA 9.1: Graf kojim je u tabeli simbola predstavljena struktura `Polinom`



### 9.3 Predstavljanje simbola korišćenjem objektno-orijentisanog pristupa

Sa slike 9.1 se može uočiti da, ukoliko se koristi *flat* pristup za predstavljanje simbola, mnoga polja u strukturama ostaju neiskorišćena, tj. troši se mnogo više memorijskog prostora nego što je zaista potrebno za memorisanje podataka o simbolu. Zbog toga se danas češće koristi objektno-orijentisani pristup. Kod ovog pristupa bi osnovna klasa za predstavljanje imenovanih entiteta (`ObjectNode`) sadržala samo ono što je zajedničko za sve tipove imenovanih entiteta: ime (`name`), vrstu entiteta (`kind`) i pokazivač na sledeći (`next`). Ono što je karakteristično za pojedine tipove entiteta bilo bi definisano u izvedenim klasama. Analogno tome i za predstavljanje tipova bi bila definisana osnovna klasa (`TypeNode`) koja bi sadržala samo vrstu tipa, dok bi se u izvedenim klasama pamtili dodatni podaci vezani za konkretne tipove. Kompletan dijagram klasa za predstavljanje simbola u navedenom hipotetičkom programskom jeziku dat je na slici 9.2.

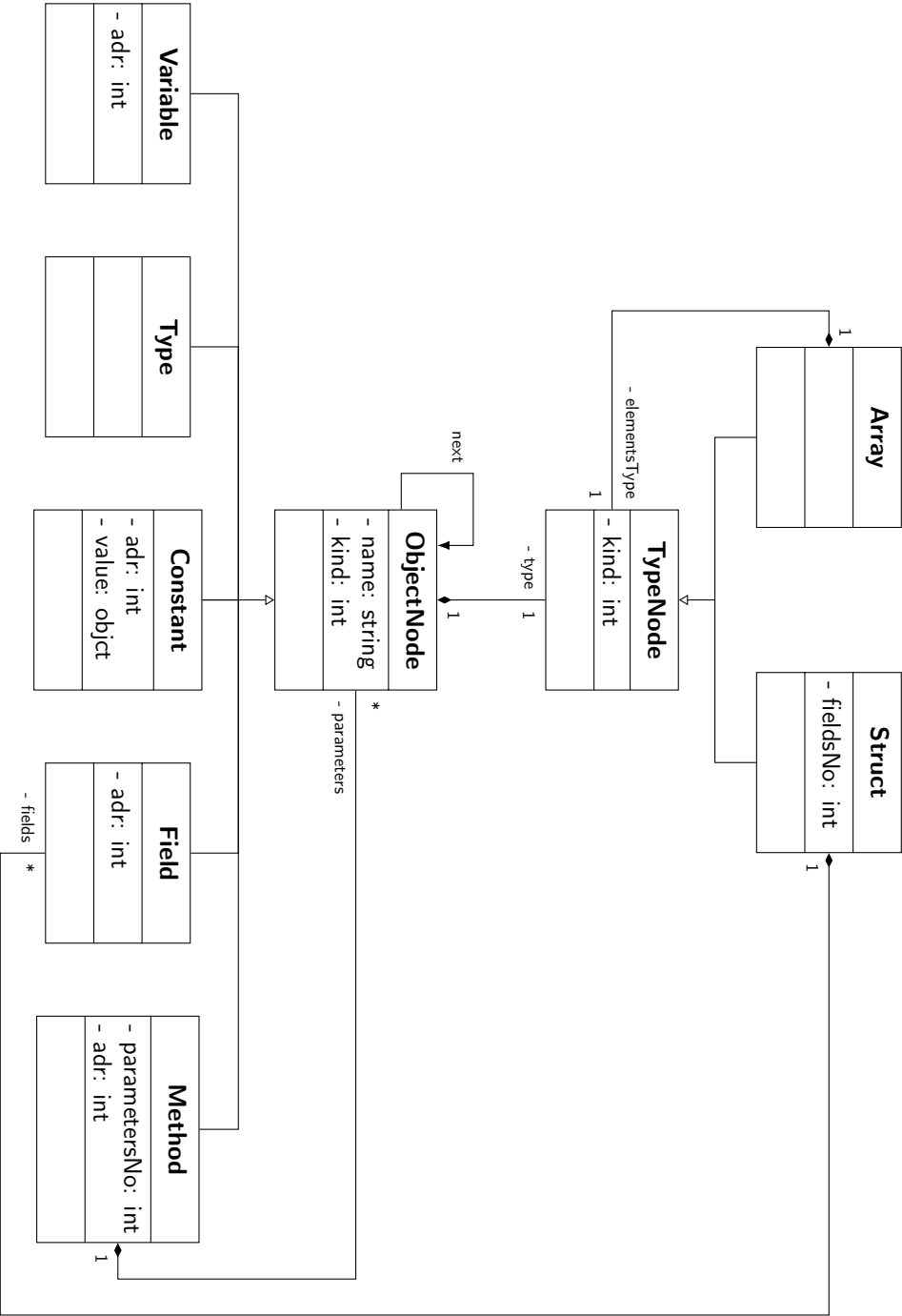
Kada se koristi objektno-orijentisani pristup za predstavljanje simbola, pri kreiranju novog simbola potrebno je znati vrstu entiteta koji on imenuje. Zbog toga mora da se malo promeni struktura kompilatora u odnosu na onu koja je prikazana na slici 4.1. U ovom slučaju leksički analizator, kada prvi put prepozna neko simboličko ime, ne može da kreira objekat koji će ga predstavljati u tabeli simbola jer ne zna kontekst u kojem je ime upotrebljeno. Zbog toga se struktura kompilatora neznatno menja i u novijim kompilatorima tabela simbola počinje da se koristi tek od faze sintaksne analize. Izmenjena struktura kompilatora prikazana na slici 9.3.

### 9.4 Predstavljanje tabele simbola

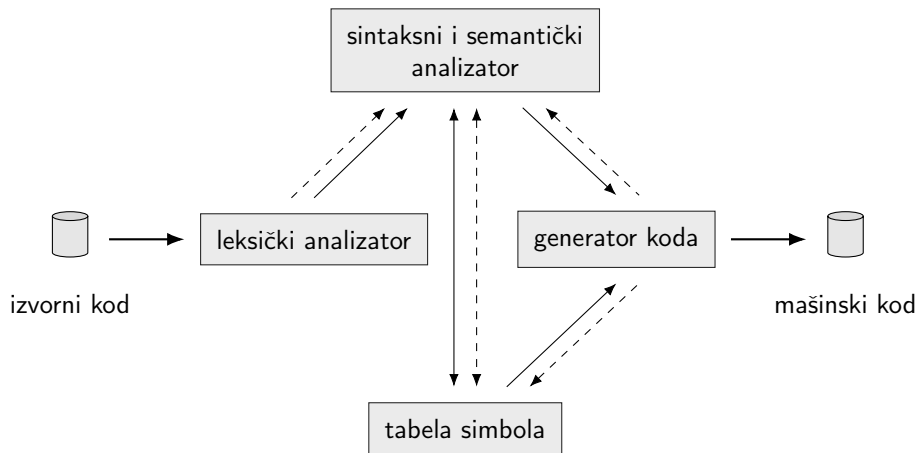
Sadržaj tabele simbola se stalno menja: dodaju se novi simboli kada se naiđe na novu definiciju, a izbacuju se simboli kada se napusti opseg njihovog važenja. Zato se za predstavljanje tabele simbola mora koristiti neka od dinamičkih struktura podataka. Najčešće korišćene su lančane liste, uređena binarna stabla i rasute (*hash*) tabele. Prednosti i nedostatke svake od ovih struktura ćemo analizirati tako što ćemo posmatrati deo tabele simbola koji se kreira na osnovu sledećeg dela koda:

```
void m() { /* ... */ }
class T { /* ... */ };
int a, b, c;
const int n = 10;
```

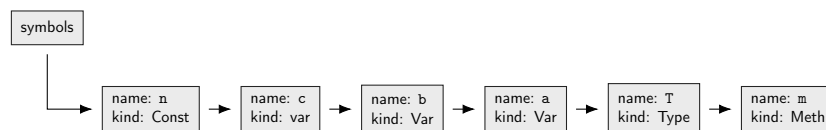
Tabela simbola realizovana kao lančana lista prikazana je na slici 9.4. Osobine tabele simbola realizovane na ovaj način su u stvari osobine same lančane liste kao strukture podataka:



Slika 9.2: Dijagram klasa za predstavljanje simbola



SLIKA 9.3: Izmenjena struktura kompilatora



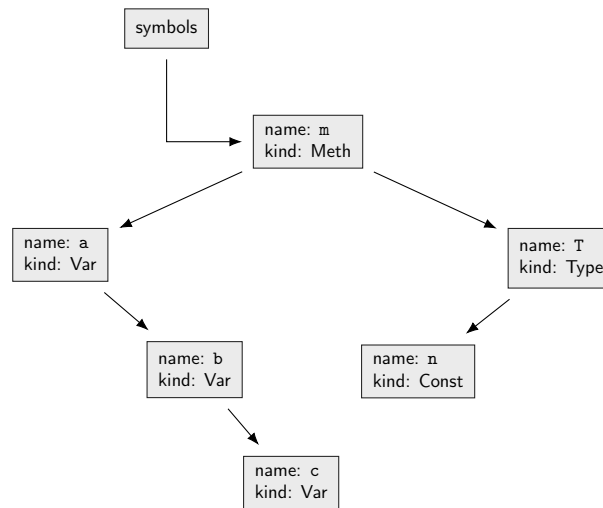
SLIKA 9.4: Tabela simbola realizovana kao lančana lista

- traženje simbola u ovakvoj strukturi je sporo (vrši se linearno traženje pa je prosečno vreme traženja  $n/2$ ),
- dodavanje novih simbola je jako brzo (dodavanje se uvek vrši na početak liste),
- brisanje simbola je, takođe, brzo (uvek se brišu najpre simboli sa početka liste što će biti objašnjeno u sledećem poglavlju).

Zbog navedenih osobina, tabele simbola realizovane na ovaj način se koriste u situacijama kada očekivani broj simbola u jednoj tabeli nije veliki.

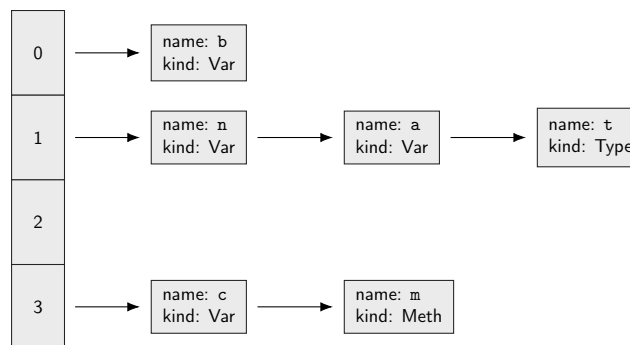
Tabela simbola realizovana kao uređeno binarno stablo prikazana je na slici 9.5. Osobine ovako realizovane tabele simbola su:

- traženje simbola je brže nego u slučaju lančane liste (prosečno vreme traženja u uređenom binarnom stablu je  $\log_2 n$ , ali, s obzirom da je ovako kreirano binarno stablo prilično neizbalansirano, to vreme može biti i mnogo duže),
- dodavanje simbola je sporije (jer se prvo vrši traženje mesta pa je i kompleksnost dodavanja simbola takođe  $\log_2 n$ )
- brisanje simbola jako sporo (vrši se najpre traženje simbola koji treba da budu obrisani po celom stablu, a onda i samo brisanje)



SLIKA 9.5

- utrošak memorijskog prostora je veće nego kod lančane liste jer se uz svaki element pamte po dva pokazivača na potomke.



SLIKA 9.6: Tabela simbola realizovana kao rasuta tablica

Na slici 9.6 je prikazana tabela simbola realizovana kao hash tabela pri čemu je pretpostavljeno da se za smeštanje sinonima koristi spoljašnje ulančavanje. Osobine ovako realizovane tabele simbola su:

- traženje simbola je izuzetno brzo ( $\approx 1$ ),
- dodavanje simbola takođe brzo ( $\approx 1$ ),
- brisanje simbola, takođe, sporo (vrši se najpre traženje simbola koji treba da budu obrisani u celoj tabeli, dok je samo uklanjanje simbola brzo) i
- utrošak memorijskog prostora je veći nego kod lančane liste jer u hash tabeli uvek ostaje dosta nepopunjenih polja (čak i kada se unapred zna

koliko će elemenata sadržati hash tabela, preporučljivo je da njena iskorišćenost bude 80%, a u slučaju tabele simbola broj elemenata koji se upisuju u nju je kranje nepredvidiv).

## 9.5 Tabele simbola i oblast važenja simboličkih imena

Vidljivost simboličkih imena u programu zavisi od mesta u kojem su definisana. Deo programa u kojem definisani simbol može da se koristi se naziva oblast važenja ili oblast vidljivosti simboličkog imena (*scope*). Većina savremenih programskih jezika je organizovano blokovski, tj. oblast važenja imena odgovara bloku u kojem su definisani. Kako se blokovi mogu ugnježdavati jedan u drugi, ime definisano u jednom bloku je vidljivo u tom bloku i u svim blokovima ugnježdenim u njega. Ako je isto ime definisano u više ugnježdenih blokova, u posmatranoj tački važi poslednja definicija, tj. definicija najbliža posmatranoj tački.

Na primer, u programskom jeziku C++ možemo uočiti sledeće opege važenja:

- opseg jezika – sadrži imena definisana u samom jeziku
- globalni oseg – sadrži imena definisana na globalnom nivou,
- opseg klase – sadrži članove klase,
- opseg funkcije – sadrži parametre funkcije i lokalne promenljive definisane u funkcijskom bloku i
- opseg bloka – sadrži promenljive definisane samo u tom bloku.

Na listingu 2 prikazan je jedan deo Java koda u kojem su obeležene oblasti važenja pojedinih imena. Uzet je programski jezik Java jer se u njemu implementacija funkcija članica klasa piše u samoj klasi pa je lakše uočiti ugnježdavanje opega.

Ako se uzme u obzir postojanje opsega, traženje imena u tabeli simbola bi trebalo modifikovati tako da se prvo vrši traženje u poslednjem otvorenom opsegu, pa tek ako se tu ime ne nađe, traži se u spoljašnjem, pa u prethodnom, itd. To znači da se u tabeli simbola mora da čuva i informacija o tome u kom opsegu je simbol definisan. Ovaj problem se rešava na dva načina:

1. svi simboli se smeštaju u jedinstvenu tabelu simbola pri čemu se uz svaki simbol pamti i nivo opsega na kojem je definisan i po zatvaranju bloka iz tabele se brišu svi simboli definisani u njemu, ili
2. za svaki opseg važenja imena se pravi posebna tabela simbola i one se smeštaju u stek i po zatvaranju bloka za koji je opseg vezan, iz steka se izbacuje tabela koja odgovara tom opsegu.

```

public class Foo {
    private int value = 39;
    public int test() {
        int b = 3;
        return value + b;
    }
    public void setValue(int c) {
        value = c;
        {
            int d = c;
            c = c + d;
            value = c;
        }
    }
}

public class Bar {
    private int value = 42;
    public void setValue(int c) {
        value = c;
    }
}

```

Diagram illustrating the scope of names in the provided Java code:

- Class Foo:**
  - Scope of `b`:** The block `int b = 3; return value + b;` is enclosed in a brace labeled "opseg imena b".
  - Scope of `c`:** The block `public void setValue(int c) { ... }` is enclosed in a brace labeled "opseg imena c".
  - Scope of `d`:** The block `{ int d = c; c = c + d; value = c; }` is enclosed in a brace labeled "opseg imena d".
  - Global Scope:** The entire class `Foo` is enclosed in a brace labeled "opseg imena value, test i setValue".
- Class Bar:**
  - Scope of `c`:** The block `public void setValue(int c) { value = c; }` is enclosed in a brace labeled "opseg imena c".
  - Global Scope:** The entire class `Bar` is enclosed in a brace labeled "opseg imena value i setValue".

LISTING 2: Jedan deo Java koda sa obeleženim oblastima važenja imena

Svaki od navedenih pristupa ima svoje prednosti i svoje nedostatke. Problemi koji se javljaju kada se koristi posebna tabela simbola za svaki opseg su sledeći.

- Ako se pristupa nekom imenu koje nije definisano u tekućem opsegu, pretražuje se veći broj tabela simbola što usporava traženje.
- Ako su tabele realizovane kao hash tabele, za svaki opseg se kreira nova tabela pa kada je broj imena u opsegu mali dolazi do velikog rasipanja memorijskog prostora. Zbog toga se u ovakvim slučajevima tabele simbola obično realizuju kao lančane liste.

Ukoliko se koristi jedinstvena tabela simbola za sve opsege javlja se, opet, problem trošenja dodatnog memorijskog prostora jer se uz svaki simbol pamti i opseg kojem pripada. Što se brzine traženja u tabeli tiče, prednost korišćenja hash tabela naročito dolazi do izražaja u ovom slučaju jer opseg kojem simbol pripada ne utiče na njegov položaj u hash tabeli pa je svedjedno da li je korišćeni simbol definisan u tekućem opsegu ili nekom dalekom spoljašnjem. Uređena binarna stabla su daleko najnepogodnija za jedinstvenu tabelu simbola, jer će simboli definisani u tekućem opsegu uvek biti na najnižim nivoima u stablu. Zbog toga se preporučuje da kada se koristi jedinstvena tabela simbola, koristi hash tabela, a kada se prave posebne tabele za svaki opseg, prednost se daje

lančanim listama i uređenim binarnim stablima.

### Primer 9.2

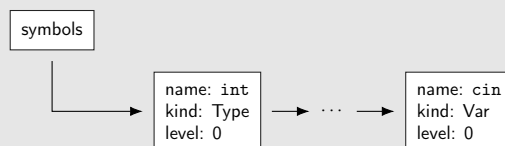
Prikazati sadržaj tabele simbola u obeleženim tačkama datog C++ koda. Posmatrati slučaj jedinstvene tabele i posebne tabele simbola za svaki opseg važenja.

```
// tacka 1
const double pi = 3.14;
class Polinom {
    int stepen;
    int koef[100];
    // tacka 2
    double vrednost(double x) {
        double s = 0;
        // tacka 3
        for (int i = 0; i < stepen; i++) {
            s = s * x + koef[i];
        }
    }
}
// ...
};

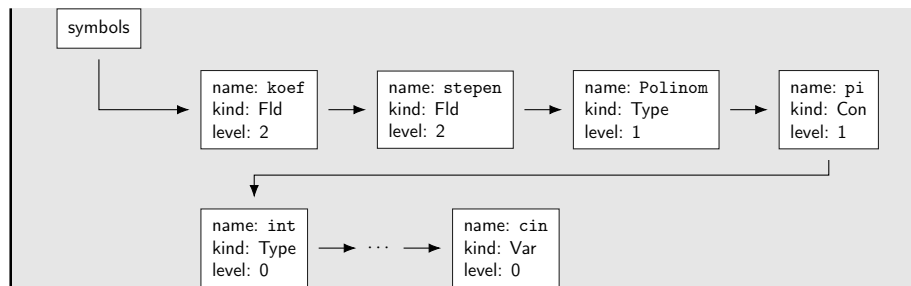
void main() {
    Polinom p;
    // tacka 4
    // ...
}
```

**Prvi slučaj.** Jedinstvena tabela simbola – pored svakog simbola se pamti i nivo na kojem je definisan.

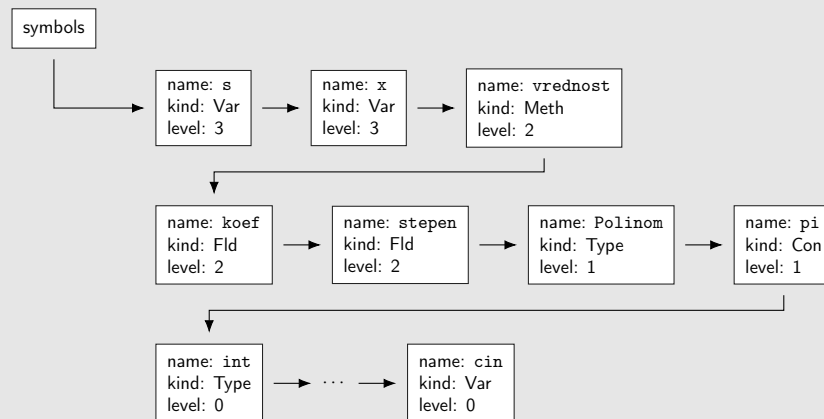
*Tačka 1:* U tabeli simbola su upisani samo simboli ugrađeni u programski jezik. Na primer, videti sledeću sliku.



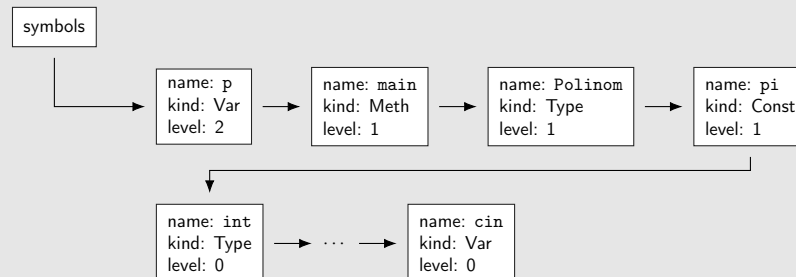
*Tačka 2:* Do ovog trenutka u kodu su definisana dva globalna objekta (konstanta Pi i klasa Polinom) i članovi klase Polinom (stepen i koef).



*Tačka 3:* Tabeli simbola je dodata metoda vrednost, njen parameter *x* i lokalna promenljiva *s*.

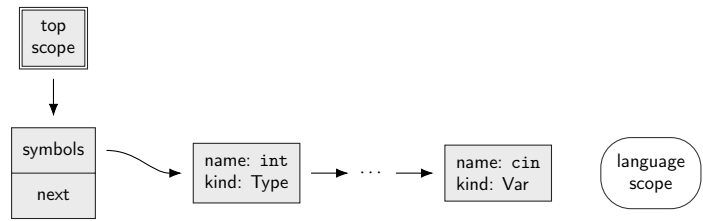


*Tačka 4:* Ova tačka se nalazi u opsegu funkcije *main*. Osim lokalnih promenljivih ove funkcije, u tački su vidljiva i imena iz spoljašnjih opsega (globalnog i opsega samog jezika).

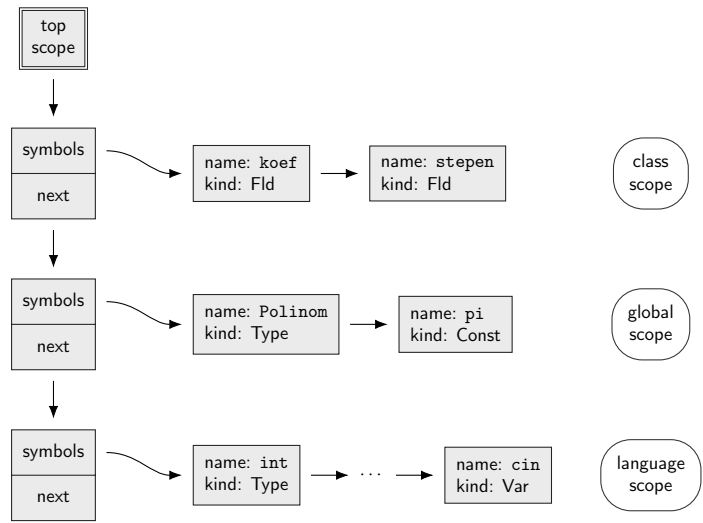


**Drugi slučaj.** Kreira se posebna tabela simbola za svaki opseg. Sadržaj tih tabela u navedenim tačkama 1, 2, 3 i 4, respektivno, je prikazan na slikama 9.7, 9.8, 9.9 i 9.10.

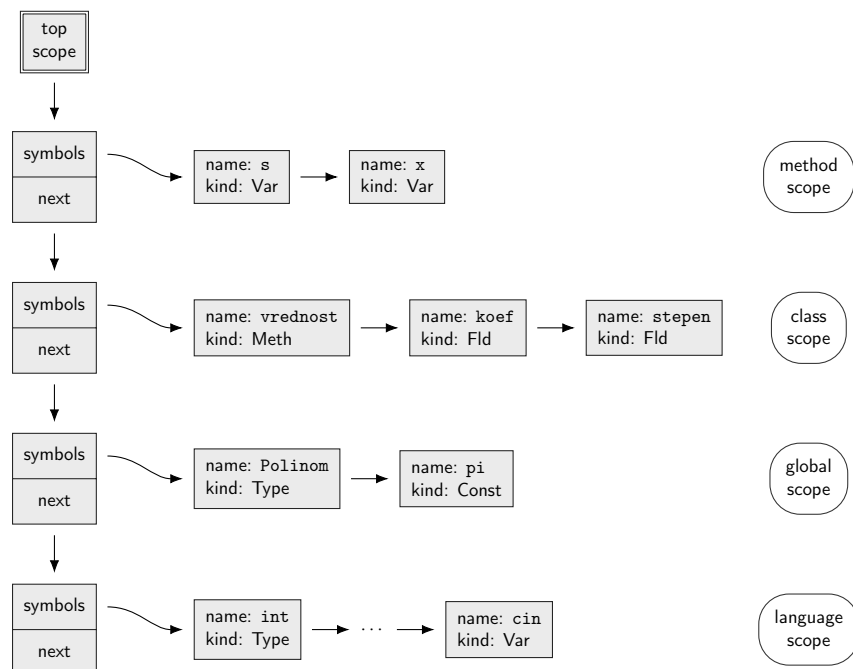




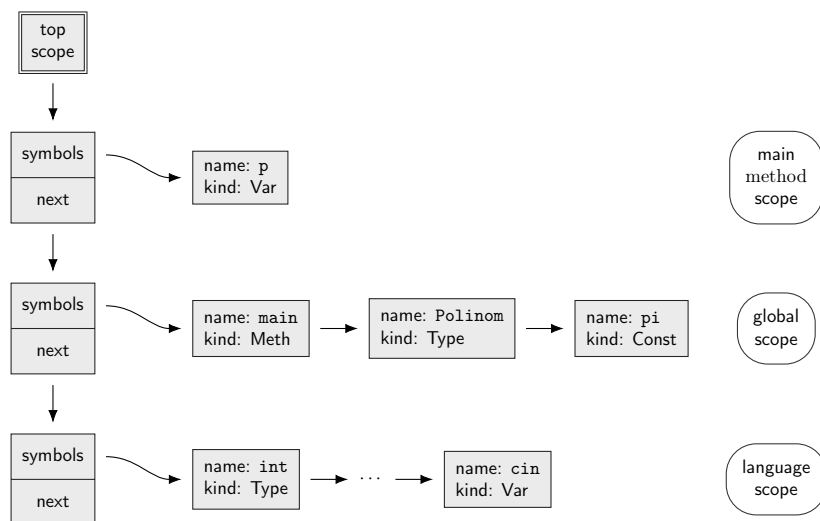
SLIKA 9.7: Tačka 1 za drugi slučaj uz primer 9.2



SLIKA 9.8: Tačka 2 za drugi slučaj uz primer 9.2



SLIKA 9.9: Tačka 3 za drugi slučaj uz primer 9.2



SLIKA 9.10: Tačka 4 za drugi slučaj uz primer 9.2

## 9.6 Pitanja

1. Šta je tabela simbola?
2. Navesti metode za predstavljanje podataka o tipovima i imenovanim entitetima u tabeli simbola.
3. Objasniti kako se tipovi i imenovani entiteti predstavljaju u tabeli simbola kada se koristi *flat* metoda implementacije.
4. Objasniti kako se tipovi i imenovani entiteti predstavljaju u tabeli simbola kada se koristi objektno-orijentisani pristup implementacije.
5. Navesti strukture podataka koje se koriste za predstavljanje tabele simbola
6. Navesti prednosti i nedostatke tabele simbola realizovane pomoću lančane liste.
7. Navesti prednosti i nedostatke tabele simbola realizovane pomoću uređenog binarnog stabla.
8. Navesti prednosti i nedostatke tabele simbola realizovane pomoću rasute tablice.
9. Kako se informacija o oblasti važenja simbola predstavlja u tabeli simbola.
10. Koja se struktura podataka najčešće koristi za predstavljanje tabele simbola ukoliko se organizuje jedinstvena tabela simbola za sve oblasti važenja? Zašto?
11. Koja se struktura podataka najčešće koristi za predstavljanje posebnih tabela simbola za svaku oblast važenja? Zašto?

## 9.7 Zadaci

1. Proširiti strukturu podataka koja se koristi za predstavljanje tipova u *flat* metodi tako da bude primenljiva i na jezike koje sadrže i klase kao korisničke tipove podataka. Grafički predstaviti strukturu podataka kojim bi se korišćenjem tako definisane strukture klasa *Krug* predstavila u tabeli simbola:

```
class Krug
{
    private:
        int r;
    public:
        void postaviR(int r);
        float obima();
}
```

```
float površina();  
};
```

2. Za zadati deo koda prikazati kako bi izgledala organizacija tabele simbola u obliku lančane u obeleženim tačkama, ako se koristi:

- (a) jedinstvena tabela simbola za sve opsege važenja;
- (b) posebna tabela simbola za svaki opseg.

```
public class Krug  
{  
    private int r;  
    /* tacka 1 */  
    void postaviR(int r)  
    {  
        this.r = r;  
        /* tacka 2 */  
    }  
    /* ... */  
};
```

## Glava 10

# Generatori leksičkih i sintaksnih analizatora

U ovom poglavlju biće predstavljeni generatori leksičkih analizatora Lex i JFlex i generatori sintaksnih analizatora Yacc i Cup.

Lex i Yacc su izabrani jer su to alati ovog tipa koji su se prvi pojavili. Razvijeni su *Bell*-ovim laboratorijama kao deo UNIX operativnog sistema još 1975. godine. Format njihovih specifikacija se, sa nekim sitnim izmenama, koristi i u svim alatima ove vrste koji su razvijani kasnije. Prve verzije ovih alata generišu kod na programskom jeziku C. Iako su početno definisani u UNIX-u, Lex i Yacc su kasnije postali i obavezan deo *Honeywell*-ovog operativnog sistema GCOS, kao i IBM-ovih operativnih sistema.

JFlex i Cup su alati koji generišu kod u jeziku Java. Njihova dobra osobina je što su to programi otvorenog koda (*open source*; kod im je besplatno raspoloživ na internetu), zbog čega nalaze veoma široku primenu u praksi.

### 10.1 Lex – generator leksičkih analizatora

Lex generiše kod koji prepoznaje instance zadatih regularnih izraza u ulaznom tekstu, prepoznaje u ulaznom tekstu podnizove koji odgovaraju zadatim regularnim izrazima. Osnovna namena Lex-a jeste generisanje leksičkih analizatora, ali se Lex može koristiti i u programima za bilo kakvo procesiranje teksta.

Lex generiše kod na osnovu ulazne Lex specifikacija koja sadrži tri osnovna dela koja se zapisuju na sledeći način:

```
<definicije>
%%
```

```
<pravila>
%%
<korisnički_potprogrami>
```

Lex specifikacija se upisuje u fajl čije ime obavezno ima ekstenziju `.l`.

### 10.1.1 Pravila u Lex specifikaciji

Centralni deo Lex specifikacije su pravila i to je jedini obavezni deo specifikacije. Minimalni sadržaj Lex specifikacije je:

```
%%
<pravila>
```

Jedno pravilo Lex specifikacije sadrži dva dela:

- **uzorak** – regularni izraz koji definiše sekvencu simbola koja se traži u ulaznom tekstu;
- **akciju** – kod na odredišnom jeziku (na programskom jeziku na kojem alat generiše izlazni kod) koji će se izvršiti kada se navedeni uzorak prepozna. Akcija se od uzorka odvaja bar jednim belim simbolom (blanko znakom, tabulatorom, ili prelazom na novi red).

Kada se Lex koristi za generisanje leksičkog analizatora, uzorcima se definišu regularni izrazi reči (leksema) programskog jezika za koji se analizator pravi, a akcija se obično završava return naredbom koja vraća token koji odgovara prepoznatoj reči.

U regularnim izrazima za definisanje uzorka mogu da se koriste tekst simboli (koji se baš takvi kakvi jesu porede sa znacima iz ulaznog teksta) i metasimboli koji imaju svoje specijalno značenje. U tabeli 10.1 prikazan je skup metasimbola i njihovih značenja u regularnim izrazima Lex specifikacije.

#### Primer 10.1 Regularni izrazi za prepoznavanje numeričkih konstanti

Krenimo od najjednostavnijeg oblika numeričkih konstanti. U većini programskih jezika definisan je neoznačeni ceo broj kao niz dekadnih cifara koji ne sme da počne nulom. Regularni izraz koji odgovara ovoj definiciji neoznačenog celog broja je

$$[1-9][0-9]^*.$$

Ovim regularnim izrazom izdvajaju se podnizovi koji počinju nekim od simbola iz intervala  $[1 \dots 9]$ , nakon kojeg se simboli iz intervala  $[0 \dots 9]$  pojavljuju nijednom ili više puta.

Sledeće proširenje definicije numeričkih konstanti dobija se uvođenjem pred-

TABELA 10.1: Metasimboli za definisanje regularnih izraza u Lex specifikaciji

ZNAK	ZNAČENJE
+	prethodni znak ili grupa se ponavlja jednom ili više puta
*	prethodni znak ili grupa se ponavlja nijednom ili više puta
?	prethodni znak može ali i ne mora da se pojavi
{n}	prethodni znak ili grupa se ponavlja tačno $n$ puta
{n,m}	prethodni znak ili grupa se ponavlja minimalno $n$ , a maksimalno $m$ puta
{name}	poziv makroa (ime lex promenljive)
[ ]	alternativa – izbor jednog od navedenih simbola unutar zagrade
-	sa prethodnim i narednim znakom definiše opseg simbola – može se koristiti samo unutar alternative
^	Koristi se na početku alternative ili na početku regularnog izraza. Ukoliko se nalazi na početku alternative, označava negaciju skupa znakova koji sledi – vrši se izbor jednog od simbola koji nisu navedeni u alternativni. Ukoliko se nalazi na početku regularnog izraza, označava da se taj regularni izraz primenjuje samo za izdvajanje podstringova (particija) sa početka linije.
\$	koristi se na kraju regularnog izraza i označava da se taj regularni izraz primenjuje samo za prepoznavanje podnizova koji se nalaze na kraju linije
/	označava da se podnizovi definisani regularnim izrazom koji prethodi izdvajaju samo ukoliko ukoliko se iza njih nađe podniz definisan regularnim izrazom koji sledi
( )	grupisanje simbola
\	poništava specijalno značenje metakaraktera koji sledi poništavaju specijalna dejstva svih metasimbola između
	izbor alternative (bira se između simbola (ili grupe) koji prethodi i simbola (ili grupe) koji sledi
!	negira znak koji sledi – na toj poziciji se može pojaviti svaki znak osim navedenog
~	prihvata sve simbole do pojave znaka ili grupe koja sledi, uključujući i taj znak/grupu
.	zamenjuje bilo koji znak
<name>	koristi se na početku regularnog izraza i označava da se primenjuje samo ukoliko prepoznavanje podniza počinje od navedenog stanja

znaka, čime se dolazi do definicija celog broja. Celi broj se definiše kao označeni ili neoznačeni niz dekadnih cifara, može ali ne mora da sadrži predznak. Odgovarajući regularni izraz je

$$[+-]?[1-9][0-9]^*$$

Konačno, numerička konstanta se u većini programskih jezika definiše kao označeni ili neoznačeni niz dekadnih cifara koji opciono sadrži decimalnu tačku i/ili eksponent. Kompletan regularni izraz za prepoznavanje numeričkih konstanti je:

$$[+-]?[0-9]+(\.[0-9]+)?([Ee][=-]?[1-9]+)?$$

\. označava da se na toj poziciji očekuje pojavljivanje baš tačke (ne bilo kog simbola što tačka kao metasimbol označava). Decimalna tačka i niz cifara iza nje su opciono, zbog toga su udruženi u grupu (navedeni unutar okruglih zagrada) iza koje stoji simbol ?. Na isti način je navedeno i opciono pojavljivanje eksponenta.

#### **Primer 10.2** Regularni izraz za prepoznavanje ključnih reči i simboličkih imena

U nastavku je dat regularni izraz za prepoznavanje ključnih reči i simboličkih imena.

$$_*[a-zA-Z][a-zA-Z0-9_]^*$$

#### **Primer 10.3** Regularni izrazi za prepoznavanje operatora u čijem zapisu figurišu metasimboli

Operator ++, u kojem se dva puta pojavljuje znak + koji se koristi i kao metasimbol, može da se definiše bilo kojim od sledećih regularnih izraza:

$$"++", \quad \backslash+\backslash+ \quad \text{ili} \quad \backslash+\{2\}.$$

#### **Primer 10.4** Navođenje alternativa

Regularni izrazi za prepoznavanje logičkih konstanti mogu da se predstavljaju na jedan od sledećih dva načina, pri čemu se znak | koristi za razdvajanje alternativa.

$$"true"|"false" \quad \text{ili} \quad (true)|(false).$$

#### **Primer 10.5** Korišćenje simbola ^



Programska linija koja na prvoj poziciji ima slovo C predstavlja komentar u programskom jeziku FORTRAN. To znači da početak komentara možemo definisati regularnim izrazom

$\text{^C.}$

#### Primer 10.6 Korišćenje simbola \$

Regularni izraz za prepoznavanje simbola koji u programskom jeziku BASIC označava da se tekuća naredba nastavlja u sledećem redu.

$\text{-\$}$

### 10.1.2 Definicije u Lex specifikaciji

Definicije u Lex specifikaciji mogu da sadrže dva dela.

- Deo koji se prepisuje na početak generisanog fajla. Ovaj deo počinje linijom koja sadrži samo  $\text{\%{\}}$  i završava se linijom koja sadrži  $\text{\%}$ . Navedeni delimiteri se zapisuju obavezno počev od prve pozicije. U okviru ovog dela navode se:
  - definicije globalnih tipova i podataka;
  - include direktive i slično.
- Lex definicije koje obuhvataju:
  - definicije specijalnih početnih stanja i
  - definicije makroa.

### 10.1.3 Specijalna početna stanja

Prilikom prepoznavanja svake nove reči, polazi se od unapred definisanog početnog stanja konačnog automata. Po podrazuemvanim podešavanjima, početno stanje generisanog konačnog automata je stanje koje ima redni broj 0. Nekada se javlja situacija da iste reči imaju različita značenja zavisno od konteksta u kojem su upotrebljene. Da bi se ta specijalna značenja pojedinih reči prepoznala uvode se nova početna stanja konačnog automata koji prepoznaje reči jezika.

Specijalna početna stanja mogu da budu:

- Potpuno definisana specijalna početna stanja – za njih treba definisati prelaze za svaki simbol ulazne azbuke. U engleskoj literaturi se ovakva stanja nazivaju *exclusive states*. Simbolička imena takvih stanja se u Lex specifikaciji navode u liniji koja počinje sa  $\text{\%x}$ .

- Parcijalno definisana specijalna početna stanja – za njih se mogu definisati samo prelazi za one simbole koje se nalaze na početku reči koje imaju drugačije značenje od onog koje imaju kada analiza počinje od početnog stanja 0. Za prepoznavanje onih reči koje imaju isto značenje i u jednom i u drugom slučaju koristiće se uzorci navedeni za podrazumevano početno stanje. Za ovakva stanja se u engleskoj literaturi koristi termin *inclusive states*. Simbolička imena takvih stanja se u Lex specifikaciji navode u liniji koja počinje sa %s.

Prelaz na korišćenje novog početnog stanja prilikom izdvajanja sledeće reči (podniza) koristi se funkcija **begin**.

Komentari u programskom jeziku C počinju simbolom // i završavaju se prelazom na novi red ili se navode između simbola /\* i \*/. Reči navedene u komentaru imaju drugačije značenje nego kad su van komentara (tj. nemaju nikakvo značenje). Zato definišemo dva specijalna početna stanja – za prepoznavanje tj. ignorisanje sadržaja komentara, kao što je prikazano u sledećem primeru.

**Primer 10.7** Deo Lex specifikacije za ignorisanje komentara u programskom jeziku C

```
%x COMMENT1 COMMENT2
%%
\\\/ begin(COMMENT1);
<COMMENT1> \n begin(0);
<COMMENT2> . ;
\\\/* begin(COMMENT2);
<COMMENT2> \*\\\/ begin(0);
<COMMENT2> . ;
```

#### 10.1.4 Definicije makroa

Ukoliko postoje regularni izrazi koji se koriste u većem broju uzoraka, ili su neki uzorci vrlo komplikovani, moguće je definisati makro čija je vrednost jednaka nekom regularnom izrazu, a onda se u uzorcima koji sadrže takve regularne izraze vrši poziv makroa.

Format za definisanje makroa:

ImeMakroa RegularniIzraz

Kao što je u tabli 10.1 već navedeno, u uzorcima se makro poziva navođenjem imena makroa, na sledeći način:

{ImeMakroa}

**Primer 10.8** Regularni izraz za izdvajanje numeričkih konstanti sa korišćenjem makroa

```
cifra [0-9]
%%
[+-]?{cifra}+(\.{cifra}+)?((Ee)[+-]?{cifra}+)?
```

### 10.1.5 Korisnički potprogrami

Da bi Lex specifikacija bila preglednija, treba se truditi da kod naveden u akcijama bude što kraći. Ukoliko akcija treba da bude komplikovanija, uobičajeno je da se u delu za korisničke potprograme implementira funkcija, koja se samo poziva u okviru akcije. Deo sa korisničkim potprogramima sadrži C funkcije koje se kompletno prepisuju u generisani kod.

### 10.1.6 Upotreba Lex generatora

Lex kompilator se poziva sa

```
lex <ime_specifikacije>.l
```

i on generiše izlazni fajl pod imenom `lex.yy.c`. Kada se ovaj kod dalje prevodi UNIX-ovim kompilatorom `cc`, treba u pozivu kompilatora navesti opciju `-ll`, tj. generisani kod se dalje prevodi komandom:

```
cc -ll lex.yy.c.
```

U generisanom C kodu, glavna funkcija, funkcija koja izdvaja sledeći podstring iz ulaznog teksta, ili ako se radi o leksičkom analizatoru, funkcija koja izdvaja sledeću reč programskog jezika je funkcija `int yylex()`.

Funkcija vraća rezultat tipa `int`, jer se u daljem toku analize sve reči programskog jezika identifikuju pomoću svog jedinstvenog celobrojnog identifikatora (tokena). Osim funkcije `yylex()`, Lex generiše i niz pomoćnih promenljivih i funkcija koje definiše na globalnom nivou, tako da korisnik može da ih koristi u svojim akcijama ili korisničkim potprogramima.

Najbitnije promenljive i funkcije koje lex generiše su:

- `char* yytext` – izdvojena reč;
- `int yylength` – dužina izdvojene reči;
- `void yyless(int n)` – dužinu izdvojene reči menja na `n` karaktera (prvih `n` karaktera iz reči `yytext` zadržava, preostale vraća u ulazni tok);
- `void yybegin(int state)` – menja početno stanje konačnog automata;

- **FILE\*** `yyin` – datoteka iz koje se čita ulazni tekst; ukoliko korisnik ne otvori ovu datoteku pre prvog poziva funkcije `yylex()`, učitavanje podataka će se vršiti sa standardnog ulaza.

### 10.1.7 Primer kompletne Lex specifikacije

U okviru ovog poglavlja generisaćemo prevodilac za programski jezik definisan sledećom gramatikom.

$$\begin{aligned}
 \textit{Program} &\rightarrow \textit{NizNaredbi} \backslash \mathbf{n} \textit{ end} \\
 \textit{NizNaredbi} &\rightarrow \textit{NizNaredbi} \backslash \mathbf{n} \textit{ Naredba} \mid \textit{Naredba} \\
 \textit{Naredba} &\rightarrow \textit{Ulaz} \mid \textit{Izlaz} \mid \textit{Dodela} \\
 \textit{Dodela} &\rightarrow \mathbf{id} = \textit{Izraz} \\
 \textit{Izraz} &\rightarrow \textit{Izraz} + \textit{Proizvod} \\
 \textit{Proizvod} &\rightarrow \textit{Proizvod} * \textit{Faktor} \\
 \textit{Faktor} &\rightarrow (\textit{Izraz}) \mid \mathbf{id} \mid \mathbf{cons} \\
 \textit{Ulaz} &\rightarrow \mathbf{read}(\mathbf{id}) \\
 \textit{Izlaz} &\rightarrow \mathbf{write}(\textit{Izraz})
 \end{aligned}$$

Neka u jeziku postoje samo numeričke konstante i neka konstante i identifikatori imaju isti zapis kao i u programskom jeziku C.

Uočimo najpre tipove reči u ovom jeziku:

1. ključne reči **end**, **read**, **write**;
2. operatori `+`, `*`;
3. separatori `=`, `(`, `)`, `\n`;
4. simbolička imena **id**;
5. konstante **const**.

Za svaki tip reči treba definisati neki ceo broj koji ga identifikuje. Kako izvesni specijalni znaci sami predstavljaju posebnu vrstu reči (za konkretan jezik to su znaci `+`, `-`, `=`, `(` i `)`), za njih se ne moraju definisati posebni identifikatori – mogu se identifikovati svojim ASCII kodom. Za ostale tipove reči treba eksplicitno definisati konstante koje ih identifikuju. Treba voditi računa o tome da identifikatori tih reči ne smeju da se poklapaju sa bilo kojim ASCII kodom.

Za izvesne tipove reči za dalji tok prevođenja nije dovoljno znati samo kojoj vrsti pripadaju. Recimo, ako se radi o konstanti, treba znati koja je njena vrednost, ako se radi o simboličkom imenu, treba znati njegovo ime (a kasnije i ostale informacije – šta imenuje, koji mu je tip...). Informacije o simboličkim imenima

se pamte u tabeli simbola i leksički analizator, kako je ranije rečeno, proverava da li je prepoznato ime već upisano u tabelu simbola. Ako nije, upisuje ga i daljem stepenima analize treba da dostavi neki ukazatelj na podatke o tom simbolu. U ovom jednostavnom primeru, tabela simbola će biti organizovana kao jedan običan niz, pa će leksički analizator kada prepozna simboličko ime, vraćati i poziciju u tabeli simbola gde je taj simbol zapisan.

Ovakve dodatne informacije o izdvojenoj reči se obično nazivaju atributima i kako je pokazano tipovi podataka kojim se predstavljaju ti dodatni atributi su različiti za različite vrste reči. Kako je pokazano, u ovom slučaju za konstante je dodatni atribut realan broj (vrednost konstante), a za simbolička imena ceo broj (redni broj u tabeli simbola gde su zapisani). Zbog toga se tip podataka kojima se predstavljaju dodatni atributi simbola, uglavnom, definišu kao unije.

Identifikatore reči (tokene) i tipove dodatnih atributa definišaćemo u posebnom fajlu. Neka je to fajl `Tokens.h`.

**Primer 10.9** Definicije tokena i tipa podataka za predstavljanje dodatnih atributa tokena

```
// Tokens.h
#define END      257
#define READ    258
#define WRITE    259
#define ID       260
#define CONST    261
#define ERRSIMBOL 262

union AttributeType
{
    float value;
    int    index;
};
```

O strukturama podataka za predstavljanje podataka o simboličkom imenu je bilo reči u poglavlju 9. Pošto u našem primeru simbolička imena identifikuju samo promenljive, ta struktura će biti nešto jednostavnija. Jedino što leksički analizator zna o simbolu je njegovo ime. Ostali elementi stukture su vezane za naredne nivoe prvođenja. Zbog toga ćemo u ovom trenutku tu strukturu ostaviti nepopunjenu. Nju ćemo definisati u fajlu `Symbol.h`.

**Primer 10.10** Nepotpuna definicija tipa podataka za predstavljanje simbola u tabeli simbola

```
// Symbol.h
struct Symbol
{
    char name[32];
    // ...
};
```

Kreiraćemo i samu lex specifikaciju koja će sadržati i `main` funkciju. U `main` funkciji će se pozivati funkcija za izdvajanje sledeće teči iz ulaznog teksta dok se ne dođe do kraja ulazne datoteke. Za svaku izdvojenu reč, na standardni izlaz prikazaće se kako izgleda, njen token i atribut ukoliko je definisan.

**Primer 10.11** Kompletna Lex specifikacija za prepoznavanje reči datog jezika

```
// MyLang.l - LEX specifikacija
%{
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include "Tokens.h"
#include "Symbol.h"

struct Symbol symbolTable[200];
unsigned symbolNo;
union AttributeType attribute;
unsigned lineNo;
int getSymbol();
%}

/* definicije makroa */
cifra  [0-9]
slovo  [a-zA-Z]

%%
/* pravila */
/* kljucne reci */
"end"   return ( END );
"read"  return ( READ );
"write" return ( WRITE );
```

```
/* simbolicka imena */
{slovo}|_({slovo}|{cifra}|_)*
{
    attribute.index = getSymbol();
    return (ID);
}

/* konstante */
{cifra}+(\.{cifra})*?([Ee][+-]?{cifra}+)?
{
    attribute.value = atof(yytext);
    return (CONST);
}

/* ignorisanje belih simbola */
[\t ];

/* prelaz na novi red oznacava kraj naredbe */
\n { lineNo++; return yytext[0]; }

/* operatori i separatori */
[+*=( );] return( yytext[0] );

/* bilo koji drugi simbol tretira se kao greska */
. return ( ERRSYMBOL );

%%
/* korisnicki potprogrami */

// funkcija koja trazi izdvojenu rec u tabli simbola
// ako je ne nađe, upisuje je
int getSymbol()
{
    int i;
    for (i = 0; i < symbolNo &&
        strcmp(symbolTable[i].name, yytext) != 0;
        i++);
    if (i == symbolNo)
    {
        strcpy(symbolTable[i].name, yytext);
        symbolNo++;
    }
    return i;
}
```

```

void main(int argv, char** args)
{
    int token;
    yyin=fopen(args[1], "r");

    symbolNo=0;
    lineNo=1;

    /* funkcija za izdvajanje nove reči se poziva
    dok se ne dođe do karaja ulaznog fajla,
    kada yylex vraća nulu */
    while ((token = yylex()) !=0)
    {
        if (token == ERRSIMBOL)
        {
            printf("nepoznat simbol %s u liniji %d\n", yytext,
                  lineNo);
        }
        else
        {
            printf("rec: %s, token: %d", yytext, token);
            switch (token)
            {
                case CONST:
                    printf(" vrednost: %f\n", attribute.value);
                    break;
                case ID:
                    printf(" r. br. u tabeli: %d\n", attribute.index);
                    break;
                default:
                    printf("\n");
            }
        }
    }
    fclose(yyin);
}

```

## 10.2 Yacc – generator sintaksnih analizatora

Yacc je alat za generisanje sintaksnih analizatora. Razvijen je u isto vreme kad i Lex i osnovna je ideja da ova dva alata rade u sprezi. Yacc generiše LR sintakсни analizator na osnovu ulazne Yacc specifikacije.

Yacc specifikacija, po strukturi potpuno odgovara Lex specifikaciji i sastoji se



od sledećih delova:

```
definicije
%%
pravila
%%
korisnički potprogrami
```

Obavezni deo, kao i u Lex specifikaciji, su pravila, dok su definicije i korisnički potprogrami opcioni.

### 10.2.1 Definicije u Yacc specifikaciji

Deo za definicije sadrži zaglavlje koje se direktno umeće u generisani analizator i posebne Yacc definicije.

Zaglavlje koje se upisuje u generisani C program ima isti format i istu ulogu kao u Lex specifikaciji (počinje linijom sa simbolom `%{`, a završava se linijom sa simbolom `%}`).

Yacc definicijama se definišu terminalni simboli gramatike (tokeni), startni simbol gramatike, prioritet i asocijativnost operatora, tip dodatnih atributa terminalnih i neterminalnih simbola gramatike.

### 10.2.2 Definicija startnog simbola gramatike

Startni simbol gramatike definiše se na sledeći način:

```
%start <ime_startnog_simbola>
```

### 10.2.3 Definicije terminalnih simbola gramatike

Terminalni simboli gramatike definišu se na sledeći način:

```
%token <ime_term_simbola_1> [<ime_term_simbola_2>]...
```

Kao što je već rečeno, ukoliko nekom tokenu odgovara samo jedna reč dužine jedan znak, moguće je da se za takav token ne koristi poseban identifikator (tj. da se on ne navede u ovoj listi tokena), već da se odgovarajuća leksema u smenama navodi pod literalima.

Pošto se Yacc imena tokena zamenjuju celim brojevima, imena tokena ne smeju biti ključne reči programskog jezika u kojem se generiše kod analizatora.

### 10.2.4 Definicije prioriteta i asocijativnosti operatora

Operatori mogu biti levo-asocijativni, desno-asocijativni i neasocijativni. Za svaki tip asocijativnosti postoji posebna Yacc definicija (`%left`, `%right`, `%nonassoc`). Prioritet operatora je definisan redosledom navođenja njihovih asocijativnosti i to tako što se uvek navode najpre asocijativnosti operatora najnižeg prioriteta. Na primer, asocijativnost i prioriteti osnovnih aritmetičkih operatora u yacc specifikaciji mogu biti definisani na sledeći način:

```
%left '+', '-'
%left '*', '/'
%right STEPEN
```

Operatori, koji su navedeni u definicijama asocijativnosti i prioriteta, ne moraju biti navedeni u listi tokena.

### 10.2.5 Definicije tipova atributa terminalnih i neterminalnih simbola

Ukoliko se eksplicitno de definiše, tip dodatnih atributa svih simbola gramatike (terminalnih i neterminalnih) je `int`. Ukoliko tip `int` nije dovoljan za predstavljanje dodatnih atributa svih simbola, u Yacc specifikaciji može eksplicitno da bude definisan tip dodatnih atributa simbola. S obzirom da se za različite simbole kao dodatni atributi pamte podaci različitog tipa, tip dodatnih atributa se definiše kao unijana sledeći način:

```
%union
{
    <type1>    <value1>;
    <type2>    <value2>;
    ...
    <typeN>    <valueN>;
}
```

Da se ne bi, pri korišćenju dodatnih atributa, stalno pisala komponenta koja je aktivna, u Yacc specifikaciji se može za svaki terminalni i neterminalni simbol definisati koji član iz navedene unije se koristi kao njegov atribut. Definisanje tipa dodatnih atributa terminalnih simbola se navodi u definiciji `%token`, dok se za definisanje tipa dodatnih atributa neterminalnih simbola koristi definicija `%type`.

Format definicije `%token` koja određuje i tip dodatnih atributa navedenih tokena je:

```
%token<active_value> <token_list>.
```

Format definicije `%type` je:

```
%type<active_value> <nonterminal_list>.
```

**Primer 10.12** Definicije tipova dodatnih atributa terminalnih i neterminalnih simbola u Yacc specifikaciji

```
%union
{
    float floatValue;
    int intValue;
    struct Symbol
    {
        char name[32];
        char type[12];
    } symbolValue;
    /* ... */
}

/* ... */

%token<floatValue>  CONST
%token<symbolValue> ID
%type<intValue>     PARAMETERLIST
                    // npr. atribut je broj parametara
/* ... */
```

### 10.2.6 Pravila u Yacc specifikaciji

Deo za pravila počinje linijom sa simbolom `%`. Ako u specifikaciji postoje i korisnički potprogrami, isti simbol se koristi i za razdvajanje pravila od korisničkih potprograma.

Za svaku smenu gramatike kreira se po jedno pravilo u Yacc specifikaciji. Pravilo sadrži smenu i akciju koja će se izvršiti kada se u postupku LR sintaksne analize izvrši redukcija po navedenoj smeni.

Pravilo u Yacc specifikaciji ima sledeći format:

```
A : TELO [ AKCIJA ];;
```

gde je:

- A – neterminalni simbol sa leve strane smene;
- TELO – desna strana smene;

- **AKCIJA** – blok naredbi koji se izvršava u trenutku redukcije po navedenoj smeni.

Ukoliko u gramatici postoji veći broj smena koje preslikavaju isti neterminalni simbol, moguće je za svaku smenu pisati nezavisno pravilo; npr:

```
A : B C D;
A : E F;
A : G;
```

Alternativno, može se koristiti skraćeni zapis za višestruko preslikavanje:

```
A : B C D
    | E F
    | G
    ;
```

U akcijama se vrlo često koriste vrednosti dodatnih atributa simbola koji učestvuju u smeni za koju je akcija vezana. Promenljive koje se koriste kao dodatni atributi simbola su:

- **\$\$** – atribut simbola koji se nalazi na levoj strani smene,
- **\$k** – atribut *k*-tog simbola na desnoj strani smene.

Ukoliko je za definisanje tipa dodatnih atributa simbola korišćena definicija **%union**, a nije eksplicitno specificiran tip simbola (definicijom **%token** ili **%type**), pri korišćenju ovih simbola potrebno je pisati:

```
<ime_simbola>.<ime_aktivnog_clana>.
```

Ako je tip simbola specificiran, dovoljno je pisati samo ime simbola. Ukoliko u pravilu nije navedena akcija, podrazumevana akcija je:

```
{ $$ = $1; }
```

S obzirom da tip atributa simbola koji se nalazi na levoj strani smene i prvog simbola na desnoj strani ne mora da bude isti, korišćenje ove podrazumevane akcije je vrlo rizično (kompajler može da prijavi grešku u neslaganju tipa koju vi u kodu ne možete da uočite). Zbog toga je preporučljivo da se, kada prilikom neke redukcije analizator ne treba ništa dodatno da uradi, odgovarajućoj smeni pridruži akcija bez dejstva:

```
{ ; }
```

### 10.2.7 Korisnički potprogrami

Osim za implementaciju funkcija koje se pozivaju iz akcija, u ovom delu u Yacc specifikaciji može da bude implementiran i leksički analizator. Ukoliko se u ovom delu implementira leksički analizator on mora da ima istu deklaraciju kao i leksički analizator koji generiše Lex (funkcija **yylex()** tipa **int**).

### 10.2.8 Upotreba Yacc generatora

Yacc kompilator se poziva komandom

```
yacc [opcije] ime.y.
```

Generisani kod biće u fajlu `y.tab.c`. Prilikom prevođenja ovog fajla UNIX-ovim `cc` kompilatorom, treba navesti opciju `-ly`.

Ukoliko leksički analizator nije implementiran u okviru Yacc specifikacije, tabela tokena, kao i tip podataka za predstavljanje dodatnih atributa simbola treba da bude vidljiva i onom fajlu u kojem se nalazi implementacija leksičkog analizatora. Da bi tablica tokena i definicija tipa atributa bili izdvojeni u poseban fajl, u pozivu yacc kompilatora treba navesti opciju `-d`. U tom slučaju Yacc generiše i fajl pod imenom `y.tab.h`.

Kao što je već rečeno, Yacc generiše LR sintaksni analizator. Ako korisnik želi da vidi kako izgleda LR sintaksna tabela koju Yacc generiše, pri pozivu Yacc kompilatora treba navesti opciju `-v` i generisana sintaksna tabela biće zapisana u datoteci sa imenom `y.out`.

Funkcija za sintaksnu analizu, generirana na osnovu Yacc specifikacije je funkcija:

```
int yyparse();
```

Rezultat ove funkcije je 0 ukoliko je analiza uspešno izvršena do kraja ulaznog fajla.

Tip dodatnih atributa simbola je `YYSTYPE`. Globalna promenljiva koja se koristi za privremeno pamćenje dodatnih atributa terminalnih simbola je `YYSTYPE yyval`. Funkcija koja se poziva ukoliko se u procesu analize pronađe greska koja nije predviđena, ili dođe do bilo kakve druge greške u radu analizatora je funkcija

```
void yyerror(char* s);
```

gde je `s` string koji signalizira do kakve je greške došlo.

#### Primer 10.13 Yacc specifikacija za generisanje sintaksnog analizatora aritmetičkih izraza

Neka su aritmetički izrazi definisani gramatikom

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \text{num.}$$

gde token **num** predstavlja jednocifrene celobrojne konstante. Potreban leksički analizator je implementiran u okviru Yacc specifikacije.

```
%{
#include <ctype.h>
%}
```

```

//definicija tokena
%token NUM

//definicija prioriteta i asocijativnosti operatora
%left '+' '-'
%left '*' '/'

%%

//pravila
E1 : E '\n'    {puts("prepoznat aritmeticki izraz");};
E  : E '+' E    {;};
    | E '-' E    {;};
    | E '*' E    {;};
    | E '/' E    {;};
    | '(' E ')'  {;};
    | NUM        {;};
    ;
%%
int yylex()
{
    int c;
    while ((c = getchar()) == ' ');
    if (isdigit(c))
        return NUM;
    return c;
}

void main()
{
    yyparse();
}

```

### 10.2.9 Oporavak od grešaka

Sintaksni analizator generisan na osnovu specifikacije iz primera 10.13 će samo utvrditi da li je izraz korektno zapisan ili ne. U trenutku kada se u kodu pojavi prva greška, analiza će se prekinuti, biće pozvana funkcija `yyerror` koja će prikazati poruku **Syntax error**. Nedostatak ovog analizatora je što neće lokalizovati grešku, tj. neće nam dati informacije gde se ona nalazi, ni o kakvoj se grešci radi; a neće ni otkriti sve greške u kodu. Cilj analizatora je da prijavi sve greške i da korisniku ukaže gde se nalaze i kako da ih ispravi. Postupak koji nam omogućava da i kada se greška otkrije, analizator nastavi svoj rad se popularno

naziva “oporavak od greške”.

Pretpostavimo da je u nekom izrazu (koji se prepoznaje gramatikom koja je definisana u primeru 10.11) izostavljena zatvorena zagrada. Prilikom prepoznavanja izraza sa zagradama koristilo bi se pravilo:

$$E : '(' E ')'$$

Oporavak od ove greške bi podrazumevao da, kada zagrada nedostaje, prijavimo grešku, a da se analiza ipak nastavi. To znači da treba uvesti fiktivnu smenu po kojoj će se vršiti redukcija ako zagrada nedostaje. U takvim smenama koristi se ključna reč *error* koja zamenjuje skup simbola proizvoljne dužine za koji akcija u određenom stanju nije definisana. U ovom slučaju, ako zagrada nedostaje, vršiće se redukcija po smeni:

$$E : '(' E \text{ error } /* \text{ prijava greske } */ ;$$

Pri definisanju ovih fiktivnih smena treba voditi računa da se što je moguće više spreči višestruko prijavljivanje iste greške (što je nemoguće potpuno izbeći) i da se ne desi da se u istom stanju pri pojavi istog simbola na ulazu može vršiti redukcija po većem broju smena. U takvim slučajevima yacc će prijaviti takozvane *reduce/reduce* konflikte (koje on ne može da prevaziđe) i kod neće generisati. Ako se prilikom kreiranja sintaksne tabele pojave *shift/reduce* konflikti (njihov uzrok je objašnjen u poglavlju o LR sintaknim analizatorima), prijavitiće se semo upozorenje, a u sintaksoj tabeli biće upisana *shift* akcija.

**Primer 10.14** Yacc specifikacija za generisanje sintaksnog analizatora koji podržava oporavak od grešaka

Kreiraćemo Yacc specifikaciju za generisanje sintaksnog analizatora jezika koji je definisan gramatikom navedenom u primeru 10.11. Uz sintaksne greške, analizator će prijavljivati i jednostavnije semantičke, npr. korišćenje simbola kojem nije dodeljena vrednost i sl.

Struktru za pamćenje podataka o simboličkim imenima proširićemo komponentama potrebnim za semantičku analizu.

```
// Fajl Symbol.h
struct Symbol
{
    char name[32];
    // linija u kojoj je symbol poslednji put definisan
    int lastDef;
    // linija u kojoj je symbol poslednji put korišćen
    int lastUse;
};
```

```

// Fajl Program.y
%{
#include "Symbol.h"
#include <stdio.h>
extern int    yylex();
extern int    lineNo;
extern int    symbolNo;
extern struct Symbol symbolTable[];
extern FILE*  yyin;
extern char*  yytext;
%}

%union
{
    int index;
    float value;
}

%token END READ WRITE
%token<index> ID
%token<value> CONST

%%
Program : NizNar END { printf("analiza je završena\n"); };
NizNar  : NizNar ';' Naredba { ; }
        | NizNar ';' error
          { printf("linija %d: očekuje se naredba nakon ;",
                    lineNo); }
        | NizNar error
          { printf("linija %d: očekuje se ; ili end",
                    lineNo); }
        | Naredba { ; }
        | error
          { printf("linija %d: neočekivani simbol %s\n",
                    lineNo, yytext); }
        ;

Naredba : Ulaz    { ; }
        | Izlaz   { ; }
        | Dodela  { ; }
        ;

Ulaz : READ '(' ID ')',
      { symbolTable[$3].posDef = lineNo; }
      | READ '(' ID error

```



```

    {
        printf("linija %d: nedostaje )\n", lineNo );
        symbolTable[$3].lastDef = lineNo;
    }
| READ '(' error
  { printf("linija %d: nedostaje identifikator\n", lineNo); }
| READ error
  { printf("linija %d: nedostaje (\n", lineNo); }
;

Izlaz : WRITE '(' Izraz ')' { ; }
| WRITE '(' Izraz error
  { printf("linija %d: nedostaje )\n", lineNo); }
| WRITE '(' error
  { printf("linija %d: nekorektan izraz\n", lineNo); }
| WRITE error
  { printf("linija %d: nedostaje (\n", lineNo); }
;

Dodela : ID '=' Izraz { symbolTable[$1].lastDef = lineNo; }
| ID '=' error
  {
      printf("linija %d: nekorektan je izraz\n", lineNo);
      symbolTable[$1].lastDef = lineNo;
  }
| ID error
  {
      printf("linija %d: ocekuje se =\n", lineNo);
      symbolTable[$1].lastDef = lineNo;
  }
;

Izraz : Izraz '+' Proizvod { ; }
| Izraz '+' error
  { printf("linija %d: nekorektan izraz\n", lineNo); }
| Proizvod { ; }
;

Proizvod : Proizvod '*' Faktor { ; }
| Proizvod '*' error
  { printf("linija %d: nekorektan izraz\n", lineNo); }
| Faktor
;

Faktor : '(' Izraz ')' { ; }

```

```

| '(' Izraz error
{ printf("linija %d: nedostaje )\n", lineNo); }
| '(' error
{ printf("linija %d: nekorektan izraz\n", lineNo); }
| ID
{
    if ( symbolTable[$1].lastDef == 0 )
        printf("linija %d: %s se koristi pre inicijalizacije\n",
            lineNo, yytext );
    tabelaSimbola[$1].posKor = brLinije;
}
| CONST {;}
;

%%
#include "lex.yy.c"

void main()
{
    int i;
    lineNo = 1;
    symbolNo = 0;
    yyin = fopen("ulaz.ml", "r");
    yyparse();
    for (i = 0; i < symbolNo; i++)
        if (symbolTable[i].lastDef > symbolTable[i].lastUse)
        {
            printf("upozorenje: vrednost %s postavljena u liniji %d",
                symbolTable[i].name, symbolTable[i].lastDef);
            printf("nije nigde koriscena.\n");
        }
}

```

Za generisanje potrebnog leksičkog analizatora se može koristiti Lex specifikacija iz primera 10.11 uz sledeće promene.

- Umesto fajla `Tokens.h` treba uključiti fajl `y.tab.h`.
- Promenljivu `attribute` zameniti promenljivom `yylval` tipa `YYSTYPE`.

### 10.2.10 Generisanje interpretatora pomoću Yacc-a

Izvršenje programa vrši se po njegovom sintaksnom stablu i to tako što se najpre izvršavaju operacije bliže listovima, da bi se tek na kraju izvršila operacija koja se nalazi u korenu stabla. S obzirom da Yacc generiše sintakсни analizator koji sintakсно stablo generiše u tom istom redosledu (od listova ka korenu),

paralelno sa procesom generisanja sintaksnog stabla programa, on može i da se izvršava. Ako smenama gramatike pridružimo akciju koja izvršava naredbu koja se smenom definiše, Yacc će generisati interpretator jezika koji je definisan odgovarajućom gramatikom.

**Primer 10.15** Yacc specifikacija za generisanje interpretatora aritmetičkih izraza

Neka su aritmetički izrazi definisani gramatikom koja je data u primeru 10.13.

```
%{
#include <ctype.h>
%}

//definicija tokena
%token NUM

//definicija prioriteta i asocijativnosti operatora
%left '+' '-'
%left '*' '/'

%%

//pravila
E1 : E '\n'      { printf("%d\n", $1); };
E  : E '+' E      { $$ = $1 + $3; }
    | E '-' E      { $$ = $1 - $3; }
    | E '*' E      { $$ = $1 * $3; }
    | E '/' E      { $$ = $1 / $3; }
    | '(' E ')'    { $$ = $2; }
    | NUM          { $$ = $1; }
    ;

%%

int yylex()
{
    int c;
    while ((c=getchar()) == ' ');
    if (isdigit(c))
    {
        yylval = c - '0';
        return NUM;
    }
    return c;
}
```

```

}

void main()
{
    yyparse();
}

```

**Primer 10.16** Yacc specifikacija za generisanje interpretatora jezika koji je definisan u primeru 10.11

Strukturi Symbol ćemo dodati i član koji čuva vrednost simbola.

```

// Fajl Symbol.h
struct Symbol
{
    char name[32];
    // linija u kojoj je symbol poslednji put definisan
    int lastDef;
    // linija u kojoj je symbol poslednji put korišćen
    int lastUse;
    // vrednost simbola
    float value;
};

// Fajl Program.y
%{
#include "Symbol.h"
#include <stdio.h>
extern int yylex();
extern int lineNo;
extern int symbolNo;
extern struct Symbol symbolTable[];
extern FILE* yyin;
extern char* yytext;
%}

%union
{
    int index;
    float value;
}

%token END READ WRITE
%token<rbr> ID

```

```

%token<vrednost> CONST
%type<vrednost> Izraz Proizvod Faktor

%%

Program : NizNar END { printf("analiza je završena\n"); };

NizNar : NizNar ';' Naredba { ; }
      | Naredba { ; }
      ;

Naredba : Ulaz { ; }
      | Izlaz { ; }
      | Dodela { ; }
      ;

Ulaz : READ '(' ID ')'
     {
       scanf("%f", &(symbolTable[$3].value));
       symbolTable[$3].lastDef = lineNo;
     }
     ;

Izlaz : WRITE '(' Izraz ')'
     {
       printf("%f\n", $3);
     }
     ;

Dodela : ID '=' Izraz
     {
       symbolTable[$1].lastDef = lineNo;
       symbolTable[$1].value = $3;
     }
     ;

Izraz : Izraz '+' Proizvod { $$ = $1 + $3; }
      | Proizvod { $$ = $1; }
      ;

Proizvod : Proizvod '*' Faktor { $$ = $1*$3; }
         | Faktor { $$ = $1; }
         ;

Faktor : '(' Izraz ')' { $$ = $2; }

```

```

| ID
{
    if (symbolTable[$1].lastDef == 0)
    {
        printf("linija %d: promenljiva %s se",
            lineNo, yytext );
        printf(" koristi pre nego što");
        printf(" je inicijalizovana\n");
        return 1;
    }
    $$ = symbolTable[$1].value;
    tabelaSimbola[$1].posKor = brLinije;
}
| CONST { $$ = $1; }
;

%%
void main()
{
    lineNo = 1;
    symbolNo = 0;
    yyin = fopen("ulaz.ml", "r");
    yyparse();
}

```

## 10.3 JFlex

Na Princeton univerzitetu 1996. godine razvijen je alat JLex, koji generiše izvorni kod leksičkih analizatora u programskom jeziku Java. JFlex je modifikacija i nadgradnja funkcionalnosti JLex-a. Ovo je besplatan i open source alat.

JFlex specifikacija, takođe, sadrži tri osnovna dela, ali su drugačije raspoređeni nego u Lex specifikaciji. Dakle, format JFlex specifikacije je:

```

<korisnički_kod>
%%
<opcije_i_direktive>
%%
<pravila>

```

Korisnički kod sadrži java naredbe koje se prepisuju na početak generisanog fajla. Tu se uglavnom pišu `import` i `package` naredbe. Opcije i deklaracije sadrže:

- opcije za podešavanje generisanog leksera:

- JFlex direktive i
- Java kod koji se uključuje u različite delove leksičkog analizatora (Java klase koja se generiše);
- deklaracije posebnih početnih stanja
- definicije makroa.

### 10.3.1 Opcije kojima se podešavaju parametri generisanog leksičkog analizatora

JFlex direktive koje se najčešće koriste su sledeće.

- Direktive kojima se podešavaju parametri generisane klase – leksičkog analizatora:
  - `%class <ImeKlase>` definiše ime generisane klase. Ukoliko ova opcija nije navedena, ime generisane klase je `Yylex`.
  - `%implements <intrefejs1>[,<interfejs2>][,...]` – ukoliko je ova direktiva navedena, generisana klasa implementira navedene interfejse.
  - `%extends <ImeKlase>` – ukoliko je ova direktiva navedena, generisana klasa nasleđuje navedenu klasu.
- Direktive kojima se podešavaju parametri funkcije koja vrši izdvajanje sledeće reči iz ulaznog teksta:
  - `%function <ImeFunkcije>` definiše ime generisane funkcije. Ukoliko ova direktiva nije navedena, ime generisane funkcije je `yylex()`.
  - `%int` ili `%integer` postavlja povratni tip funkcije koja vrši leksičku analizu na tip `int`.
  - `%type <ImeTipa>` postavlja povratni tip funkcije koja vrši leksičku analizu na navedeni tip.

Ukoliko ni jedna od prethodne dve direktive nije navedena, podrazumevani povratni tip leksičkog analizatora je `Ytoken`.
- Direktive kojima se uključuju brojači linija i karaktera:
  - `%line` uključuje brojač linija u izvornom fajlu. Generiše se promenljiva `yyline` (tipa `int`) koja pamti redni broj tekuće linije (brojanje počinje od 0).
  - `%char` uključuje brojač karaktera u izvornom fajlu. Generiše se promenljiva `yychar` (tipa `int`) koja pamti redni broj prvog karaktera izdvojene reči (brojanje počinje od 0).

- `%column` uključuje brojač karaktera u tekućoj liniji. Generiše se promenljiva `yycolumn` (tipa `int`) koja pamti redni broj prvog karaktera izdvojene reči u tekućoj liniji (brojanje počinje od 0).
- Direktive kojima se nalaže generisanje samostalne aplikacije (tj. generisanje `main` metoda):
  - `%debug` generiše se `main` metod koji poziva leksički analizator sve dok se ne dođe do kraja ulaznog fajla. Na standardni izlaz (tj. u Java konzoli) ispisuju se svi izdvojeni tokeni kao i kod akcije koja je tom prilikom izvršena.
  - `%standalone` generiše se `main` metod koji poziva leksički analizator sve dok se ne dođe do kraja ulaznog fajla. Ispravno prepoznati tokeni se ignorišu, a na standardni izlaz se ispisuju neprepoznati delovi ulaznog fajla.

Delovi Java koda koji se ugrađuju u različite delove generisane klase se definišu na sledeći način.

- Sledeći deo koda se ugrađuje u generisanu klasu. Tu se definišu dodatni atributi i metodi klase.

```
%{
  <code>
%}
```

- Sledeći deo koda se ugrađuje u konstruktor generisane klase.

```
%init{
  <code>
%init}
```

- Sledeći deo koda se ugrađuje u generisanu funkciju koja iz funkcije za leksičku analizu poziva u prenutku kada se prepozna `EOF`. Ovaj deo koda treba da vrati vrednost koja signalizira da se došlo do kraja koda koji se analizira. Ukoliko ovaj deo nije naveden, a povratni tip funkcije koja vrši leksičku analizu je `int`, vraća se vrednost konstante `YYEOF` (javna statička konstanta generisane klase). Ukoliko je povratni tip neki korisnički definisani tip, funkcija će vratiti konstantu `null`.

```
%eofval{
  <code>
%eofval}
```



### 10.3.2 Deklaracije posebnih početnih stanja

Deklaracije posebnih početnih stanja u JFlex specifikaciji imaju istu sintaksu kao i u Lex specifikaciji. Jedino se kao identifikator podrazumevanog početnog stanja koristi konstanta YYINITIAL.

### 10.3.3 Definicije makroa

U JFlex specifikaciji se makroi definišu na sledeći način:

`<ImeMakroa> = <RegularniIzraz>`

Za razliku od lex specifikacije, ovde se nakon imena makroa navodi znak =.

### 10.3.4 Pravila

I pravila u JFlex specifikaciji se definišu na isti način kao i u Lex-u. Jedina razlika u tome je što akcija mora da bude navedena kao blok naredbi (između zagrada `{}` i kada ona sadrži samo jednu Java naredbu).

### 10.3.5 Korišćenje JFlex-a

JFlex specifikacija se zapisuje sa ekstenzijom `.flex`. JFlex kompilator se može pozvati iz komandne linije na sledeći način:

`jflex <ime_specifikacije>`

JFlex generiše samo jednu java klasu i, ukoliko direktivama nije drugačije naloženo, ona je definisana na sledeći način:

- ime klase: `Ylex`,
- konstruktor: `public Ylex(System.io.Reader r),`
- funkcija za leksičku analizu: `public Ytype ylex(),`
- funkcija koja menja početno stanje analize: `void yybegin(int state),`
- funkcija koja menja početno stanje na podrazumevano: `void yybegin(),`
- funkcija koja vraća trenutno početno stanje: `int yystate(),`
- funkcija koja vraća izdvojenu reč: `String yytext(),`
- redni broj tekuće linije: `int yyline,`
- redni broj tekućeg karaktera u ulaznom tekstu: `int yychar,`
- redni broj tekućeg karaktera u liniji: `int yycolumn.`

### 10.3.6 Primer JFlex specifikacije

Kreiraćemo JFlex specifikaciju za generisanje leksičkog analizatora za isti jezik za koji je kreirana Lex specifikacija u primeru 10.11.

**Prvi korak.** Kreiramo klasu koja sadrži definicije tokena.

**Primer 10.17** Java klasa koja sadrži definicije tokena

```
// sym.java
public class sym
{
    public static int EOF;      // kraj ulaznog teksta
    public static int END;      // ključna reč end
    public static int READ;     // ključna reč read
    public static int WRITE;    // ključna reč write
    public static int PLUS;     // operator +
    public static int MUL;      // operator *
    public static int ASSIGN;   // separator =
    public static int LPAR;     // separator (
    public static int RPAR;     // separator )
    public static int NEWLINE;  // separator '\n'
    public static int ID;       // identifikator
    public static int CONST;    // konstanta
}
```

**Drugi kroak.** Kreiramo klasu čije će objekte vraćati funkcija za leksičku analizu. Ova klasa obično sadrži:

- izdvojenu reč,
- njeno značenje (celobrojni identifikator značenja),
- liniju koda iz koje je reč izdvojena,
- početnu poziciju u liniji od koje je reč izdvojena,
- poziciju u liniji na kojoj se izdvojena reč završava,
- dodatni atribut (ukoliko je potrebno).

**Primer 10.18** Primer Java klase koja predstavlja rezultat leksičkog analizatora

```
// Ytoken.java
```

```

class Ytoken {
    public int _token;
    public String _text;
    public int _line;
    public int _charBegin;
    public int _charEnd;
    public Object _value;

    Ytoken (int token, String text, int line,
            int charBegin, int charEnd, Object value) {
        _token = token;
        _text = text;
        _line = line;
        _charBegin = charBegin;
        _charEnd = charEnd;
        _value = value;
    }

    public String toString() {
        return "Reč: " + _text +
            "\ntoken: " + _token +
            "\nlinija: " + _line +
            "\npocetak: " + _charBegin +
            "\nkraj: " + _charEnd +
            "\natribut: " + _value;
    }
}

```

**Treći korak.** Kreiramo JFlex specifikaciju.

#### **Primer 10.19** JFlex specifikacija

```

// import section
package compiler;

%%
// declaration section
%class MyScanner
%function scan
%line
%column

%debug

```

```

// novi član generisane klase:
// kreira objekat koji funkcija scan vraća
%{
private Ytoken createToken( int tokenCode, Object value )
{
    return new Ytoken(tokenCode, yytext(), yyline, yycolumn,
                      yycolumn+yylength(), value);
}
%}

%eofval{
return createToken(sym.EOF, null);
%eofval}

//definicije makroa
cifra = [0-9]
Slovo = [a-zA-Z]

%%
// pravila
// ključne reci
"end"   { return createToken(sym.END, null); }
"read"  { return createToken(sym.READ, null); }
"write" { return createToken(sym.WRITE, null); }

// simbolička imena
{slovo}|_({slovo}|{cifra}|_)*
{
    return createToken(sym.ID, yytext());
}

// konstante
{cifra}+(\.{cifra})*?([Ee][+-]?{cifra}+)?
{
    return createToken(sym.CONST, new Float(yytext()));
}

// ignorisanje belih simbola
[\t ]    { ; }

// operatori
\+       { return createToken(sym.PLUS, null); }
\*       { return createToken(sym.MUL, null); }

```

```
// separatori
= { return createToken(sym.ASSIGN, null); }
\ ( { return createToken(sym.LPAR, null); }
/ ) { return createToken(sym.RPAR, null); }
\n { return createToken(sym.NEWLINE, null) };

// ukoliko se pojavi bilo koji drugi simbol, prijaviti grešku
. { System.out.println("Unknown symbol: " + yytext()); }
```

## 10.4 Cup – generator sintaksnih analizatora

Cup (*Construct Useful Parser*) je alat koji generiše LR sintaksne analizatore u Javi i sam je pisan u Javi. I ovo je besplatan alat otvorenog koda. Uglavnom radi u sprezi sa JFlex-om.

Sintakсни analizator koji se generiše pomoću Cup-a ima dva dela:

- fiksni deo koji je definisan u paketu `java_cup.runtime` koji je nezavistan od gramatike jezika za koji se prevodilac piše i
- promenljivi deo koji je zavistan od gramatike jezika koji se generiše na osnovu Cup specifikacije.

Paket `java_cup.runtime` sadrži sledeće.

- Klasu `Symbol` koja služi za predstavljanje podataka o svim simbolima gramatike. Atributi ove klase koji se koriste u specifikaciji su:

- `public int sym;` – celobrojni identifikator simbola,
- `public int left, right;` – početna i završna pozicija u ulaznom tekstu,
- `public Object value;` – dodatni atribut simbola.

Postoje i atributi klase `Symbol` koje postavlja i koristi funkcija za parsiranje i oni se ne smeju u kodu menjati. Takvi atributi su:

- `public int parse_state;` – stanje LR analizatora, ovaj atribut je postavljen samo ukoliko se simbol nalazi u radnom magacinu LR analizatora. Ovaj atribut je uveden jer se radnom magacinu ne pamte naizmenično simboli i stanja, već samo simboli pa se njima pridružuju stanja analize.
- `public boolean used_by_parser;` – indikator koji kaže da je simbol generisao parser usled neke greške u ulaznom kodu.
- Interfejs `Scanner` – klasa koja služi za leksičku analizu mora da implementira ovaj interfejs. U interfejsu je definisana funkcija za leksičku analizu (`next_token()`) koja vraća rezultat tipa `Symbol`.

- Klasu `lr_parser` čije su najbitnije funkcije:
  - `public Symbol parse()` – funkcija za LR sintaksnu analizu koja se bazira na korišćenju sintaksnih tabela koje su zavisne od jezika pa se genrišu i izvedenoj klasi. Rezultat funkcije `parse()` je tipa `Symbol` i ako je analiza uspešno završena, on će predstavljati startni simbol gramatike.
  - `public Symbol debug_parse()` – radi isto što i `parse` funkcija i uz to u tok `System.err` upisuje poruke u vezi s debugiranjem.

Generisani deo aplikacije (zavistan od jezika) sadrži:

- klasu `sym` koja sadrži definicije tipova terminalnih simbola (celobrojne identifikatore tokena);
- klasu `parser` (čije ime može biti i promenjeno) koja je izvedena iz klase `lr_parser` i koja sadrži funkcije za inicijalizaciju potrebnih tabela:
  - `production_tab` – tabela koja za svaku smenu gramatike pamti identifikacioni broj simbola sa leve strane smene i dužinu desne strane,
  - `action_tab` – sadrži deo za akcije LR sintaksne tabele i
  - `reduce_tab` – sadrži deo za prelaze nakon redukcija u LR sintakсноj tabeli.

Privatnu klasu `CUP$action` koja sadrži definicije akcija (akcije koje korisnik definiše u Cup specifikaciji). Najbitnija funkcija ove klase je `CUP$do_action()`.

Na osnovu strukture koda koji Cup generiše, može se zaključiti da leksički analizator koji se koristi u sintaksnom analizatoru generisanom pomoću Cup-a treba da zadovolji sledeće uslove:

- implementira interfejs `Scanner`,
- funkcija za leksičku analizu je `next_token()` i
- rezultat leksičke analize je objekat klase `Symbol`,

Kada se dođe do kraja ulaznog teksata, analizator treba da vrati simbol sa identifikatorom `<CUPSYM>.EOF`.

To znači da, ukoliko potreban leksički analizator generišemo pomoću JFlexa, u JFlex specifikaciji treba da stoje sledeće direktive:

```
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
%eofval {
    return new java_cup.runtime.Symbol(<CUPSYM>.EOF);
%eofval }
```

Umesto svih ovih direktiva, dovoljno je navesti direktivu `%cup` koja kaže da generisani kod treba da bude kompatibilan sa kodom koji generiše Cup.

### 10.4.1 Cup specifikacija

Cup specifikacija po svojoj strukturi malo odstupa od strukture koju imaju Lex, Yacc i JFlex specifikacija. Delovi cup specifikacije se međusobno ne odvajaju simbolom `%%`, i ključne reči ne počinju simbolom `%`. Delovi Cup specifikacije su:

- `import` sekcija,
- korisnički kod,
- liste terminalnih i neterminalnih simbola,
- prioriteti i asocijativnosti operatora i
- opis gramatike.

Ovi delovi u specifikaciji mora da budu napisani u redosledu kojim su ovde navedeni.

#### Import sekcija

`Import` sekcija sadrži `import` i `package` naredbe koje se prepisuju na početak fajla sa kodom analizatora i njihova sintaksan ispravnost se ne proverava.

Korisnički kod sadrži delove java koda koji se ugrađuju u različite tačke generisanih klasa.

Novi članovi (atributi i metode) generisane klase `CUP$action` se navode u konstrukciji:

```
action code {: /* java code */ :}
```

Metode definisane u ovom delu se mogu pozivati iz akcija koje se pridružuju smenama u delu za definisanje gramatike.

Novi članovi (atributi i metode) generisane klase `parser` se navode u konstrukciji:

```
parser code {: /* java code */ :}
```

Deo koda koji se izvršava pre izdvajanja prve reči iz ulaznog fajla (tj. pre prvog poziva `scanner-a`) navodi se u konstrukciji:

```
init with {: /* java code */ :}
```

Deo koda koji definiše kako parser poziva leksički analizator, tj. kako traži izdvajanje sledeće reči, navodi se u konstrukciji:

```
scan with {: /* java code */ :}
```

### Liste terminalnih i netermianlnih simbola

Liste terminalnih i netermianlnih simbola sadrže definicije svih simbola koje gramatika sadrži. Terminalni simboli se uvode deklaracijom:

```
terminal [<ImeKlase>] <ImeSimbola1>, <ImeSimbola2>, ... ;
```

<ImeKlase> je tip člana value u objektu klase `Symbol` koji odgovara datom terminalnom simbolu. Ukoliko se ovaj tip navede, atribut simbola se koristi bez eksplicitnog kastovanja u svoj tip.

Neterminalni simboli gramtike se definišu deklaracijom:

```
non terminal [<ImeKlase>] <ImeSimbola1>, <ImeSimbola2>, ... ;
```

### Asocijativnost operatora

Asocijativnosti operatora se definišu deklaracijama

```
precedence left <ImeSimbola1>, <ImeSimbola2>, ... ;
precedence right <ImeSimbola1>, <ImeSimbola2>, ... ;
precedence nonassoc <ImeSimbola1>, <ImeSimbola2>, ... ;
```

### Prioritet operatora

Prioritet operatora je, kao i u Yacc specifikaciji, određen redosledom navođenja njihovih asocijativnosti.

#### Primer 10.20 Definicije prioriteta aritmetičkih operatora u Cup specifikaciji

Neka su aritmetički izrazi definisani gramatikom

$$\begin{aligned} \text{Izraz} \rightarrow & \text{Izraz} + \text{Izraz} \\ & | \text{Izraz} - \text{Izraz} \\ & | \text{Izraz} * \text{Izraz} \\ & | \text{Izraz} / \text{Izraz} \\ & | -\text{Izraz} \\ & | \text{CONST} \end{aligned}$$

Pošto je ova gramatika nejednoznačna, navođenje asocijativnosti i prioriteta operatora u cup specifikaciji je u ovom slučaju obavezno.

Definicije prioriteta i asocijativnosti operatora sadržale bi sledeće Cup opise:

```
precedence left PLUS, MINUS;
```



```
precedence left PUTA, PODELJENO;
precedence right UMINUS;
```

Kao što se vidi, za isti simbol (–) definisana su dva različita prioriteta (MINUS i UMINUS). Koji će se prioritet primeniti zavisice od konteksta u kojem je simbol naveden.

### Opis gramatike

Opis gramatike sadrži (opciono) definiciju startnog simbola gramatike i (obavezno) definicije njenih smena. Format definicija startnog simbola gramatike je:

```
start with <ImeSimbola>;
```

Ukoliko je definicija startnog simbola gramatike izostavljena, kao i u Yacc specifikaciji, podrazumeva se da je startni simbol gramatike simbol koji se nalazi na levoj strani prve navedene smene.

Smene gramatike se u Cup specifikaciji definišu parvilima koja se navode u formatu:

```
<LevaStrana> ::= <DesnaStrana> [ {<Akcija> :} ];
```

Akcija je java kod koji se izvršava u trenutku kada se vrši redukcija po navedenoj smeni.

Više smena koje na levoj strani imaju isti neterminalni simbol se mogu skraćeno navoditi na isti način kao u Yacc specifikaciji.

Za oporavak od grešaka se takođe koristi ista notacija kao u Yacc specifikaciji (uvode se pomoćne smene sa **error** simbolom).

### 10.4.2 Uvođenje referenci na atribut simbola u smenama

Ukoliko u akcijama treba koristiti dodatne atribut simbola koji se nalaze na desnoj strani smene, uvode se reference na te atribut na sledeći način:

```
<ImeSimbola> : <Ime>
```

<Ime> je u ovom slučaju referenca na član value objekta klase `Symbol`.

value atributu simbola koji se nalazi na levoj strani smene pristupa se korišćenjem reference `RESULT`.

**Primer 10.21** Pravila Cup specifikacije za generisanje interpretatora aritme-

tičkih izraza

Pretpostavimo da su aritmetički izrazi definisani gramatikom navednom u primeru 10.20.

```
Izraz ::= Izraz:i1 PLUS Izraz:i2
      {
        RESULT = new Integer(i1.intValue() + i2.intValue());
      }
| Izraz:i1 MINUS Izraz:i2
      {
        RESULT = new Integer(i1.intValue() - i2.intValue());
      }
| Izraz:i1 PUTA Izraz:i2
      {
        RESULT = new Integer(i1.intValue() * i2.intValue());
      }
| Izraz:i1 PODELJENO Izraz:i2
      {
        RESULT = new Integer(i1.intValue() / i2.intValue());
      }
| UMINUS Izraz:i
      {
        RESULT = new Integer(- i.intValue());
      }
      %prec UMINUS // smena Izraz --> - Izraz
| CONST:c
      {
        RESULT = c;
      }
;
```

Naglašeni deo pravila koje odgovara smeni  $Izraz \rightarrow -Izraz$  označava da se u tom pravilu operator MINUS koristi sa prioritetom UMINUS. Dakle, leksički analizator će i u jednom i u drugom slučaju da vrati simbol MINUS, ali će se pri redukciji po ovoj smeni za njega koristiti drugi prioritet (takozvani kontekstni prioritet).

### 10.4.3 Prevođenje Cup specifikacije

Pošto je Cup implementiran u Javi, prevođenje cup specifikacije se vrši pomoću java interpretatora na sledeći način:

```
java java_cup.Main options ime_specifikacije
```

Ime specifikacije mora da ima ekstenziju .cup.

Opcijama se može prilagoditi izgled generisanog koda. Opcije koje se najčešće koriste su:

- `-package ime_paketa` definiše ime paketa kojima će pripadati generisane klase.
- `-parser ime_klase` definiše ime klase koja će sadržati `parse()` metod. Ukoliko se ova opcija ne navede, ime generisane klase je `parser`.
- `-symbols` definiše ime klase u kojoj su kao statičke konstante definisani identifikatori tokena. Ukoliko se ova opcija ne navede, ime generisane klase je `sym`.
- `-nonterminals` zahteva da u klasi sa identifikatorima tokena budu generisani i identifikatori neterminalnih simbola. Poznavanje ovih identifikatora olakšava praćenje rada analizatora prilikom debugiranja.

#### 10.4.4 Korišćenje generisanog analizatora

Da bi se izvršila analiza pomoću generisanog analizatora, treba kreirati objekat klase `parser` i pozvati funkciju `parse()`. Konstruktor klase `parser` ima argument tipa `Scanner`, što znači da to treba da bude objekat klase koja implementira ovaj interfejs, tj. koja sadrži funkciju `next_token()`.

##### Primer 10.22 Korišćenje analizatora koji je generisan pomoću Cup-a

```
public static void main(String[] args) {  
    try {  
        FileReader file = new FileReader(args[0]);  
        java_cup.runtime.Scanner myScanner = MyScanner(file);  
        parser myParser = parser(myScanner);  
        myParser.parse();  
    }  
    catch(Exception e) {  
        System.out.println(e);  
    }  
}
```

Ukoliko je analizirani kod korektno zapisan, funkcija `parse()` će izvršiti redukciju tog koda na startni simbol gramatike i vratiti taj simbol kao rezultat. Ukoliko se u kodu pojavi bilo kakva sintaksna greska, poziva se funkcija

```
void syntax_error(Symbol currentToken),
```

a zatim se pokušava da se izvrši oporavak od grške. Ukoliko oporavak ne uspe, poziva se funkcija:

```
void unrecovered_syntax_error(Symbol currentToken).
```

Funkcije `syntax_error` i `unrecovered_syntax_error` su definisane u klasi `lr_parser` i one samo pozivaju odgovarajuće metode za prikaz poruke o grešci.

Metode za prikaz poruka o greškama su sledeće.

- `public void report_error(String message, Object info)` – Ova funkcija služi za prikaz poruke o sintaksoj grešci. U postojećoj implementaciji ona samo prikazuje datu poruku, dok se drugi argument prosto ignoriše. Drugi argument služi ako u predefinisanoj funkciji želimo da poruku o grešci “obogatimo” dodatnim informacijama.
- `public void report_fatal_error(String message, Object info)` – Ovu metodu treba pozvati za prikaz poruke o grešci kada oporavak nije moguć. U postojećoj implementaciji ona poziva metodu `report_error()`, poziva funkciju za prekid parsiranja `done_parsing()` i na kraju prijavljuje izuzetak.

Kao što se vidi, implementacije svih ovih funkcija su jako “siromašne” pa je generalna preporuka da se sve one predefinišu u izvedenoj klasi – klasi koja se generiše. To znači da nove implementacije ovih funkcija treba pisati u delu

```
parser code {: ... :}.
```

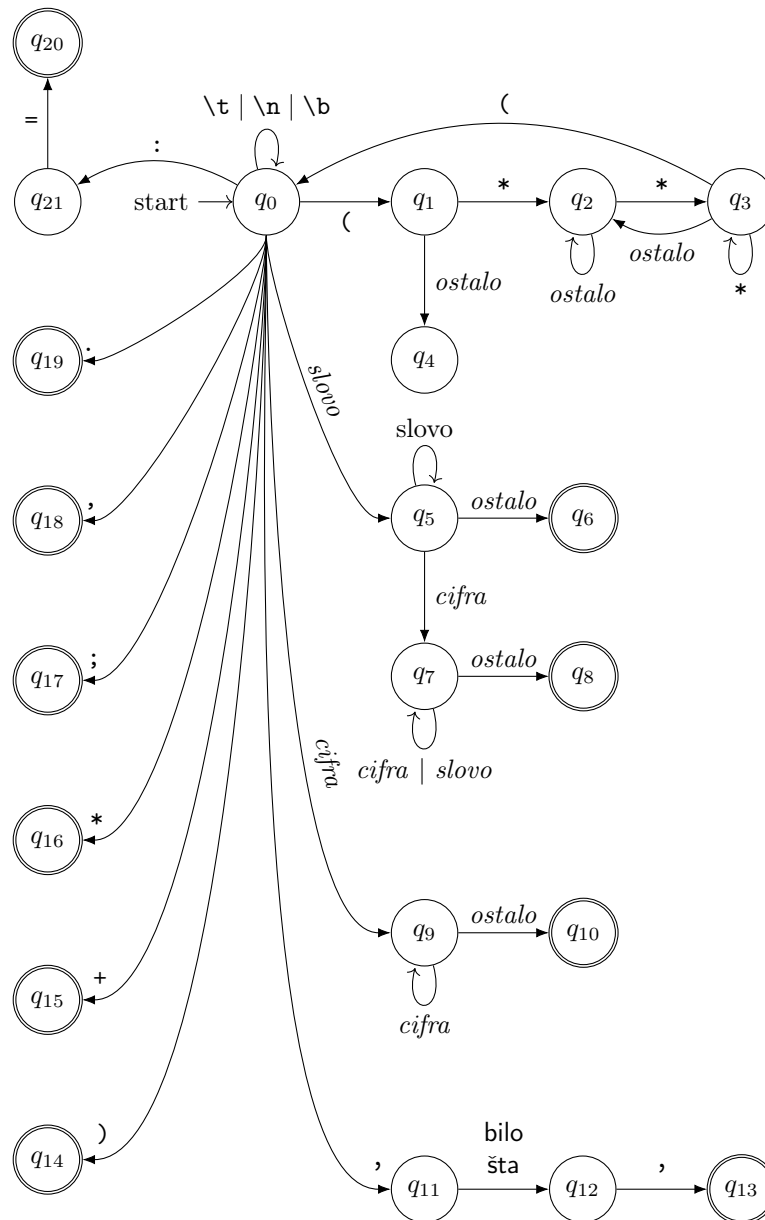
## 10.5 Pitanja

1. Šta je Lex/JFlex?
2. Objasniti strukturu Lex/Jflex specifikacije.
3. Šta znače specijalna početna stanja u Lex/Jflex specifikaciji. Kako se ona definišu? Kako se koriste?
4. Šta su makroi u Lex/Jflex specifikaciji. Kako se definišu? Kako se koriste?
5. Objasniti kako se koriste dodatni atributi simbola u Lex/Jflex specifikaciji.
6. Šta je Yacc/Cup?
7. Objasniti strukturu Yacc/Cup specifikacije.
8. Objasniti kako se koriste dodatni atributi simbola u Yacc/Cup specifikaciji.
9. Objasniti mehanizam za oporavak od grešaka u Yacc/Cup specifikaciji.
10. Objasniti statički deo koda sintaksnog analizatora generisanog pomoću Cupa.

11. Objasniti deo koda sintaksnog analizatora koji se generiše na osnovu Cup specifikacije.

## 10.6 Zadaci

1. Napisati Lex/Jflex specifikaciju za analizator iz zadatka 2 u poglavlju 4.
2. Kreirati Lex/Jflex specifikaciju za generisanje leksičkog analizatora jezika mPascal, koji je predstavljen grafom na slici 10.1.



SLIKA 10.1: Graf automata leksičkog analizatora jezika  $\mu$ Pascal

## Glava 11

# Atributne gramatike

Opis naredbi jezika preko beskonteksnih gramatika obično nije dovoljan za uspešno prevođenje jezika. Postoji mnogo problema koji se ne mogu predstaviti formalnim opisom jezika. Zbog toga se kod praktičnih rešenja pravilima kojima se definiše jezik pridružuje skup dodatnih informacija tako što se simbolima gramatike dodaju određeni atributi. Vrednosti ovih atributa se izračunavaju preko semantičkih rutina koje se pridružuju pravilima gramatike. Koriste se dva načina da se semantičke rutine pridruže pravilima gramatike:

- sintaksno upravljane definicije (*Syntax Directed Definitions*)
- translacione šeme (*Translation Schemes*)

Sintaksno upravljane definicije su jednostavno definicije rutina pridružene pravilima. Daju se na dosta visokom nivou i ne sadrže informaciju o tome kako se realizuju i kojim redosledom se primenjuju. Translacione šeme pokazuju redosled u kome se primenjuju semantička pravila što na neki način utiče i na njihovu implementaciju. To znači da su nešto nižeg nivoa od definicija.

### 11.1 Sintaksno upravljane definicije

Sintaksno upravljane definicije su modifikovane beskonteksne gramatike kod kojih su simbolima gramatike pridodati atributi. Ako simbol gramatike zamislimo kao čvor u sintaksnom stablu, pri čemu se taj čvor implementira kao slog sa više polja, atributi odgovaraju imenima tih polja. Atributi mogu da predstavljaju bilo šta: string, broj, memorijsku lokaciju ili bilo šta drugo što može da zatreba u toku analize. Vrednost atributa određena je semantičkom rutinom koja je pridružena pravilu koje se koristi u odgovarajućem čvoru sintaksnog stabla.

Atributi mogu da budu generisani i nasleđeni. Vrednosti generisanih atributa se izračunavaju kao funkcije atributa potomaka odgovarajućeg čvora dok se vred-

nosti nasleđenih atributa izračunavaju kao funkcije atributa roditeljskih čvorova i atributa ostalih čvorova na istom nivou.

Na osnovu semantičkih rutina mogu da se generišu grafovi zavisnosti (*Dependency Graphs*) koji pokazuju zavisnost između atributa i određuju kojim redosledom se izračunavaju vrednosti atributa.

Sintaksno stablo koje sadrži vrednosti atributa pridruženih čvorovima se naziva označemp sintaksno stablo (*Annotated Syntax Tree*).

## 11.2 Generisani atributi

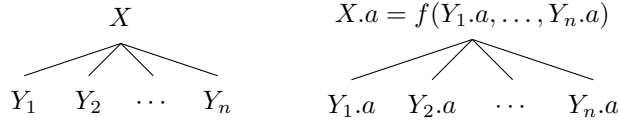
U sintaksno upravljanoj definiciji gramatike svakom pravilu gramatike pridružuje se skup semantičkih rutina koje određuje kako se izračunava vrednost atributa koji je pridružen neterminalnom simbolu na levoj strani pravila. Vrednosti atributa se izračunavaju u funkciji vrednosti atributa pridruženih simbolima na desnoj strani pravila (slika 11.1).

Za pravila oblika

$$X \rightarrow Y_1 Y_2 \dots Y_n$$

semantičke rutine su oblika

$$X.a = f(Y_1.a, Y_2.a, \dots, Y_k.a, \dots, Y_n.a).$$



SLIKA 11.1: Generisan atribut

Generisani atributi se prosleđuju naviše kroz sintaksno stablo. Leksički analizator obično obezbeđuje atribute terminalnih simbola od kojih se dalje generišu atributi neterminala i prosleđuju naviše kroz sintaksno stablo.

U tabeli 11.1 dat je skup pravila gramatike kojom se definiše jednostavan kalkulator koji izvršava sabiranje i množenje cifara.

Semantičkim rutinama koje su pridružene pravilima u tabeli 11.1 definiše se kako se izračunavaju vrednosti atributa pridruženih neterminalnim simbolima gramatike. Preko atributa val izračunavaju se vrednost izraza i štampa preko funkcije print, koja predstavlja semantičku rutinu pridruženu startnom simbolu. Funkcija `top.get(digit.lexval)` preuzima iz tablice simbola vrednost cifre predstavljene preko tokena **digit**.



TABELA 11.1: Sintaksno upravljana definicija jednostavnog kalkulatora

PRAVILO GRAMATIKE	SEMANTIČKE RUTINE
$L \rightarrow E\#$	<code>print(<math>E.val</math>)</code>
$E \rightarrow E + T$	<code><math>E.val := E.val + T.val</math></code>
$E \rightarrow T$	<code><math>E.val := T.val</math></code>
$T \rightarrow T * F$	<code><math>T.val := T.val * F.val</math></code>
$T \rightarrow F$	<code><math>T.val := F.val</math></code>
$F \rightarrow (E)$	<code><math>F.val := E.val</math></code>
$F \rightarrow \text{digit}$	<code><math>F.val := \text{top.get}(\text{digit.lexval})</math></code>

**Primer 11.1** Izračunavanje izraza  $5 * (3 + 4)$ 

Na slici 11.2 prikazano je obeleženo sintaksno stablo prema kome se izračunava vrednost izraza  $5 * (3 + 4)$ , pri čemu se koriste semantička pravila data u tabeli 11.1. Vrednost izraza se izračunava preko generisanih atributa obilaskom stabla odozdo naviše.

U tabeli 11.2 prikazan je redosled izračunavanja i vrenosti atributa za razmatrani primer.

## 11.3 Nasledeni atributi

Nasledeni atributi se prosleđuju naniže kroz sintaksno stablo, odnosno vrednosti atributa na desnoj strani pravila se nasleđuju od atributa sa leve strane pravila (atributa roditeljskih čvorova) i od atributa drugih simbola na desnoj strani pravila (atributa čvorova na istom nivou); slika 11.3. U ovom slučaju za pravila oblika

$$X \rightarrow Y_1, Y_2, \dots, Y_n,$$

a semantičke rutine su oblika

$$Y_k.a = f(X.a, Y_1.a, Y_2.a, \dots, Y_{k-1}.a, Y_{k+1}.a, \dots, Y_n.a).$$

Ukoliko se za generisanje sintaksnog stabla korисти neki bottom-up algoritam, u ovom slučaju prvo se generiše sintaksno stablo pa se prolaskom kroz stablo izračunavaju vrednosti atributa.

Ovi atributi se koriste da se proslede informacije o čvorovima stabla naniže kroz stablo.

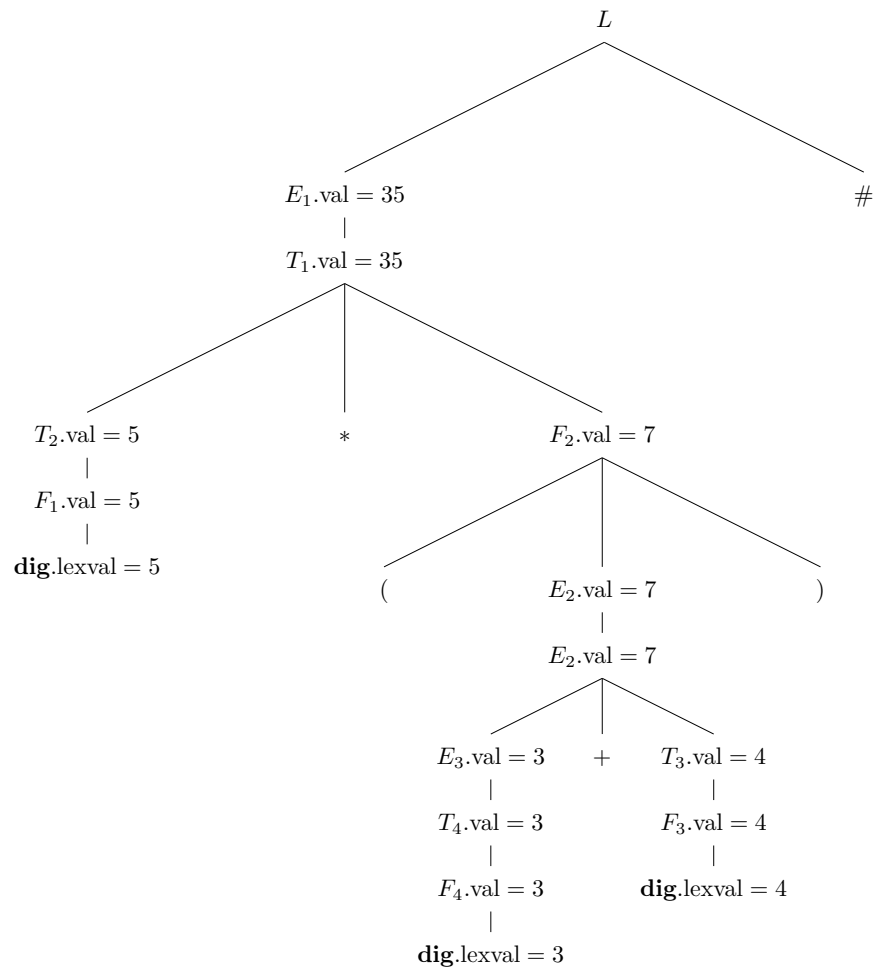
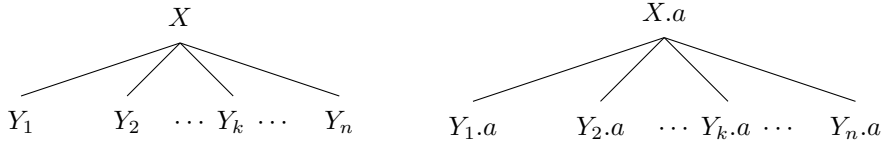
SLIKA 11.2: Označeno sintaksno stablo za izraz  $5 * (3 + 4)$  uz primer 11.1

TABELA 11.2: Redosled izračunavanje i vrednosti atributa za izraz  $5 * (3 + 4)$  uz primer 11.1

ČVOR	ATRIBUT	VREDNOST
$F_1$	$F_1.\text{val} = \mathbf{digit}.\text{lexval}$	5
$T_2$	$T_2.\text{val} := F_1.\text{val}$	5
$F_3$	$F_3.\text{val} = \mathbf{digit}.\text{lexval}$	3
$T_4$	$T_4.\text{val} := F_3.\text{val}$	3
$E_3$	$E_3.\text{val} := T_4.\text{val}$	3
$F_4$	$F_4.\text{val} = \mathbf{digit}.\text{lexval}$	4
$T_3$	$T_3.\text{val} := F_4.\text{val}$	3
$E_2$	$E_2.\text{val} := T_4.\text{val} + T_3.\text{val}$	$3 + 4 = 7$
$F_2$	$F_2.\text{val} := E_2.\text{val}$	7
$T_1$	$T_1.\text{val} := T_2.\text{val} * F_2.\text{val}$	$5 * 7 = 35$
$E_1$	$E_1.\text{val} := T_1.\text{val}$	35
$L$	$\text{print}(E_1.\text{val})$	35



SLIKA 11.3: Nasleđeni atributi

U tabeli 11.3 date su semantičke rutine za primenu nasleđenih atributa u delu gramtike kojom se definišu opisi tipova promenljivih. Uočimo da se semantičkim pravilom  $L.\text{in} := T.\text{type}$ , atribut  $L.\text{in}$  simbola  $L$  izračunava preko vrednosti atributa  $T.\text{type}$  simbola  $T$  koji je na istom nivou sa simbolom  $L$ . Takođe, semantičkim pravilom  $L_1.\text{in} := L.\text{in}$  atribut  $L_1.\text{in}$  se izračunava preko atributa  $L.\text{in}$  iz roditeljskog čvora.

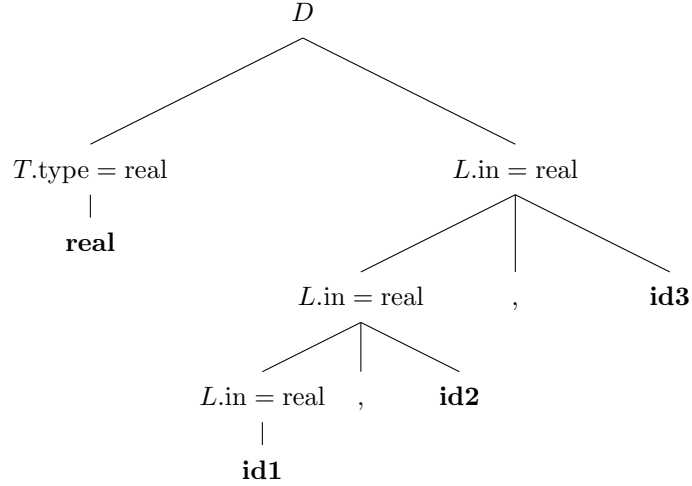
**Primer 11.2** Nasleđeni atributi – upis atributa tipa u tablicu simbola

Na slici 11.4 prikazano je označeno sintaksno stablo koje se dobija za izraz **real id1, id2, id3**.

Prolaskom kroz sintaksno stablo određuje se najpre vrednost atributa `type`

TABELA 11.3: Sintaksno upravljana definicija za upis tipova podataka

PRAVILO GRAMATIKE	SEMANTIČKE RUTINE
$D \rightarrow T L$	$L.in := T.type$
$T \rightarrow \mathbf{int}$	$T.type := integer$
$T \rightarrow \mathbf{real}$	$t.type := real$
$L \rightarrow L_1, \mathbf{id}$	$L_1.in := L.in$ i $addtype(\mathbf{id}.entry, L.in)$
$L \rightarrow \mathbf{id}$	$addtype(\mathbf{id}.entry, L.in)$



SLIKA 11.4: Označeno stablo za definiciju tipa uz primer 11.2

identifikatora  $T$ . Ta vrednost se prenosi na atribut  $L.in$  i prosleđuje naniže kroz stablo sve do terminalnih simbola  $\mathbf{id}$ , kada se upisuje u tablicu simbola u slogu koji odgovara tom identifikatoru. Ovaj upis u tablicu simbola vrši se preko funkcije  $addtype(\mathbf{id}.entry, L.in)$ , gde  $\mathbf{id}.entry$  pokazivač na slog u tablici simbola gde se pamte atributi identifikatora  $\mathbf{id}$ .

U tabeli 11.4 predstavljen je redosled i vrednosti atributa koji se izračunavaju prema pravilima datim u tabeli 11.3 za izraz **real id1, id2, id3**.

### Primer 11.3 Provera identifikatora

U tabeli 11.4 predstavljen je gramatika kojom se definiše struktura jedno-

Čvor	PRAVILO	VREDNOST	Vrsta atributa
$T$	$T.type = \mathbf{real}$	<b>real</b>	generisani
$D$	$L_1.in = T.type$	$L_1.in := \mathbf{real}$	nasleđeni
$L_1$	$L_2.in := L_1.in$ $addtype(id_3.entry, L_1.in)$	$L_2.in := \mathbf{real}$ U TS se upisuje (id <sub>3</sub> .entry, <b>real</b> )	nasleđeni
$L_2$	$L_3.in := L_2.in$ $addtype(id_2.entry, L_2.in)$	$L_3.in := \mathbf{real}$ U TS se upisuje (id <sub>2</sub> .entry, <b>real</b> )	nasleđeni
$L_3$	$addtype(id_1.entry, L_3.in)$	U TS se upisuje (id <sub>1</sub> .entry, <b>real</b> )	nasleđeni

TABELA 11.4: Redosled izračunavanja i vrednosti atributa za primer 11.2

stavnog programa (neterminal  $P$ ) koji sadrži opise ( $D$ ) i naredbe ( $S$ ). Deo sa definicijama se sastoji od liste definicija oblika **var**  $V$ , gde se neterminal  $V$  preslikava u imena promenljivih. Predviđene su samo promenljive  $x$ ,  $y$  i  $z$ .

Deo sa naredbama može da ima jednu ili više naredbi dodeljivanja oblika  $V := E$ , gde je  $E$  neterminal koji se odnosi na aritmetički izraz na desnoj strani naredbe dodeljivanja. Oblik aritmetičkih izraza ovom gramatikom nije definisan.

Semantička pravila gramatike su tako definisana da se u delu sa definicijama tipa formula lista promenljivih koje su definisane i ta lista pamti kao atribut  $D.\text{dl}$ . Ta vrednost se kao nasleđeni atribut preslikava na neterminal  $S$  kao atribut  $S.\text{dl}$ . Ova lista se koristi kod prepoznavanja svake naredbe dodeljivanja da se proveri da je promenljiva kojoj se dodeljuje vrednost prethodno definisana.

Na slici 11.5 predstavljeno je označeno sintaksno stablo koje odgovara prepoznavanju sledećeg programa definisanog gramatikom iz tabele 11.4.

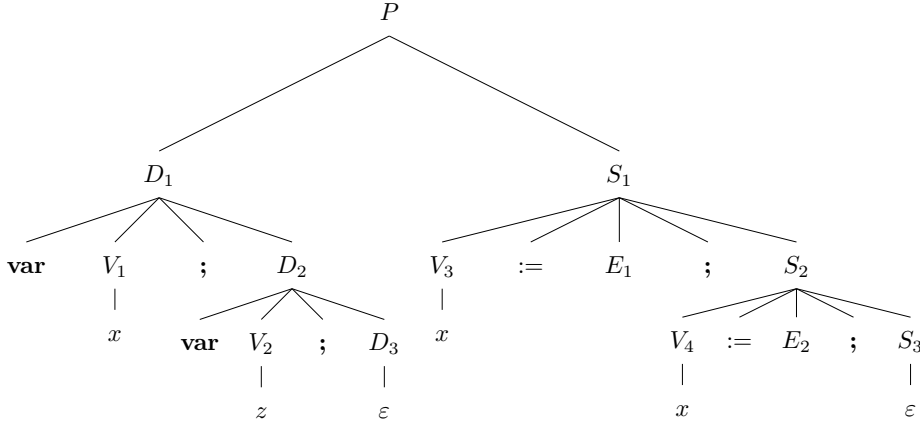
```
var x;
var z;
x := ...;
z := ...;
```

U tabeli 11.6 predstavljen je postupak i redosled izračunavanja atributa. U poslednjoj koloni nalazi se naznaka tipa atributa.

TABELA 11.5: Sintaksno upravljana definicija za proveru identifikatora

PRAVILO GRAMATIKE	SEMANTIČKA RUTINA
$P \rightarrow D S$	$S.\text{dl} = D.\text{dl}$
$D_1 \rightarrow \text{var } V D_2$	$D_1.\text{dl} = \text{addlist}(V.\text{name}, D_2.\text{dl})$
$D_1 \rightarrow \varepsilon$	$D_1.\text{dl} = \text{NULL}$
$S_1 \rightarrow V := E ; S_2$	$\text{check}(V.\text{name}, S_1.\text{dl}), \quad S_2.\text{dl} = S_1.\text{dl}$
$S_1 \rightarrow \varepsilon$	/
$V \rightarrow x$	$V.\text{name} = x$
$V \rightarrow y$	$V.\text{name} = y$
$V \rightarrow z$	$V.\text{name} = z$

Atributne gramatike se u praksi jako puno koriste za rešavanje različitih problema. U narednim poglavljima biće pokazano kako se preko atributa i seman-



SLIKA 11.5: Označeno sintaksno stablo za gramatiku iz primera 11.3

tičkih rutina definiše generisanje međukoda, kao i proveru tipova podataka.

## 11.4 Provera tipova podataka (*Type Checking*)

Sastavni deo savremenih kompilatora su i moduli za proveru tipova podataka koji se realizuju u skladu sa time kako je postavljen koncept tipova podataka u jeziku na koji se odnosi kompilator. Za proveru tipova podataka danas se obično koriste atributne gramatike.

### 11.4.1 Semantička pravila za proveru tipova podataka

Za proveru tipova podataka koriste se atributi tipa koji se za svaki element jezika koji ima svojstvo tipa čuvaju u tabeli simbola. Vrednosti ovih atributa određuju se u toku sintaksne analize definicija tipova podataka.

Razmotrićemo primer jednog jednostavnog jezika definisanog sledećom gramatikom:

$$\begin{aligned}
 P &\rightarrow D; E && \text{(program se sastoji od opisa i izvršnog dela)} \\
 D &\rightarrow D; D \mid \text{id} : T \\
 T &\rightarrow \text{char} \mid \text{integer} \mid \text{array}[\text{num}] \text{ of } T \mid ^T \\
 E &\rightarrow \text{literal} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E[E] \mid ^E
 \end{aligned}$$

U tabeli 11.7 data su semantička pravila kojima se generišu atributi tipa za identifikatore i pomoću funkcije  $\text{addType}(\text{id.entry}, T.\text{type})$  upisuju u tabelu simbola

TABELA 11.6: Izračunavanje atributa za program iz primera 11.3

ČVOR	ATRIBUTI	TIP ATRIBUTA
$V_1$	$V_1.name = x$	generisani
$V_2$	$V_2.name = z$	generisani
$D_3$	$D_3.dl = \text{NULL}$	generisani
$D_2$	$D_2.dl = \text{addlist}(z, D_3.dl) = (z)$	generisani
$D_1$	$D_1.dl = \text{addlist}(x, D_2.dl) = (x, z)$	generisani
$P$	$S_1.dl := D_1.dl = (x, z)$	nasleđeni
$V_3$	$V_3.name = x$	generisani
$S_1$	$\text{check}(x, (x, z)), S_2.dl := S_1.dl = (x, z)$	nasleđeni
$V_4$	$V_4.name = z$	generisani
$S_2$	$\text{check}(z, (x, z)), S_3.dl := S_2.dl$	nasleđeni

u polje odgovarajućeg identifikatora.

TABELA 11.7: Semantička pravila za tipove identifikatora

PRAVILA GRAMATIKA	SEMANTIČKE RUTINE
$P \rightarrow D; E$	
$D \rightarrow D; D$	
$D \rightarrow \text{id} : T$	$\text{addType}(\text{id.entry}, T.\text{type})$
$T \rightarrow \text{char}$	$T.\text{type} := \text{char}$
$T \rightarrow \text{integer}$	$T.\text{type} := \text{integer}$
$T \rightarrow \sim T_1$	$T.\text{type} := \text{pointer}(T_1.\text{type})$
$T \rightarrow \text{array}[\text{num}] \text{ of } T_1$	$T.\text{type} := \text{array}(1 \dots \text{num.val}, T_1.\text{type})$

Primer semantičkih pravila kojima se proverava kompatibilnost tipova podataka u aritmetičkim izrazima dat je u tabeli 11.8.

Atribut tipa može da se primeni i na naredbe. Ako je naredba korektna onda ovaj atribut dobija vrednost void ako nije generiše se poruka o grešci. Za generisanje atributa tipa naredbi mogu se koristiti semantička pravila data u tabeli 11.9.

Na sličan način mogu da se proveravaju i druga pravila vezana za tipove po-



PRAVILA GRAMATIKE	SEMANTIČKE RUTINE
$E \rightarrow \text{literal}$	$E.\text{type} := \text{char}$
$E \rightarrow \text{num}$	$E.\text{type} := \text{integer}$
$E \rightarrow \mathbf{id}$	$E.\text{type} := \text{lookup}(\mathbf{id}.\text{entry})$
$E \rightarrow E_1 \text{ mod } E_2$	$E.\text{type} = \begin{cases} \text{integer}, & E_1.\text{type} = \text{integer} \wedge E_2.\text{type} = \text{integer} \\ \text{type error}, & \text{inače} \end{cases}$
$E \rightarrow E_1[E_2]$	$E.\text{type} = \begin{cases} t, & E_2.\text{type} = \text{integer} \wedge E_1.\text{type} = \text{array}(s, t) \\ \text{type error}, & \text{inače} \end{cases}$
$E \rightarrow E_1^\wedge$	$E.\text{type} = \begin{cases} t, & E_1.\text{type} = \text{pointer}(t) \\ \text{type error}, & \text{inače} \end{cases}$
Pozivi funkcija $E \rightarrow E_1(E_2)$	$E.\text{type} = \begin{cases} t, & E_2.\text{type} = s \wedge E_1.\text{type} = s \rightarrow t \\ \text{type error}, & \text{inače} \end{cases}$

TABELA 11.8: Provera tipova u izrazima. U poslednja tri reda,  $t$  je elementarni tip dobijen iz strukturnog tipa  $\text{array}(s, t)$ .

TABELA 11.9: Provera strukture naredbe

PRAVILO GRAMATIKE	SEMANTIČKO PRAVILO
$S \rightarrow id := E$	$S.type = \begin{cases} \text{void}, & id.type = E.type \\ \text{type error}, & \text{inače} \end{cases}$
$S \rightarrow \text{if } E \text{ then } S_1$	$S.type = \begin{cases} S_1.type, & E.type = \text{boolean} \\ \text{type error}, & \text{inače} \end{cases}$
$S \rightarrow \text{while } E \text{ do } S_1$	$S.type = \begin{cases} S_1.type, & E.type = \text{boolean} \\ \text{type error}, & \text{inače} \end{cases}$
$S \rightarrow S_1 ; S_2$	$S.type = \begin{cases} \text{void}, & S_1.type = \text{void} \wedge S_2.type = \text{void} \\ \text{type error}, & \text{inače} \end{cases}$

dataka u jeziku. Na primer kod jezika kod kojih se koriste implicitna pravila za konverziju tipova u izrazima koji sadrže operande različitog tipa mogu da se definišu semantička pravila kojima se to realizuje. U tabeli 11.10 data su semantička pravila za konverziju tipa *integer* u tip *real*.

TABELA 11.10: Konverzija tipova

PRAVILO GRAMATIKE	SEMANTIČKO PRAVILO
$E \rightarrow num$	$E.type := \text{integer}$
$E \rightarrow num.num$	$E.type := \text{real}$
$E \rightarrow id$	$E.type := \text{lookup}(id.entry)$
$E \rightarrow E_1 \text{ op } E_2$	$E.type = \begin{cases} \text{integer}, & E_1.type = \text{integer} \wedge E_2.type = \text{integer} \\ \text{real}, & E_1.type = \text{integer} \wedge E_2.type = \text{real} \\ \text{real}, & E_1.type = \text{real} \wedge E_2.type = \text{integer} \\ \text{real}, & E_1.type = \text{real} \wedge E_2.type = \text{real} \\ \text{type error}, & \text{inače} \end{cases}$

## 11.5 Pitanja

1. Objasniti ulogu atributa i semantičkih rutina atributnih gramatika.

2. Objasniti koncept generisanih atributa i dati primer gramatike sa ovim atributima.
3. Objasniti koncept nasleđenih atributa i dati primer gramatike sa ovim atributima.
4. Šta je zadatak modula za proveru tipova podataka.
5. Dati primer semantičkih pravila kojima se generišu atributi tipa identifikatora.
6. Dati primer semantičkih pravila kojima se vrši provera kompatibilnosti tipova u izrazima.
7. Dati primer semantičkih pravila kojima se vrši provera upravljačkih naredbi programa.
8. Dati primer semantičkih pravila kojima se vrši konverzija tipa u izrazima.

## 11.6 Zadaci

1. Pokazati kako se izračunava izraz  $(5 + 3 + 4) * (2 + 5)$  korišćenjem gramatike i semantičkih pravila datih u tabeli 11.1. Izračunavanje predstaviti obeleženim sintaksnim stablom.
2. U tabeli 11.11 data je gramatika sa semantičkim pravilima kojima se definišu binarni brojevi i izračunava njihova dekadna vrednost. Korišćenjem ove gramatike,
  - (a) nacrtati sintakсно stablo za binarni broj 1011;
  - (b) na označenom sintaksnom stablu prikazati izračunavanje vrednosti binarnog broja 1011 i
  - (c) prikazati koji su atributi generisani, a koji nasleđeni.
3. U tabeli 11.12 data je gramatika sa semantičkim pravilima kojima se definišu identifikatori kod kojih se pojavljuju slova  $a$ ,  $b$  i  $c$ , ali ne u parovima. Za zadatau gramatiku
  - (a) prikazati koji su atributi generisani, a koji nasleđeni;
  - (b) nacrtati sintakсно stablo za identifikator  $bca$  i
  - (c) na označenom sintaksnom stablu prikazati proveru za  $bca$ .

TABELA 11.11: Pravila za izračunavanje vrednosti binarnih brojeva

PRAVILA GRAMATIKE	SEMANTIČKE RUTINE
$B \rightarrow 1$	$B.\text{pos} = 1$ $B.\text{val} = D.\text{val}$ $D.\text{pow} = 0$
$B_1 = DB_2$	$B_1.\text{pos} = B_2.\text{pos} + 1$ $B_1.\text{val} = B_2.\text{val} + D.\text{val}$ $D.\text{pow} = B_2.\text{pos}$
$D = 0$	$D.\text{val} = 0$
$D = 1$	$D.\text{val} = 2^{D.\text{pow}}$

TABELA 11.12: Provera identifikatora.  $\circ$  je znak za nadovezivanje.

PRAVILA GRAMATIKE	SEMANTIČKE RUTINE
$D \rightarrow I$	$I.\text{str} = \{\}$ $I.\text{last} = \varepsilon$ $D.\text{val} = I.\text{val}$ accept, ako je $D.\text{val} \neq \text{error}$
$I \rightarrow LI_1$	$L.\text{str} = I.\text{str}$ $L.\text{last} = I.\text{last}$ $I_1.\text{str} = L.\text{val}$ $I.\text{val} = I_1.\text{val}$
$I \rightarrow L$	$L.\text{str} = I.\text{str}$ $L.\text{last} = I.\text{last}$ $I.\text{val} = L.\text{val}$
$L \rightarrow a$	$L.\text{val} = \begin{cases} L.\text{str} \circ a & L.\text{last} \neq a \\ \text{error} & \text{inače} \end{cases}$ $L.\text{last} = a$
$L \rightarrow b$	$L.\text{val} = \begin{cases} L.\text{str} \circ b & L.\text{last} \neq b \\ \text{error} & \text{inače} \end{cases}$ $L.\text{last} = b$
$L \rightarrow c$	$L.\text{val} = \begin{cases} L.\text{str} \circ c & L.\text{last} \neq c \\ \text{error} & \text{inače} \end{cases}$ $L.\text{last} = c$

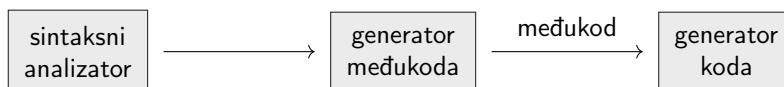
## Glava 12

# Međukodovi

U ovom poglavlju biće razmotrene vrste međukodova koji se koriste u okviru kompilatora kao i tehnike za njihovo generisanje. Akcenat će biti na apstraktnom sintaksnom stablu kao strukturi koja predstavlja osnovu za generisanje koda kao i na troadresnom međukodu koji se danas najviše koristi u praksi.

### 12.1 Mesto međukoda u procesu prevođenja

Proces analize koda se obično završava generisanjem međukoda koji je mašinski nezavisna reprezentacija koda pogodna za generisanje asemblerskog ili mašinskog koda različitih ciljnih mašina (slika 12.1).



SLIKA 12.1: Mesto međukoda u okviru kompilatora

Kompilator može da koristi i međukod u više nivoa, kako je to predstavljeno na slici 12.2.



SLIKA 12.2: Međukod u više nivoa

Postoji više razloga zbog kojih se uvodi međukod, od kojih su najznačajniji:

- Prenosivost koda – na osnovu međukoda može se generisati kod za različite ciljne mašine.

- Optimizacija koda – na nivou međukoda može da se vrši mašinski nezavisna optimizacija.

Međukod može da bude generisan u različitim oblicima. Ovde ćemo razmotriti apstraktno sintaksno stablo i troadresni međukod kao dva najšire rasprostranjena oblika međukoda. Nekada se kao međukod koristila i poljska inverzna notacija koja je pogodna za interpretiranje koda i generisanje mašinskog koda korišćenjem steka. Veoma često se kao međukod koristi i programski jezik C, zato što se naredbe ovog jezika postavljene dosta blizu asemblerskom jeziku i lako se transformišu u asemblerski jezik.

## 12.2 Apstraktno sintaksno stablo

Apstraktno sintaksno stablo je struktura koja se generiše na osnovu sintaksnog stabla i na neki način predstavlja redukovanu reprezentaciju sintaksnog stabla. Obično se generiše direktno u toku same sintaksne analize ili se najpre generiše sintaksno stablo (*parse tree*) a nakon toga ta struktura transformiše u apstraktno sintaksno stablo. U tabeli 12.1 predstavljena su pravila gramatike kojom se definišu aritmetički izrazi, kao i semantička pravila koja se pridružuju ovim pravilima, koja se koriste za generisanje apstraktnog sintaksnog stabla. U slučaju bootum-up analize, semantička pravila se primenjuju posle redukcije pravilom gramatike kome su pridružena dok se u slučaju top-down analize najpre generiše sintaksno stablo (utvrdi redosled primene pravila gramatike), a zatim to stablo obilazi odozdo naviše i pri tome se primenom semantičkih pravila generiše apstraktno sintaksno stablo.

TABELA 12.1: Semantička pravila kojima se generiše apstraktno sintaksno stablo

PRAVILA GRAMATIKA	SEMANTIČKE RUTINE
$S \rightarrow \text{id} := E$	$S.\text{sptr} := \text{mknode}(:=, \text{mkleaf}(\text{id}, \text{id.entry}), E.\text{nptr})$
$E \rightarrow E_1 + E_2$	$E.\text{nptr} := \text{mknode}(+, E_1.\text{nptr}, E_2.\text{nptr})$
$E \rightarrow E_1 * E_2$	$E.\text{nptr} := \text{mknode}(*, E_1.\text{nptr}, E_2.\text{nptr})$
$E \rightarrow -E_1$	$E.\text{nptr} := \text{mknode}(-, E_1.\text{nptr})$
$E \rightarrow (E_1)$	$E.\text{nptr} := E_1.\text{nptr}$
$E \rightarrow \text{id}$	$E.\text{nptr} := \text{mkleaf}(\text{id}, \text{id.entry})$

Korišćenjem semantičkih pravila datih u tabeli 12.1 može se konstruisati apstraktno sintaksno stablo za naredbe dodeljivanja definisane datom gramatikom.

U tabeli 12.1, izraz oblika

$$P.\text{nptr} := \text{mknode}(V, A.\text{nptr}, B.\text{nptr})$$

predstavlja poziv funkcije kojom se generiše čvor apstraktnog sintaksnog stabla pri čemu je  $P.nptr$  pokazivač na taj čvor, prvi argument ( $V$ ) vrednost u čvoru stabla, a druga dva argumenta su pokazivači na čvorove potomke.

Izraz

$$P.nptr := \text{mkleaf}(\text{id}, \text{id.entry})$$

predstavlja poziv funkcije kojom se generiše list stabla (terminalni čvor) za identifikator čija se prava vrednost preuzima iz tabele simbola preko pokazivača  $\text{id.entry}$ , a  $P.nptr$  je pokazivač na taj čvor.

### Primer 12.1 Generisanje apstraktnog sintaksnog stabla

U slučaju naredbe dodeljivanja  $a := b * (d + c) + b * (d + c)$  koju sintakсни analizator dobija u obliku  $\text{id} := \text{id} * (\text{id} + \text{id}) + \text{id} * (\text{id} + \text{id})$ , primenom semantičkih pravila iz tabele 12.1 biće generisano sintakšno stablo prikazano na slici 12.3. Na slici 12.4 prikazan je i odgovarajući dijagram koji se dobija redukcijom stabla tako što se ne generišu novi čvorovi u slučaju da odgovarajući već postoje.

Na osnovu date gramatike, levim izvođenjem izraz  $\text{id} := \text{id} * (\text{id} + \text{id}) + \text{id} * (\text{id} + \text{id})$  se generiše primenom sledećih smena:

$$\begin{aligned} S &\rightarrow \text{id} := E \\ &\rightarrow \text{id} := E + E \\ &\rightarrow \text{id} := E * E + E \\ &\rightarrow \text{id} := \text{id} * E + E \\ &\rightarrow \text{id} := \text{id} * (E) + E \\ &\rightarrow \text{id} := \text{id} * (E + E) + E \\ &\rightarrow \text{id} := \text{id} * (\text{id} + E) + E \\ &\rightarrow \text{id} := \text{id} * (\text{id} + \text{id}) + E \\ &\rightarrow \text{id} := \text{id} * (\text{id} + \text{id}) + E * E \\ &\rightarrow \text{id} := \text{id} * (\text{id} + \text{id}) + \text{id} * E \\ &\rightarrow \text{id} := \text{id} * (\text{id} + \text{id}) + \text{id} * (E) \\ &\rightarrow \text{id} := \text{id} * (\text{id} + \text{id}) + \text{id} * (E + E) \\ &\rightarrow \text{id} := \text{id} * (\text{id} + \text{id}) + \text{id} * (\text{id} + E) \\ &\rightarrow \text{id} := \text{id} * (\text{id} + \text{id}) + \text{id} * (\text{id} + \text{id}) \end{aligned}$$

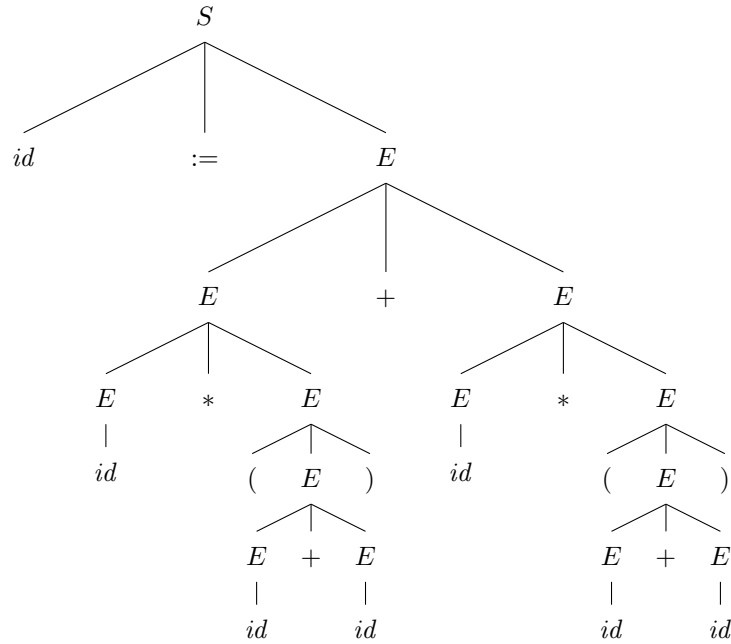
U ovom slučaju prilikom generisanja apstraktnog sintaksnog stabla biće

izvršen sledeći niz koraka.

```

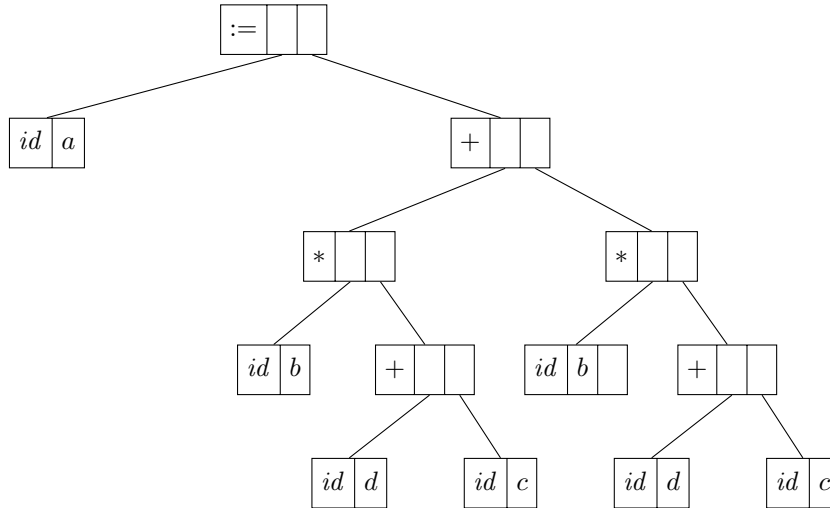
 $p_1 = \text{mkleaf}(d, d.\text{entry})$ 
 $p_2 = \text{mkleaf}(c, c.\text{entry})$ 
 $p_3 = \text{mknode}(' + ', p_1, p_2)$ 
 $p_4 = \text{mkleaf}(b, b.\text{entry})$ 
 $p_5 = \text{mknode}(' * ', p_3, p_4)$ 
 $p_6 = \text{mkleaf}(d, d.\text{entry})$ 
 $p_7 = \text{mkleaf}(c, c.\text{entry})$ 
 $p_8 = \text{mknode}(' + ', p_6, p_7)$ 
 $p_9 = \text{mkleaf}(b, b.\text{entry})$ 
 $p_{10} = \text{mknode}(' * ', p_8, p_9)$ 
 $p_{11} = \text{mknode}(' + ', p_5, p_{10})$ 
 $p_{12} = \text{mknode}(' := ', \text{mkleaf}(a, a.\text{entry}), p_{11})$ 

```



SLIKA 12.3: Sintaksno stablo za primer 12.1. Odgovarajuće apstraktno sintaksno stablo je dato na slici 12.4.





SLIKA 12.4: Apstraktno sintakšno stablo za primer 12.1. Odgovarajuće sintakšno stablo je dato na slici 12.3.

## 12.3 Implementacija apstraktnog sintaksnog stabla

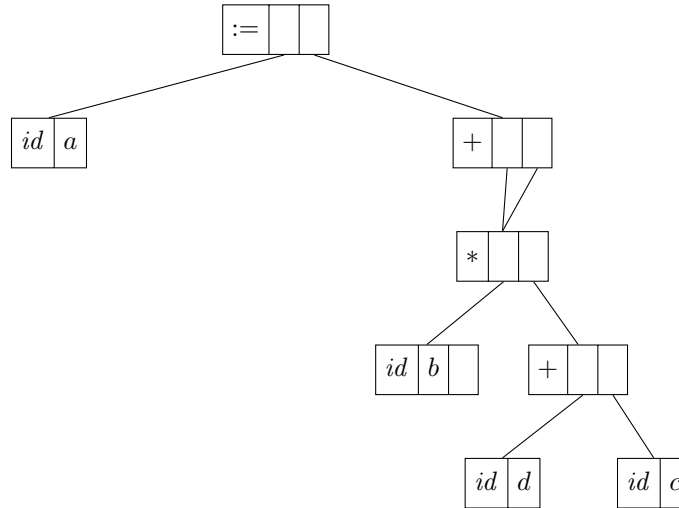
Apstraktno sintakšno stablo se može memorisati kao statička ili dinamička struktura.

Procedure iz tabele 12.1 definisane su tako da se generiše dinamička struktura koja je za razmatrani primer prikazana na slici 12.4. Prilikom generisanja strukture mogu da se izvršavaju i procedure za redukciju stabla tako da se ne generišu novi čvorovi ako u stablu već postoje odgovarajući. U tom slučaju dobija se struktura dijagrama u okviru koje do nekih čvorova vodi više pokazivača, za razmatrani primer, slika 12.5.

U slučaju statičke implementacije umesto pokazivača u poljima slogova pamte se adrese odgovarajućih slogova sa kojima su povezani. Za naredbu dodeljivanja iz primera 12.1, statička implementacija je data u tabeli 12.2.

## 12.4 Troadresni međukod

Troadresni međukod je neka vrsta pseudoasemblerkog jezika koji se može lako preslikati u drugi asemblerki jezik ili mašinski kod. Sastoji se od jednostavnih naredbi koje mogu da imaju najviše jedan operator tako da se složeni izrazi predstavljaju sekvencom naredbi. Na primer, izraz  $x + y * z$  može da se pred-



SLIKA 12.5: Apstraktno sintaksno stablo memorisano kao dijagram

stavi sekvencom troadresnih instrukcija

$$t_1 = y * z$$

$$t_2 = t_1 + x$$

gde su  $t_1$  i  $t_2$  privremene (temporalne) promenljive koje generiše kompilator. Takođe, troadresni kod sadrži i jednostavne upravljačke naredbe, kao što su bezuslovni skokovi i uslovni skokovi, koje se direktno preslikavaju u asemblerske naredbe.

Lista osnovnih naredbi troadresnog koda se sastoji od sledećih naredbi.

1. Naredba dodeljivanja  $x := y \text{ op } z$ , gde je op aritmetički ili logički binarni operator.
2. Naredba dodeljivanja  $x := \text{op } y$ , gde je op aritmetički ili logički unarni operator (npr. minus, logička negacija, *shift* operator, operatori za konverziju tipova i sl).
3. Naredba kopiranja  $x := y$  kojom  $x$  dobija vrednost  $y$ .
4. Oznaka ili labela  $L$  :
5. Naredba bezuslovnog skoka **goto**  $L$ .
6. Naredba uslovnog skoka **if**  $x \text{ rel op } y$  **goto**  $L$ .
7. **param**  $x$ , navođenje parametara.
8. **call**  $p, n$  za poziv potprograma. Pre ove naredbe navode se parametri

TABELA 12.2: Statička implementacija apstraktnog stabla

0	id	d	
1	id	c	
2	+	0	1
3	id	b	
4	*	2	3
5	id	d	
6	id	c	
7	+	5	6
8	id	b	
9	*	7	8
10	id	a	
11	:=	10	9

potprograma, na primer:

```

param  $x_1$ 
param  $x_2$ 
       $\vdots$ 
param  $x_n$ 
call  $p, n$ 

```

9.  $x := \mathbf{fcall} \ f, n$  – poziv funkcijskog potprograma.  $n$  označava broj argumenata potprograma koji se kao i u prethodnom slučaju navode pre naredbe za poziv potprograma.
10. **return** – povratak iz potprograma.
11. **freturn**  $x$  – povratak iz funkcijskog potprograma kada se kao rezultat vraća vrednost promenljive  $x$ .
12.  $x := y[i]$  – preuzimanje vrednosti elementa polja.
13.  $x[i] := y$  – dodeljivanje vrednosti elementu polja.

**Primer 12.2** Primer troadresnog međukoda

U slučaju naredbe dodeljivanja

$$a := b * (d + c) + b * (d + c),$$

biće generisan sledeći troadresni međukod.

```

t1 := d + c
t2 := b * t1
t3 := d + c
t4 := b * t1
t5 := t2 + t4
a := t5

```

Navedeni niz naredbi troadresnog koda je u suštini samo simbolička reprezentacija ovih naredbi. Kompilator može da koristi svoje specifične strukture podataka za reprezentaciju troadresnih naredbi. U praksi se sreću tri modela koji se obično nazivaju *quadruples*, *triples* i *indirect triples*.

#### 12.4.1 *Quadruples* struktura

U slučaju reprezentacije pomoću *Quadruples* strukture koriste se slogovi sa po četiri polja koja su namenjena smeštaju operatora (op), prvog argumenta (arg<sub>1</sub>), drugog argumenta (arg<sub>2</sub>) i rezultata (rez). Kod nekih naredbi naka od ovih polja ostaju neiskorišćena. U tabeli 12.3 date su osnovne naredbe troadresnog koda predstavljene u formatu *Quadruplesa*.

##### **Primer 12.3** *Quadruples* struktura

Troadresni kod iz primera 12.2 kojim je predstavljena naredba dodeljivanja

$$a := b * (d + c) + b * (d + c)$$

biće memorisan preko *quadruplesa* prikazanih u tabeli 9.5. Napomenimo da će kompilator umesto imena promenljivih *a*, *b* i *c* koristiti pointere na ove identifikatore u tabeli simbola. Temporalne promenljive kompilator može takođe da ubaci u tabelu simbola kao da se radi o promenljivima koje je uveo programer ili za njih generiše posebnu klasu *Temp*.

#### 12.4.2 *Triples* struktura

Kod reprezentacije troadresnog međukoda pomoću *triples* strukture koriste se slogovi sa tri polja. Uočimo da se kod reprezentacije pomoću *quadruples* slogova polje rez uglavnom koristi za smeštaj temporalnih promenljivih koje se generišu

TABELA 12.3: Osnovne troadresne naredbe predstavljene kao *Quadruples*

	NAREDBA	Quadruples	OPIS				
1	$x := y$ <b>binop</b> $z$	<table><tr><td>op</td><td><math>x</math></td><td><math>y</math></td><td><math>z</math></td></tr></table>	op	$x$	$y$	$z$	<b>binop</b> je binarni operator: +, −, *, /, ∧, ∨, <, ≤, =, ≥, >, ≠ op ∈ {PLUS, MINUS, TIMES, DIV, LT, LE, EQ, GE, NE}
op	$x$	$y$	$z$				
2	$x :=$ <b>unop</b> $y$	<table><tr><td>op</td><td><math>x</math></td><td><math>y</math></td><td>−</td></tr></table>	op	$x$	$y$	−	<b>unop</b> je unarni operator: −, ¬ op ∈ {NEG, NOT}
op	$x$	$y$	−				
3	$x := y$	<table><tr><td>ASSIGN</td><td><math>x</math></td><td><math>y</math></td><td>−</td></tr></table>	ASSIGN	$x$	$y$	−	kopiranje vrednosti $y$ u $x$
ASSIGN	$x$	$y$	−				
4	$L :$	<table><tr><td>LABEL</td><td><math>L</math></td><td>−</td><td>−</td></tr></table>	LABEL	$L$	−	−	oznaka (labela)
LABEL	$L$	−	−				
5	<b>goto</b> $L$	<table><tr><td>GOTO</td><td><math>L</math></td><td>−</td><td>−</td></tr></table>	GOTO	$L$	−	−	naredba bezuslovnog skoka
GOTO	$L$	−	−				
6	<b>if</b> $x$ <b>goto</b> $L$	<table><tr><td>IF</td><td><math>L</math></td><td><math>x</math></td><td>−</td></tr></table>	IF	$L$	$x$	−	naredba uslovnog skoka
IF	$L$	$x$	−				
7	<b>paramf</b> $x$	<table><tr><td>PAR</td><td>−</td><td><math>x</math></td><td>−</td></tr></table>	PAR	−	$x$	−	definisanje parametara potprograma
PAR	−	$x$	−				
8	<b>call</b> $p, n$	<table><tr><td>CALL</td><td>−</td><td><math>p</math></td><td><math>n</math></td></tr></table>	CALL	−	$p$	$n$	poziv potprograma $p$ sa $n$ argumenata
CALL	−	$p$	$n$				
9	$x :=$ <b>fcall</b> $f, n$	<table><tr><td>FCALL</td><td><math>x</math></td><td><math>f</math></td><td><math>n</math></td></tr></table>	FCALL	$x$	$f$	$n$	poziv funkcijskog potprograma
FCALL	$x$	$f$	$n$				
10	<b>return</b>	<table><tr><td>RET</td><td>−</td><td>−</td><td>−</td></tr></table>	RET	−	−	−	povratak iz potprograma
RET	−	−	−				
11	<b>return</b> $x$	<table><tr><td>FRET</td><td>−</td><td><math>x</math></td><td>−</td></tr></table>	FRET	−	$x$	−	povratak iz funkcijskog potprograma kada se kao rezultat vraća $x$
FRET	−	$x$	−				
12	$x := y[k]$	<table><tr><td>LDAR</td><td><math>x</math></td><td><math>y</math></td><td><math>k</math></td></tr></table>	LDAR	$x$	$y$	$k$	preuzimanje vrednosti promenljive sa indeksom
LDAR	$x$	$y$	$k$				
13	$x[k] := y$	<table><tr><td>STAR</td><td><math>x</math></td><td><math>k</math></td><td><math>y</math></td></tr></table>	STAR	$x$	$k$	$y$	dodeljivanje vrednosti promenljivoj sa indeksom
STAR	$x$	$k$	$y$				

TABELA 12.4: Međukod memorisan kao *quadruples* struktura

	op	rez	arg <sub>1</sub>	arg <sub>2</sub>
0	+	$t_1$	$d$	$c$
1	*	$t_2$	$b$	$t_1$
2	+	$t_3$	$d$	$c$
3	*	$t_4$	$b$	$t_3$
4	+	$t_5$	$t_2$	$t_4$
5	=	$a$	$t_5$	

u procesu generisanja međukoda i uglavnom služe za čuvanje međurezultata. U slučaju *triples* strukture na temporalne promenljive se ukazuje preko pozicije odgovarajućeg sloga umesto preko rednog broja promenljive.

**Primer 12.4** *Triples* format

Troadresni kod iz primera 12.2 kojim je predstavljena naredba dodeljivanja

$$a := b * (d + c) + b * (d + c)$$

biće memorisan preko *triples* slogova kao u tabeli 12.5.

TABELA 12.5: Međukod memorisan kao *triples* struktura

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	+	<i>d</i>	<i>c</i>
1	*	<i>b</i>	0
2	+	<i>d</i>	<i>c</i>
3	*	<i>b</i>	2
4	+	1	3
5	=	<i>a</i>	4

### 12.4.3 Indirect triples struktura

U slučaju strukture *indirect triples* koristi se i dodatna lista pokazivača na *triples* slogove kojima su predstavljene naredbe. Ovo može da bude pogodno kod implementacije različitih algoritama za optimizaciju koda, kada se vrši preuređivanje naredbi. U tom slučaju razmeštanja se vrše samo u listi pokazivača, a sama struktura sa slogovima naredbi ostaje nepromenjena.

**Primer 12.5** *Indirect triples*

Troadresni kod iz primera 12.2 kojim je predstavljena naredba dodeljivanja

$$a := b * (a + c) + b * (a + c)$$

memoriše *indirect triples* strukturom kao što je to prikazano u tabeli 12.6.

TABELA 12.6: Troadresni kod predstavljen *indirect triples* strukturom

naredbe			op	arg <sub>1</sub>	arg <sub>2</sub>
35	(0)	0	+	<i>d</i>	<i>c</i>
36	(1)	1	*	<i>b</i>	0
37	(2)	2	+	<i>d</i>	<i>c</i>
38	(3)	3	*	<i>b</i>	2
39	(4)	4	+	1	3
40	(5)	5	=	<i>a</i>	4

## 12.5 Troadresni međukod za naredbe dodeljivanja

Slično kao i kod generisanja apstraktnog sintaksnog stabla, i troadresni kod se može renerisati u toku samog procesa sintaksne analize ili nakon generisanja sintaksnog stabla. I u ovom slučaju pravilima gramatike se pridružuju semantičke rutine koje se vezuju za sintaksna pravila. U tabeli 12.7 data su pravila gramatike i pridružena semantička pravila kojima se generiše troadresni međukod aritmetičkih izraza. Uočimo da se semantičkim pravilima generiše atribut `addr` neterminala *E* koji označava adresu memorijske lokacije gde se čuva vrednost za *E*. Funkcija `top.get(id.lexeme)` čita iz tabele simbola leksemu koja predstavlja ime identifikatora (simboličku adresu određenog identifikatora), dok funkcija `top.get(cons.lexval)` preuzima iz tabele simbola vrednost konstante *cons*. Funkcija `gen()` generiše navedeni kod i nadovezuje ga na već postojeći kod. Sa `new Temp()` generiše se nova temporalna promenljiva i smešta u tabelu simbola.

### Primer 12.6 Generisanje troadresnog međukoda za naredbu dodeljivanja

Na slici 12.6 Predstavljeno je sintaksno stablo koje se dobija u toku sintaksne analize naredbe dodeljivanja

$$a := b * (a + c) + b * (a + c)$$

iz primera 12.1. U čvorovima stabla predstavljeni su atributi pridreženi neterminalnim simbolima kao i vrednosti koje im se dodeljuju prema semantičkim pravilima iz tabele 12.7. Ovakvo stablo, na kome su predstavljeni i atributi naziva se označeno sintaksno stablo.

Da bi generisanje međukoda bilo jasnije, na istoj slici predstavljene su i naredbe troadresnog koda koji se generiše. Sa slike se vidi koja sintaksna

TABELA 12.7: Semantička pravila za naredbu dodeljivanja

PRAVILO GRAMATIKE	SEMANTIČKO PRAVILO
(1) $S \rightarrow id := E;$	$\{\text{gen}(\text{top.get}(id.\text{lexeme}) \text{ ' := ' } E.\text{addr})\}$
(2) $E \rightarrow E_1 + E_2$	$\{E.\text{addr} = \text{new Temp}()\}$ $\{\text{gen}(E.\text{addr} \text{ ' := ' } E_1.\text{addr} \text{ ' + ' } E_2.\text{addr})\}$
(3) $E \rightarrow E_1 * E_2$	$\{E.\text{addr} = \text{new Temp}()\}$ $\{\text{gen}(E.\text{addr} \text{ ' := ' } E_1.\text{addr} \text{ ' * ' } E_2.\text{addr})\}$
(4) $E \rightarrow -E_1$	$\{E.\text{addr} = \text{new Temp}()\}$ $\{\text{gen}(E.\text{addr} \text{ ' := ' } \text{minus } E_1.\text{addr})\}$
(5) $E \rightarrow (E_1)$	$\{E.\text{addr} = E_1.\text{addr}\}$
(6) $E \rightarrow id$	$\{E.\text{addr} = \text{top.get}(id.\text{lexeme})\}$
(6) $E \rightarrow \text{cons}$	$\{E.\text{addr} = \text{top.get}(\text{const.lexval})\}$

pravila se izvršavaju u čvorovima i kako teče generisanje koda. Napomenimo da se sintaksno stablo obilazi odozdo naviše.

U tabeli 12.8 data su semantička pravila koja omogućavaju generisanje troadresnog koda za aritmetičke izraze koji sadrže promenljive sa indeksima.

#### Primer 12.7 Međukod za promenljive sa indeksima

Razmotrićemo generisanje troadresnog međukoda za izraz

$$b = c[i] + a[i][j],$$

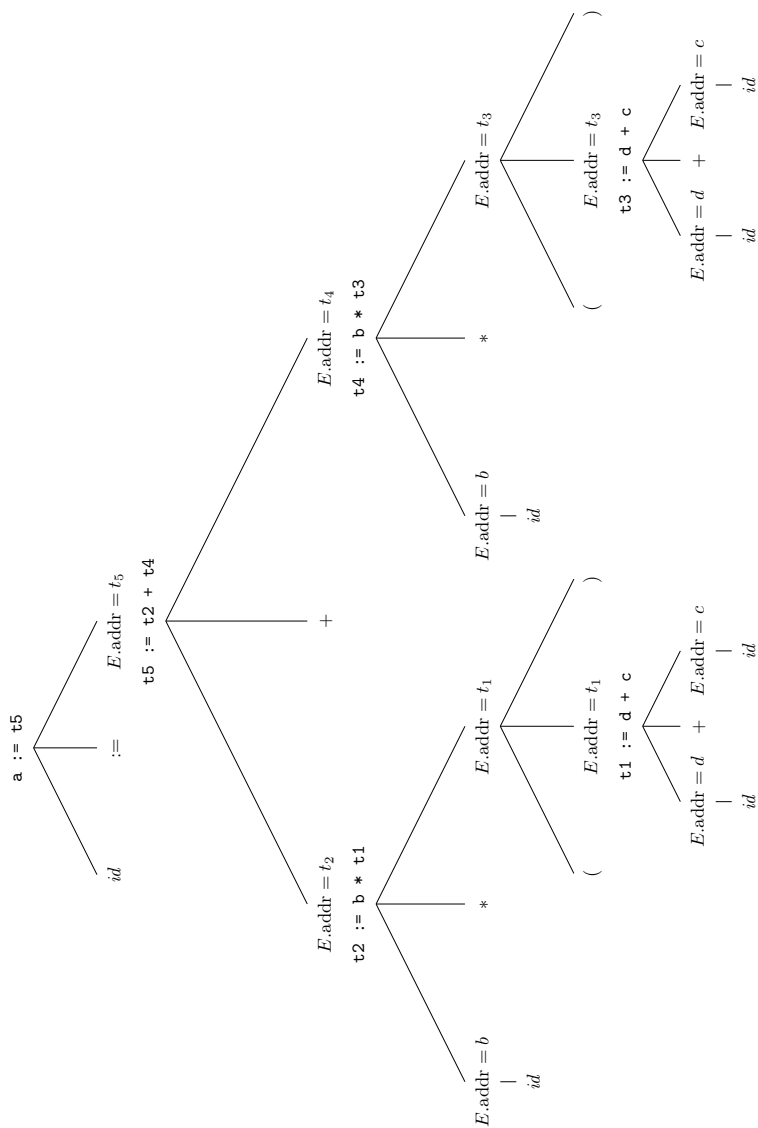
gde je  $a$  celobrojna matrica dimenzija  $4 \times 3$ ,  $c$  celobrojni vektor sa 4 elemenata a  $i$  i  $j$  su celobrojne promenljive. Ako pretpostavimo da je dužina jednog celobrojnog podatka 4 bajta, tada su u tabelu simbola uneti sledeći atributi i njihove vrednosti.

```

a.type = array(4, array(3, integer))
a.type.with = 4 × 3 × 4 = 48
a[].type = array(3, integer)
a[].type.with = 3 × 4 = 12
a[][].type = integer
a[][].type.with = 4

```





SLIKA 12.6: Označeno sintaksko stablo za generisanje troadresnog međukoda izraza iz primera 12.6

TABELA 12.8: Semantička pravila za generisanje troadresnog međukoda za izraze sa indeksima

PRAVILO GRAMATIKE	SEMANTIČKO PRAVILO
(1) $S \rightarrow id := E;$	$\{\text{gen}(\text{top.get}(id.\text{lexeme } ' := ' E.\text{addr}))\}$
(2) $S \rightarrow L := E;$	$\{\text{gen}(L.\text{array.base } '[' L.\text{addr } ']' ' := ' E.\text{addr})\}$
(3) $E \rightarrow E_1 + E_2$	$\{E.\text{addr} = \text{new Temp}()\}$ $\{\text{gen}(E.\text{addr } ' := ' E_1.\text{addr } '+' E_2.\text{addr})\}$
(4) $E \rightarrow id$	$\{E.\text{addr} = \text{top.get}(id.\text{lexeme})\}$
(5) $E \rightarrow L$	$\{E.\text{addr} = \text{new Temp}()\}$ $\{\text{gen}(E.\text{addr } ' := ' L.\text{array.case } '[' L.\text{addr } ']' ')\}$
(6) $L \rightarrow id[E]$	$\{L.\text{array} = \text{top.get}(id.\text{lexeme})\}$ $\{L.\text{type} = L.\text{array.type.elem}\}$ $\{L.\text{addr} = \text{new Temp}()\}$ $\{\text{gen}(L.\text{addr } ' := ' E.\text{addr } '*' L.\text{type.with})\}$
(7) $L \rightarrow L_1[E]$	$\{L.\text{array} = L_1.\text{array}\}$ $\{L.\text{type} = L_1.\text{type.elem}\}$ $\{t = \text{new Temp}()\}$ $\{L.\text{addr} = \text{new Temp}()\}$ $\{\text{gen}(t ' := ' E.\text{addr } '*' L.\text{type.width})\}$ $\{\text{gen}(L.\text{addr } ' := ' L_1.\text{addr } '+' t)\}$

```

c.type = array(4, integer)
c.type.with = 4 × 4 = 16
c[].type = integer
c[].type.with = 4.

```

Sintaksno stablo za izraz

$$b = c[i] + a[i][j]$$

dato je na slici 12.7, a odgovarajuće označeno sintaksno stablo na kome su prikazani svi atributi i vrednosti koje im se dodeljuju, kao i generisani kod dato je na slici 12.8.

Za zadati izraz generiše se sledeći međukod:

```

t1 := i * 4
t2 := c[t1]
t3 := i * 12
t4 := j * t4
t5 := t1 + t2
t6 := a[t5]
t7 := t2 + t6
b := t7

```

## 12.6 Troadresni međukod upravljačkih naredbi

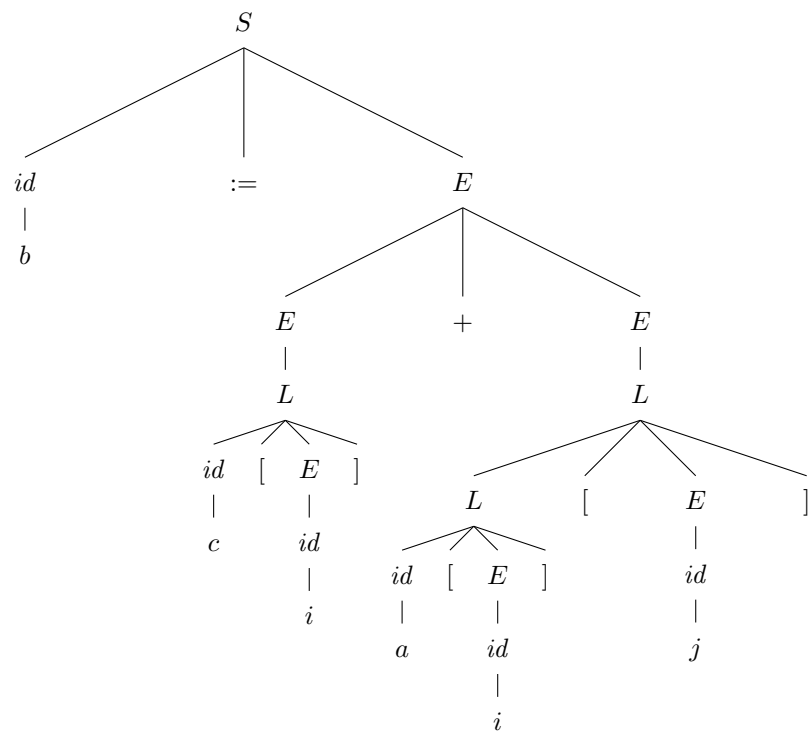
U tabeli 12.9 predstavljena su semantička pravila koja omogućavaju generisanje troadresnog međukoda za upravljačke naredbe. U ovoj tabeli funkcijom newlabel() generišu se nove labele, a funkcijom label(Labela), u kod se ubacuje labela. Atribut *code* pamti određeni deo koda (Bulovog izraza ili naredbe), a atribut *S.next* labelu koja ukazuje na naredbu koja sledi iza naredbe *S*.

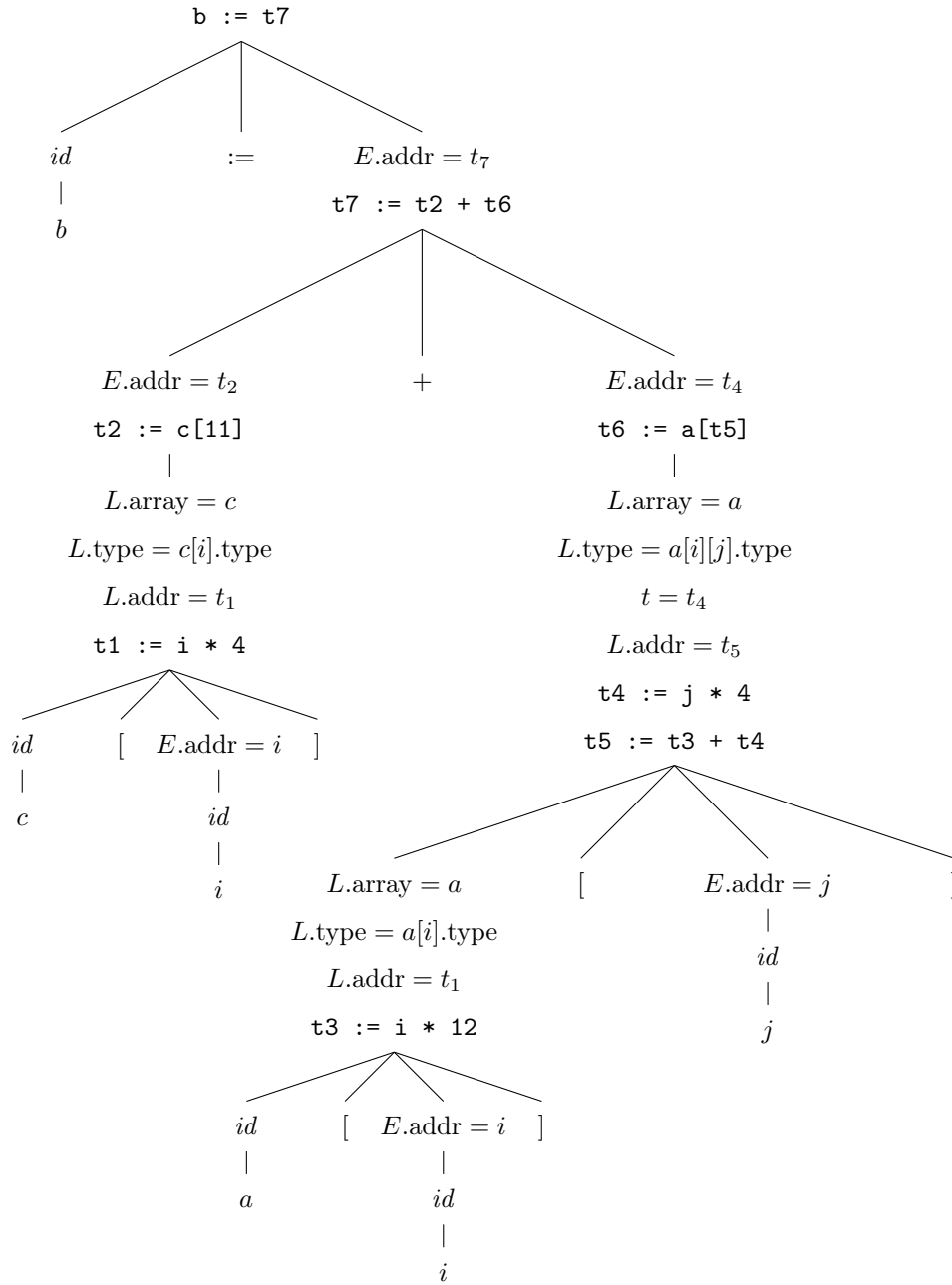
Kako se u okviru upravljačkih naredbi javljaju logički izrazi potrebno je definisati i semantička pravila za generisanje troadresnog međukoda logičkih izraza. U tabeli 12.10 data su ova semantička pravila.

### Primer 12.8 Troadresni kod za *if* naredbu

Razmotrimo sledeću *if* naredbu.

```
if (x < 100 || x > 200 && x != y) x := 0;
```

SLIKA 12.7: Sintaksno stablo za izraz  $b = c[i] + a[i][j]$  iz primera 12.7



SLIKA 12.8: Označeno sintaksno stablo sa slike 12.7 uz primer 12.7

PRAVILO GRAMATIKE	SEMANTIČKE RUTINE
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code; label(S.next)$
$S \rightarrow assign$	$S.code = assign.code$
$S \rightarrow id(B) S_1$	$B.true = ()$ $B.false = S_1.next = S.next$ $S.code = B.code; label(B.true); S_1.code$
$S \rightarrow if(B) S_1 else S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code; label(B.true); S_1.code; gen(goto S.next); label(B.false); S_2.code$
$S \rightarrow while(B) S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin); B.code; label(B.true); S_1.code; gen('goto' begin)$
$S \rightarrow S_1 S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code; label(S_1.next); S_2.code$

TABELA 12.9: Semantička pravila za generisanje troadresnog međukoda za upravljačke naredbe

TABELA 12.10: Semantička pravila za generisanje troadresnog međukoda za Bulove izraze

PRAVILO GRAMATIKE	SEMANTIČKE RUTINE
$B \rightarrow B_1 \parallel B_2$	$B_1.\text{true} = B.\text{true}$ $B_1.\text{false} = \text{newlabel}()$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B_2.\text{false}$ $B.\text{code} = B_1.\text{code}; \text{label}(B_1.\text{false}); B_2.\text{code}$
$B \rightarrow B_1 \&\& B_2$	$B_1.\text{true} = \text{newlabel}()$ $B_1.\text{false} = B.\text{false}$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B_2.\text{false}$ $B.\text{code} = B_1.\text{code}; \text{label}(B_1.\text{true}); B_2.\text{code}$
$B \rightarrow !B_1$	$B_1.\text{true} = B.\text{false}$ $B_1.\text{false} = B.\text{true}$ $B.\text{code} = B_1.\text{code}$
$B \rightarrow E_1 \text{ rel } E_2$	$B.\text{code} = E_1.\text{code}; E_2.\text{code};$ $\text{gen}(\text{'if' } E_1.\text{addr rel } E_2.\text{addr 'goto' } B.\text{true});$ $\text{gen}(\text{'goto' } B.\text{false})$
$B \rightarrow \text{true}$	$B.\text{code} = \text{gen}(\text{'goto' }, B.\text{true})$
$B \rightarrow \text{false}$	$B.\text{code} = \text{gen}(\text{'goto' }, B.\text{false})$

Primenom gramatika iz tabela 12.9 i 12.10 za zadatu naredbu generiše se sintaksno stablo prikazano na slici 12.9. Napomenimo da je operacija  $\&\&$  (*and*) višeg prioriteta od operacije  $\parallel$  (*or*).

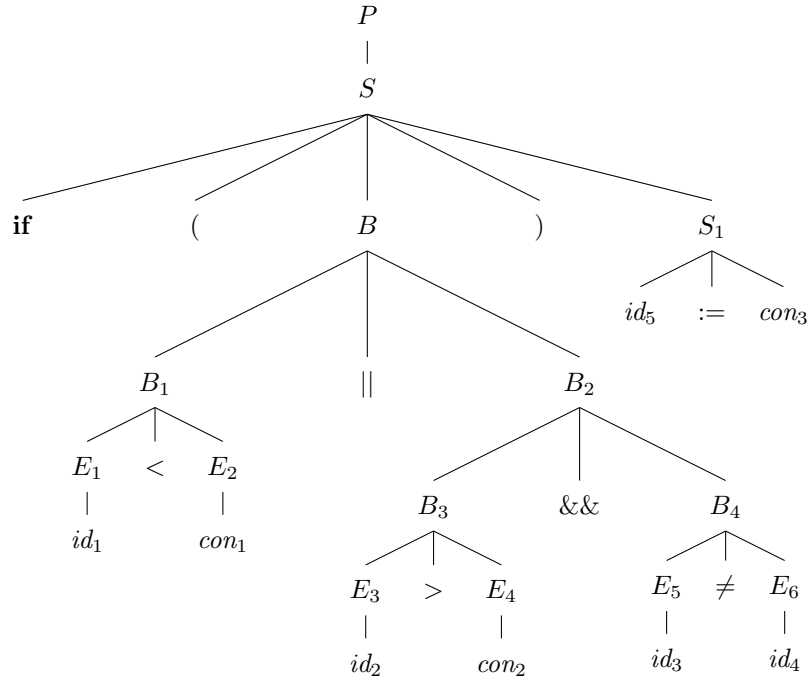
Na osnovu semantičkih pravila iz tabela biće generisane sledeće vrednosti atributa i troadresni međukod prikazan u tabeli 12.11. Čvorovi su navedeni u redosledu koji odgovara obilasku stabla odozdo naviše.

Korišćenjem semantičkih pravila datih u tabeli 12.9 i tabeli 12.10 biće generisan sledeći troadresni međukod.

```

                if  $x < 100$  goto Lab3 goto Lab2
Lab2:   if  $x < 100$  goto Lab1 goto Lab4
Lab1:   if  $x \neq y$  goto Lab3 goto Lab4
Lab3:    $x := 0$ 
Lab4:

```



SLIKA 12.9: Sintaksno stablo za naredbu `if (x < 100 || x > 200 && x != y) x := 0;` iz primera 12.8



TABELA 12.11: Generisanje troadresnog koda za naredbu `if (x < 100 || x > 200 && x != y) x := 0`; iz primera 12.8

ČVOR	ATRIBUT I KOD
$E_1$	$E_1.addr = x$
$E_2$	$E_2.addr = 100$
$B_1$	$B_1.code = \text{if } x < 100 \text{ goto } B_1.code \text{ goto } B_1.false$
$E_3$	$E_3.addr = x$
$E_4$	$E_4.addr = 200$
$B_3$	$B_3.code = \text{if } x > 200 \text{ goto } B_3.code \text{ goto } B_3.false$
$E_5$	$E_5.addr = x$
$E_6$	$E_5.addr = y$
$B_4$	$B_4.code = \text{if } x \neq y \text{ goto } B_4.code \text{ goto } B_4.false$
$B_2$	$B_3.true = \text{Lab1}$
	$B_3.false = B_2.false$
	$B_4.true = B_2.true$
	$B_4.false = B_2.false$
	$B_2.code = \text{if } x > 200 \text{ goto Lab1 goto } B_2.false$
$B$	$\text{Lab1: if } x \neq y \text{ goto } B_2.true \text{ goto } B_2.false$
	$B_1.true = B.true$
	$B_1.false = \text{Lab2}$
	$B_2.true = B.true$
	$B_2.false = B.false$
$S_1$	$B.code = \text{if } x < 100 \text{ goto } B.true \text{ goto Lab2}$
	$\text{Lab2: if } x > 200 \text{ goto Lab1 goto } B.false$
	$\text{Lab1: if } x \neq y \text{ goto } B.true \text{ goto goto } B.false$
	$S_1.code = x := 0$
	$B.true = \text{Lab3}$
$S$	$B.false = S_1.next = S.next$
	$S.code = \text{if } x < 100 \text{ goto Lab3 goto Lab2}$
	$\text{Lab2: if } x < 100 \text{ goto Lab1 goto } S.next$
	$\text{Lab1: if } x \neq y \text{ goto Lab3 goto } .next$
	$\text{Lab3:}$
$P$	$x := 0$
	$S.next = \text{Lab4}$
	$P.code = \text{if } x < 100 \text{ goto Lab3 goto Lab2}$
	$\text{Lab2: if } x < 100 \text{ goto Lab1 goto Lab4}$
	$\text{Lab1: if } x \neq y \text{ goto Lab3 goto Lab4}$
	$\text{Lab3: } x := 0$
	$\text{Lab4:}$

Sledeći primer ilustruje generisanje troadresnog međukoda na primeru jednog funkcijskog potprograma.

**Primer 12.9** Međukod funkcijskog potprograma

Primenom pravila iz tabele 12.9 i tabele 12.10 generisaćemo međukod sledećeg funkcijskog potprograma.

```
function fact(x: integer): integer;
var u: integer;
begin
  if x = 0 then u := 1
    else u := fact(x - 1) * x;
  fact := u
end {fact}
```

Biće generisan sledeći međukod.

```

                                 $t_1 := (x = 0)$ 
                                 $t_2 := \text{not } t_1$ 
                                if  $t_2$  then goto  $L_1$ 
                                 $u := 0$ 
                                goto  $L_2$ 
 $L_1$ :  $t_3 := x - 1$ 
                                param  $t_3$ 
                                 $t_4 = \text{fcall fact}, 1$ 
                                 $u := t_4 * x$ 
 $L_2$ : fret  $u$ 
```

## 12.7 Pitanja

1. Objasniti mesto i namenu međukoda u strukturi kompilatora.
2. Navseti osnovne vrste međukoda.
3. Dati primer apstraktnog sintaksnog stabla.
4. Kako se implementira sintaksno stablo?
5. Navesti semantička pravila za generisanje apstraktnog sintaksnog stabla.
6. Šta je to troadresni međukod?
7. Kako se implementira troadresni međukod?
8. Objasniti strukturu *quadruples* i dati primer.

9. Objasniti strukturu *triples* i dati primer.
10. Objasniti strukturu *indirect triples* i dati primer.
11. Navesti osnovne naredbe troadresnog međukoda.
12. Dati primer semantičkih pravila kojima se generiše troadresni međukod za izraze.
13. Dati primer semantičkih pravila kojima se generiše troadresni međukod za promenljive sa indeksima.
14. Dati primer semantičkih pravila kojima se generiše troadresni kod osnovnih upravljačkih struktura.
15. Dati primer semantičkih pravila kojima se generiše troadresni međukod za Bulove izraze.

## 12.8 Zadaci

1. Generisati apstraktno sintaksno stablo za sledeće aritmetičke izraze.

$$a + b \cdot c + a \cdot d$$

$$b + c \cdot (-a) + d + c \cdot (-a)$$

Generisana stabla prikazati kao dinamičke i kao statičke strukture.

2. Generisati označeno sintaksno stablo za generisanje troadresnog međukoda za izraze iz prethodnog zadatka.
3. Generisati troadresni međukod za izraz iz provog zadatka.
4. Dati *quadrapiles* strukturu za troadresni kod iz prethodnog primera.
5. Korišćenjem gramatike iz tabele 12.8 generisati sintaksno stablo za naredbu dodeljivanja

$$a[i] := b[i][k] + c[i].$$

6. Korišćenjem semantičkih pravila iz tabele 12.8 generisati označeno sintaksno stablo za naredbu dodeljivanja, za slučaj kada je  $b$  celobrojna matrica dimenzija  $5 \times 5$ , a  $a$  i  $c$  celobrojni vektori sa po 5 elemenata.
7. Za naredbu dodeljivanja iz prethodnog zadatka generisati troadresni međukod i predstaviti ga *quadrapiles* strukturom.
8. Korišćenjem gramatike iz tabela 12.9 i 12.10 generisati sintaksno stablo za sledeću naredbu:

**if**  $a \geq b$  **&&**  $a < c$  **then**  $a := b + c$   
**else if**  $c < d$  **then**  $a := d$

9. Korišćenjem semantičkih pravila iz tabela 12.9 i 12.10 generisati označeno sintaksno stablo za *if* naredbu iz prethodnog zadatka.
10. Za **if** naredbu iz prethodnog zadatka generisati troadresni međukod i predstaviti ga *quadrapiles* strukturom.
11. Korišćenjem gramatike iz tabela 12.9 i 12.10 generisati označeno sintaksno stablo za sledeću sekvencu naredbi.

```
sum := 0
while (num != 0)
    sum := sum + a[num]
    num := num - 1;
```

12. Korišćenjem semantičkih pravila iz tabela 12.9 i 12.10 i generisati označeno sintaksno stablo za sekvencu naredbi iz prethodnog zadatka.
13. Za sekvencu naredbi iz prethodnog zadatka generisati troadresni međukod i predstaviti ga *quadrapiles* strukturom.

## Glava 13

# Optimizacija koda

Strategija generisanja koda naredba za naredbom često daje ciljani kod programa koji sadrži redundatne naredbe i neoptimalne konstrukcije. Zbog toga se nakon generisanja prve verzije koda uvek vrši optimizacija čiji je cilj da se smanji broj naredbi i memorijski prostor koji generisani kod zahteva.

Pored ove osnovne optimizacije, koja je praktično obavezna, obično se primenjuje i čitav niz dodatnih optimizacija sa ciljem da se dobije što efikasniji kod. Često se ostavlja mogućnost da se ove dodatne optimizacije uključuju opciono, tako da se kod praktičnog rada najpre nastoji da se dobije funkcionalno ispravan kod, a nakon toga se uključuju raspoložive optimizacije. Osnovno pravilo kod svake optimizacije koda je da ne sme da se naruši ispravnost koda. Kod mora da ostane funkcionalno ispravan posle optimizacije.

### 13.1 Osnovna optimizacija koda (*peephole* optimizacija)

Osnovna optimizacija se obično vrši već na nivou međukoda i najvećim delom je mašinski nezavisna. Nekađ se u osnovnu optimizaciju uključuju i neki koraci kojima se koriste prednosti procesora za koji se generiše kod, odnosno mašinski zavisne transformacije.

Ova optimizacija se obično izvodi tako što više puta prolazi kroz kod i nastoji da se primene jednostavne transformacije kojima se optimizuje kod. Pri tome se transformacije vrše lokalno nad grupom od svega nekoliko naredbi. Naime kroz kod se prolazi tako što se u jednom trenutku posmatra samo mali deo koda i nastoji da se izmeni tako da se smanji broj naredbi, složenije naredbe zamene jednostavnijim ili utiče na memorijski prostor koji kod zahteva.

Zbog same tehnike kojom se izvodi ova transformacija, kod se posmatra kroz uski prozor, procep (*peephole*), ova optimizacija se u literaturi sreće pod imenom *peephole* optimizacija.

Transformacije koda koje se vrše u okviru peephole optimizacije se mogu svrstati u sledećih nekoliko grupa:

- eliminisanje redundantnih naredbi,
- optimizacija toka programa,
- algebarska uprošćenja i
- korišćenje mašinskih idioma.

### 13.1.1 Eliminisanje redundantnih naredbi

Ovom grupom obuhvaćene su transformacije kojima se iz koda izbacuju suvišne naredbe, ili grupe naredbi, koje su obično posledica samog automatizma koji se primenjuje kod generisanja koda.

Najpoznatiji takav primer su parovi naredbi **STORE** i **LOAD**, koji se pojavljuju zato što se jedan korak algoritma za generisanje koda završava naredbom **STORE**, a naredni korak započinje naredbom **LOAD**. Pogledajmo sledeći primer.

```
STORE T1, R1
LOAD  T1, R1
```

Očigledno je da je ovaj par naredbi bezefektan. Naime, prvo se jedna vrednost iz registra  $R_1$  prebacuje u memorijsku lokaciju  $T_1$ , a zatim se ista ta vrednost vraća iz memorijske lokacije u registar  $R_1$  registar procesora.

Očigledno je da se ove dve naredbe mogu izbaciti, a da se time ne naruši funkcionalnost koda.

Međutim, eliminisanje se ne sme izvršiti u slučaju kada je druga naredba označena labelom jer to znači da se u nekim slučajevima do naredbe **LOAD** dolazi tako da joj ne prethodi naredba **STORE**.

```
STORE T1, R1
Lab1: LOAD T1, R1
```

Sličan primer redundantnosti imamo i u sledećem slučaju:

```
MOV R0, A ; (A) => R0
MOV A, R0 ; (R0) => A
```

Druga naredba može da bude izostavljena zato što prilikom kopiranja sadržaja memorijske lokacije  $A$  u registar  $R_0$ , sadržaj te lokacije ostaje neizmenjen i nije potrebno ponovo ga obnavljati.

### 13.1.2 Optimizacija toka

U toku generisanja međukoda često se javljaju skokovi na skokove, skokovi na uslovne skokove ili uslovni skokovi na безусловne skokove. U mnogim takvim slučajevima moguće su redukcije.

Ovakvi slučajevi nastaju kao posledica generisanja međukoda u slučajevima kada se u okviru petlji javljaju *if* naredbe ili druge petlje ili kada u okviru jedne *if* naredbe imamo umetnute *if* naredbe i petlje.

#### Bezuslovni skok na безусловni skok

Na primer, sekvencu naredbi

```
GOTO L1
L1: GOTO L2
```

možemo zameniti naredbom

```
GOTO L2
L1: GOTO L2
```

U ovom slučaju je broj naredbi ostao isti, ali je nova sekvenca naredbi efikasnija zato što se umesto para naredbi `GOTO L1` i `GOTO L2`, izvršava samo naredba `GOTO L2`, dok je naredba sa labelom  $L_1$  ostala samo zbog mogućnosti da postoji skok na ovu naredbu iz nekog drugog dela programa.

#### Uslovni skok na безусловni

Sekvenca naredbi

```
IF a < b GOTO L1
L1: GOTO L2
```

se može zameniti sledećom sekvencom:

```
IF a < b GOTO L2
L1: GOTO L2
```

Objašnjenje je isto kao u prethodnom slučaju, samo što se prvi skok izvršava uslovno – kada je vrednost izraza  $a < b$  jednaka `true`.

#### Bezuslovni skok na uslovni

```
GOTO L1
; ...
L1: IF a < b GOTO L2
L3:
```

```

    ; ...
L2:

```

Navedena sekvenca se može zameniti sledećom:

```

    IF a < b GOTO L2
    GOTO L3
L3:
L2:

```

Broj naredbi je ostao isti, ali je drugi slučaj optimalniji. Dok se u prvom slučaju uvek izvršava `GOTO` i `IF`, u drugom se uvek izvršava `IF`, a `GOTO L3` samo u slučaju kada uslov  $a < b$  nije ispunjen.

### 13.1.3 Nedosegljiv kod

Eliminišu se naredbe koje se u određenom kontekstu neće nikada izvršiti. Na primer, naredba koja sledi iza naredbe bezuslovnog skoka se može izbaciti.

Razmotrimo sledeću sekvencu naredbi u C-u.

```

# define debug 0
if (debug) { /* štampanje poruke o greški */ }

```

Međukod ove naredbe je:

```

    IF debug = 1 GOTO L1
    GOTO L2
L1:    print ;...
L2:

```

Posle eliminisanja `GOTO` na `GOTO`, ova sekvenca dobija oblik:

```

    IF debug /= 1 GOTO L2
    print ;...
L2:

```

Međutim, kako je `debug = 0`, ubacivanjem konstanti u program, što je jedna od faza grerisanja koda, dobija se:

```

    IF 0 /= 1 GOTO L2
    print ;...
L2:

```

Kako je uslov `IF` naredbe uvek ispunjen, uslovna naredba može biti zamenjena bezuslovnom naredbom `GOTO L2`, pa se i sekvenca naredbi za štampanje poruke u ovom slučaju može potpuno eliminisati jer će biti nedosegljiva.



### 13.1.4 Algebarska uprošćenja

Kao rezultat automatskog generisanja koda, u nekim posebnim slučajevima mogu da budu generisane naredbe koje se mogu zameniti jednostavnijim naredbama ili potpuno izbaciti. Na primer, sledeće naredbe mogu da se potpuno izbace zato što se dodavanjem nule ili množenjem jedinicom ne menja vrednost.

```
X = X + 0
X = X * 1
```

Iz istog razloga moguće je uprostiti naredbe

```
X = Y + 0
X = Y * 1
```

i zameniti ih jednostavnijim:

```
X = Y
X = Y
```

### 13.1.5 Direktna redukcija

Sastoji se u tome da se složene operacije zamenjuju jednostavnijima koje se brže izvršavaju. Neke od mogućih transformacija su:

- Stepenovanje se zamenjuje višestrukim množenjem  $X^2 = X \cdot X$
- Množenje i deljenje stepenom dvojke se zamenjuje šift operacijama.
- Deljenje realnom konstantnom se zamenjuje množenjem sa konstantom čija je vrednost jednaka recipročnoj vrednosti te konstante, što je nekad brže.

### 13.1.6 Korišćenje mašinskih idioma

Osnovna optimizacija može da uključuje i neke mašinski zavisne transformacije, kojima se koriste prednosti i specifičnosti asemblerskog jezika mašine za koju se generiše kod. Na taj način mogu da se iskoriste neki posebni oblici adresiranja, neki specifični registri procesora ili neke posebne naredbe ciljnog jezika. Ove transformacije se najčešće odnose na korišćenje:

- samoindeksiranja,
- samoinkrementiranja,
- specifičnih adresiranja,
- specifičnih registara procesora i
- korišćenje steka.

## 13.2 Dodatna optimizacija

U cilju poboljšanja kvaliteta generisanog koda kompilator može da uključi i niz dodatnih transformacija koje se izvršavaju na nivou međukoda ili na nivou samog asemblerskog koda.

Ova optimizacija može da uključuje i neke transformacije koje zahtevaju jako složene analize koda. Ovde će biti dat samo kratak osvrt na osnovne tehnike koje su danas već postale standardne sa naglaskom na to da se na novim tehnikama za optimizaciju još uvek radi i da kompletno proučavanje ovih tehnika izlazi iz okvira ovog predmeta.

Osnovne tehnike optimizacije biće ilustrovane na primeru programa kojim se realizuje *quicksort* algoritam za uređivanje vektora dat u sledećem primeru.

### Primer 13.1 C program *quicksort* algoritma

U nastavku je dat *quicksort* algoritam napisan u C-u.

```
void quicksort(int m, int n) {
    int i, j;
    int v, x;
    if (n <= m) return;
    /* Početak dela koji optimizujemo */
    i = m - 1; j = n; v = a[n];
    while (j) {
        do i = i + 1 ; while (a[i] < v);
        do j = j - 1 ; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    /* Kraj dela koda koji nas interesuje */
    quicksort(m, j); quicksort(i + 1, n);
}
```

Kompilator će na osnovu C koda iz prethodnog priema generisati troadresni međukod prikazan u nastavku.

### Primer 13.2 Međukod *quicksort* programa

U nastavku je dat međukod *quicksort* programa iz primera 13.1.

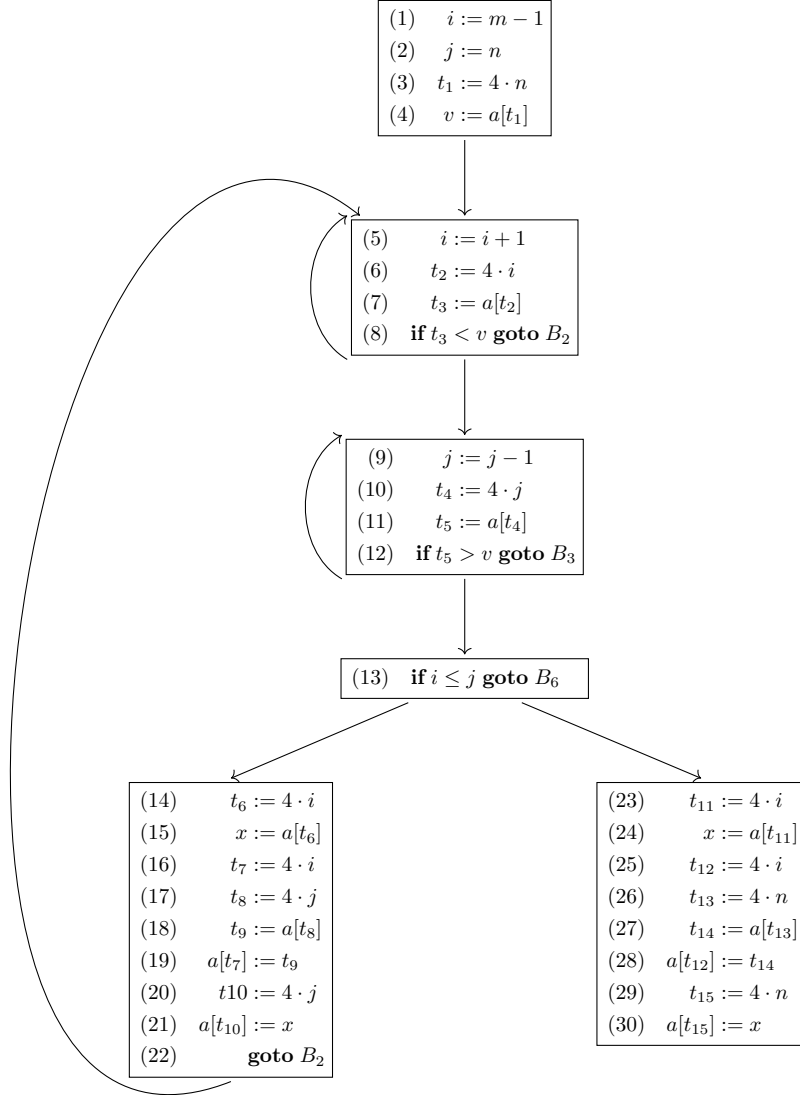
(1)	$i := m - 1$
(2)	$j := n$

```

(3)           $t_1 := 4 \cdot n$ 
(4)           $v := a[t_1]$ 
(5)           $i := i + 1$ 
(6)           $t_2 := 4 \cdot i$ 
(7)           $t_3 := a[t_2]$ 
(8)          if  $t_3 < v$  goto (5)
(9)           $j := j - 1$ 
(10)          $t_4 := 4 \cdot j$ 
(11)          $t_5 := a[t_4]$ 
(12)         if  $t_5 > v$  goto (9)
(13)         if  $i \leq j$  goto (23)
(14)          $t_6 := 4 \cdot i$ 
(15)          $x := a[t_6]$ 
(16)          $t_7 := 4 \cdot i$ 
(17)          $t_8 := 4 \cdot j$ 
(18)          $t_9 := a[t_8]$ 
(19)          $a[t_7] := t_9$ 
(20)          $t_{10} := 4 \cdot j$ 
(21)          $a[t_{10}] := x$ 
(22)         goto (5)
(23)          $t_{11} := 4 \cdot i$ 
(24)          $x := a[t_{11}]$ 
(25)          $t_{12} := 4 \cdot i$ 
(26)          $t_{13} := 4 \cdot n$ 
(27)          $t_{14} := a[t_{13}]$ 
(28)          $a[t_{12}] := t_{14}$ 
(29)          $t_{15} := 4 \cdot n$ 
(30)          $a[t_{15}] := x$ 

```

Prilikom realizacije optimizatora najpre se nastoji da se međukod programa prikaže u vidu grafa u kome su identifikovani i izdvojeni delovi koda koji predstavljaju neke celine. U međukodu iz primera 13.2 se može identifikovati šest različitih blokova. Prvi se odnosi na naredbe za inicijalizaciju koje prethode prvoj petlji. Drugi i treći blok obuhvataju petlje koje se jasno izdvajaju. Četvrti blok je **if** naredba sama za sebe, dok su njena **then** i **else** grana peti i šesti blok.



SLIKA 13.1: Graf toka za međukod iz primera 13.2

### 13.2.1 Zajednički podizrazi

Jedna od osnovnih transformacija koja se primenjuje kod optimizacije koda svodi se na eliminisanje suvišnih temporarnih promenljivih. Nастоји se da se identifikuju promenljive kojima se dodeljuje ista vrednost, i da se umesto uvođenja nove promenljive koristi stara. Ove transformacije se najpre izvršavaju na nivou blokova.

Neki izraz se naziva zajedničkim ako je pre toga već bio izračunat. Na primer, u bloku  $B_5$  generisana je promenljiva  $t_6$  kojoj je dodeljena vrednost  $t_6 := 4 \cdot i$ . Nešto niže u kodu se ista vrednost pamti i kao  $t_7$ . Isto važi i za izraze kojima se dodeljuju vrednosti promenljivima  $t_8$  i  $t_{10}$ . Na svim mestima gde se koristi  $t_7$  može da stoji  $t_6$ , a umesto  $t_{10}$  može da stoji  $t_8$ .

Posle ove transformacije, blok  $B_5$  dobija oblik prikazan na slici 13.2.

```

 $t_6 := 4 \cdot i$ 
 $x := a[t_6]$ 
 $t_8 := 4 \cdot j$ 
 $t_9 := a[t_8]$ 
 $a[t_6] := t_9$ 
 $a[t_8] := x$ 
goto  $B_2$ 

```

SLIKA 13.2: Blok  $B_5$  posle eliminacije zajedničkih podizraza

Sličnu transformaciju moguće je izvršiti i u bloku  $B_6$ . Ovde se najpre uvodi promenljiva  $t_{11}$ , dok se kasnije ista vrednost pamti i kao  $t_{12}$ . Isto važi i za  $t_{13}$  i  $t_{15}$ . Na svim mestima gde se koristi  $t_{12}$  može da stoji  $t_{11}$ , a umesto  $t_{15}$  može da stoji  $t_{13}$ . Blok  $B_6$  posle ovih transformacija prikazan je na slici 13.3.

```

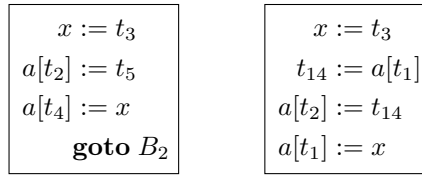
 $t_{11} := 4 \cdot i$ 
 $x := a[t_{11}]$ 
 $t_{13} := 4 \cdot n$ 
 $t_{14} := a[t_{13}]$ 
 $a[t_{11}] := t_{14}$ 
 $a[t_{13}] := x$ 

```

SLIKA 13.3: Blok  $B_6$  posle eliminacije zajedničkih podizraza

### 13.2.2 Zajednički podizrazi na globalnom nivou

Zajednički podizrazi mogu da se pronalaze i na globalnom nivou tako što se posmatraju zavisnosti između blokova i vrednosti uvedene u jednom bloku koriste u narednim. Na primer, u bloku  $B_2$  uvedena je promenljiva  $t_2 := 4 \cdot i$ , pri čemu se ta vrednost zadržava po izlasku iz tog bloka. U bloku  $B_5$  uvodi se  $t_6$  i dodeljuje mu se ista ta vrednost. Praktično,  $t_6$  nije potrebno i može da se eliminiše, a svuda umesto  $t_6$  da se stavi  $t_2$ . Posle sličnih eliminacija  $B_5$  i  $B_6$  dobijaju oblik prikazan na slici 13.4.



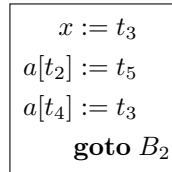
SLIKA 13.4: Blokovi  $B_5$  i  $B_6$  posle eliminacije zajedničkih podizraza na globalnom nivou

### 13.2.3 Prostiranje naredbi za kopiranje

Ukoliko u generisanom kodu postoje naredbe oblika  $f := g$  (naredbe za kopiranje), kojima se vrednost promenljive  $g$  kopira u promenljivu  $f$ . Efekat ove naredbe se može postići tako što se svuda u kodu gde se pojavljuje  $f$  ubacuje  $g$ . Prilikom optimizacije koda, prvo se izvršava ta transformacija, a nakon nje je moguće izbaciti naredbu za kopiranje kao redundantnu.

Ovo je jedna od tipičnih transformacija koja se koristi kod svih optimizatora koda.

U primeru koji razmatramo postoji naredba za kopiranje  $x := t_3$ . Prostiranjem ove naredbe blok  $B_5$  se transformiše u oblik prikazan na slici 13.5.



SLIKA 13.5: Blok  $B_5$  posle prostiranja naredbe za kopiranje  $x := t_3$

Sličnu transformaciju moguće je izvršiti i u bloku  $B_6$ , u odnosu na naredbu za kopiranje  $x := t_3$  (slika 13.6).

$x := t_3$
$t_{14} := a[t_1]$
$a[t_2] := t_{14}$
$a[t_1] := t_3$

SLIKA 13.6: Blok  $B_6$  posle prostiranja naredbe za kopiranje  $x := t_3$ 

### 13.2.4 Eliminisanje neaktivnog koda

Neaktivne naredbe su naredbe koje se nikada neće izvršiti ili nemaju nikakav efekat na program. Takođe mogu da postoje i neaktivne (mrtve) promenljive koje se nikada ne koriste u programu. Ovakve naredbe i promenljive često nastaju kao posledica prostiranja naredbi za kopiranje. U prethodnom primeru to je bio slučaj sa naredbom  $x := t_3$  u blokovima  $B_5$  i  $B_6$ . Ove naredbe se jednostavno mogu izbaciti. Blokovi  $B_5$  i  $B_6$  nakon ove transformacije prikazani su na slici 13.7.

$a[t_2] := t_5$	$t_{14} := a[t_1]$
$a[t_4] := t_2$	$a[t_2] := t_{14}$
<b>goto</b> $B_2$	$a[t_1] := t_3$

SLIKA 13.7: Blokovi  $B_5$  i  $B_6$  posle eliminisanja neaktivnog koda

### 13.2.5 Optimizacija petlji

Optimizacija petlji zauzima važno mesto u optimizaciji programa šire posmatrano, a primenjuje se i kod optimizacije koda na nivou kompilatora. Na primer, vreme izvršavanja programa može se značajno smanjiti ukoliko se smanji broj naredbi unutar petlji čak i ako se pri tome poveća broj naredbi van petlje. Optimizacija petlji može da obuhvati različite transformacije od kojih neke zahtevaju složenu analizu koda radi utvrđivanja zavisnosti između podataka u različitim iteracijama petlje i sl. Kako ove analize neće biti predmet razmatranja u ovom osvrtu na optimizaciju petlji, ovde ćemo se zadržati na nekim jednostavnijim transformacijama koje ne traže složene analize. Na primer, sledeće tehnike se mogu lako primeniti:

- pomeranje koda (*code motion*),
- eliminisanje indukcionih promenljivih i
- direktna redukcija.

### Pomeranje koda (*code motion*)

Transformacija se sastoji u tome da se iz petlje izbacuju izrazi čija se vrednost ne menja u petlji (ne zavisi od broja prolaza kroz petlju). Ako se izračunavanje takvih izraza prebaci u deo za inicijalizaciju petlje onda će se oni izračunavati samo jednom umesto da se izračunavaju prilikom svakog prolaza kroz petlju.

#### Primer 13.3 Pomeranje koda

Zaglavlje *while*-petlje

```
while (i <= limit - 2)
```

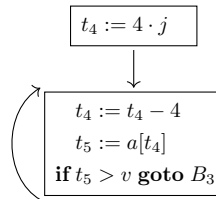
transformiše se u

```
t := limit - 2;
while (i <= t)
```

### Eliminisanje indukcionih promenljivih

Indukcione promenljive su promenljive čija promena izaziva promenu drugih promenljivih. Ove zavisnosti se mogu iskoristiti za direktnu redukciju koda i mogu da imaju poseban efekat kada se primene u petljama, zato što se uprošćavanjem jedne naredbe u petlji postiže efekat kao da je transformisan veliki broj naredbi.

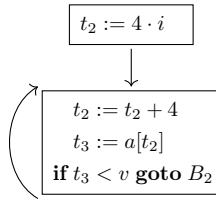
Na primer, u bloku  $B_3$  imamo  $j := j - 1$ , a zatim  $t_4 := 4 \cdot j$ . U svakom prolazu kroz petlju prethodna vrednost promenljive  $j$  se umanjuje za jedan, a to onda utiče na to da se vrednost promenljive  $t_4$  umanjuje za 4. To znači da naredba  $t_4 := 4 \cdot j$  može da bude zamenjena naredbom  $t_4 := t_4 - 4$ , koja je jednostavnija od prethodne; naredba  $j := j - 1$  može da se izbaci, ali je potrebno da se pre ulaza u petlju definiše početna vrednost promenljive  $t_4$ . Posle ovih transformacija, blok  $B_3$  dobija oblik prikazan na slici 13.8.



SLIKA 13.8: Blok  $B_3$  posle eliminisanja indukcionih promenljivih

Slična redukcija može da se izvrši i bloku  $B_2$  gde se javlja naredba  $i := i + 1$ , a zatim  $t_2 := 4 \cdot i$ . Ova naredba može da bude zamenjena naredbom  $t_2 := t_2 + 4$ . Blok  $B_2$  posle eliminisanja indukcionih promenljivih prikazan je na slici 13.9.



SLIKA 13.9: Blok  $B_2$  posle eliminisanja indukcionih promenljivih

Uočimo takođe da se umesto uslova  $i \geq j$ , može koristiti uslov  $t_2 \geq t_4$ , nakon čega se naredbe  $i := i + 1$  u bloku  $B_2$  i naredba  $j := j - 1$  u bloku  $B_3$  mogu sasvim izbaciti.

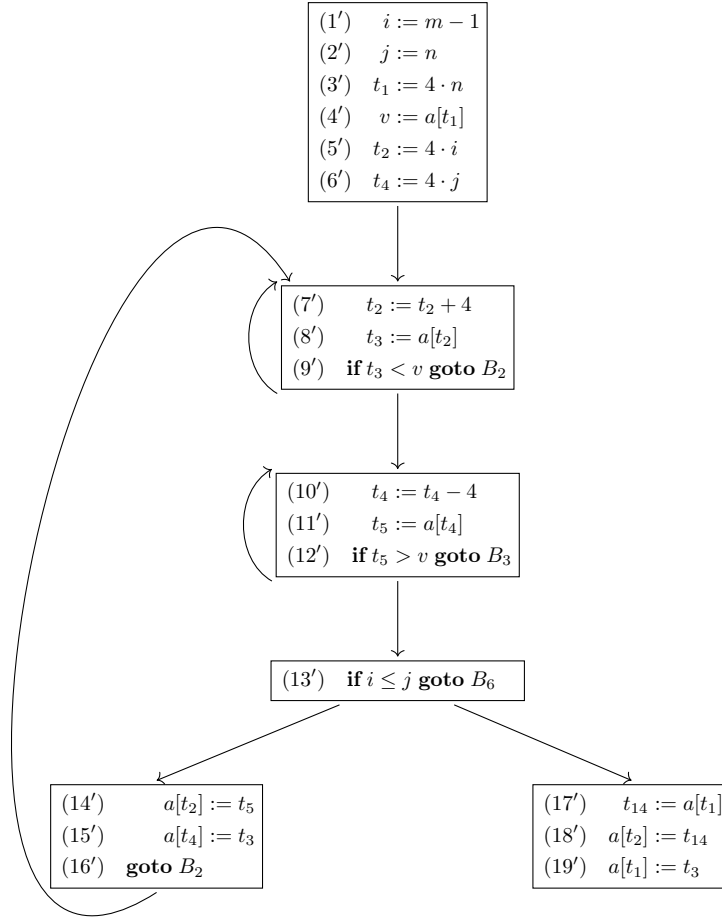
### Direktna redukcija

Direktna redukcija se odnosi na transformacije kojima se složenije (skuplje) naredbe zamenjuju jednostavnijim (jeftinijim) naredbama. U našem primeru, direktna redukcija je primenjena u sprezi sa eliminisanjem indukcionih promenljivih. U okviru petlji u blokovima  $B_2$  i  $B_3$ , naredbe množenja  $t_2 := 4 \cdot i$  i  $t_4 := 4 \cdot j$  zamenjene su jednostavnijim naredbama oduzimanja  $t_4 := t_4 - 4$  i sabiranja  $t_2 := t_2 + 4$ .

Nakon opisanih transformacija međukod programa iz našeg primera dobija oblik prikazan na slici 13.10.

## 13.3 Pitanja

1. Objasniti razloge zbog kojih se vrši osnovna optimizacija koda.
2. Objasniti kako se izvodi osnovna optimizacija koda.
3. Navesti osnovne transformacije koda koje su deo osnovne optimizacije.
4. Šta je to mašinski zavisna a šta mašinski nezavisna optimizacija?
5. Objasniti tehniku eliminisanja redundantnih naredbi.
6. Šta obuhvata optimizacija toka programa?
7. Koja algebarska uprošćenja mogu da se koriste u okviru osnovne optimizacije?
8. Šta se podrazumeva pod direktnom redukcijom.
9. Šta je cilj dodatne optimizacije koda?
10. Objasniti tehniku otkrivanja zajedničkih podnizova.



SLIKA 13.10: Graf toka za međukod iz primera 13.2 nakon optimizacije

11. Kako se koristi prostiranje naredbi za kopiranje naredbi u optimizaciji koda.
12. Objasniti značaj optimizacije petlji za optimizaciju koda i navesti osnovne transformacije koje se pri tome koriste.

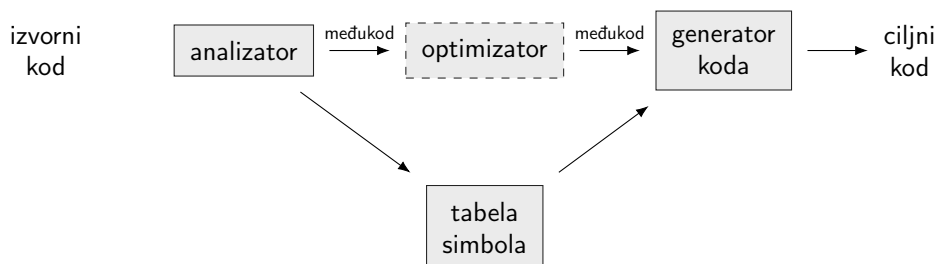


## Glava 14

# Generisanje koda

Faza generisanja koda je praktično poslednja faza u procesu prevođenja jezika. Obično se nadovezuje na generisanje međukoda i koristi informacije sačuvane u tabeli simbola (slika 14.1). Kod nekih kompilatora se nakon generisanja koda vrši njegova optimizacija mada se veliki deo optimizacije vrši na nivou međukoda, o čemu je bilo reči u prethodnom poglavlju.

Osnovni zadatak generatora koda je da generiše ispravan kod pri čemu je bitno da taj kod bude i što je moguće efikasniji.



SLIKA 14.1: Pozicija generatora koda u procesu prevođenja

Generator koda može da bude realizovan tako da generiše direktno mašinski kod sa apsolutnim adresama, mašinski kod sa relativnim adresama ili asemblerski kod koji zatim prolazi kroz proces asembliranja.

Kada se generiše mašinski kod sa apsolutnim adresama, on se direktno smešta u memorijske lokacije i neposredno posle toga izvršava. Ovo je pristup koji se obično primenjuje kod manjih kompilatora koji se realizuju kao studentski projekti.

U slučaju kada se generiše kod sa relativnim adresama, delovi programa (kao što su potprogrami) mogu da se prevode nezavisno, pa da se naknadno povezuju

pomoću linkera i ubacuju u memoriju pomoću *loader*-a.

Kada generator koda generiše asemblerski jezik, sam generator se lakše realizuje – koriste se simboličke naredbe asemblerskog jezika, kao i raspoložive makro-naredbe, ali se tada na fazu generisanja koda nadovezuje faza asembliranja koda. U ovom poglavlju bavićemo se tim pristupom u realizaciji kompilatora.

Da bi se realizovao generator koda, potrebno je poznavati:

- odredišnu mašinu na kojoj će se program izvršavati, tj.
  - ciljni jezik – skup instrukcija koje asemblerski jezik sadrži i načini adresiranja koji se koriste za pristup podacima,
  - arhitekturu procesora – skup registara koje procesor sadrži;
- strategiju organizacije operativne memorije koja je programu dodeljena u toku izvršenja;
- algoritam za generisanje koda, tj. preslikavanje međukoda u ciljni asemblerski jezik.

## 14.1 Odredišna mašina – procesor Intel 8086

Proces generisanja jezika direktno zavisi od ciljnog jezika i nemoguće je u opštem slučaju sagledati sve probleme koji mogu da nastanu. U ovom odeljku biće opisana arhitektura i asemblerski jezik procesora 8086 koji će se koristiti kao odredišna mašina u opisu generatora.

Ovaj procesor sadrži četiri šesnaestobitna regisra opšte namene: AX, BX, CX i DX, pri čemu se može pristupati posebno njihovim bajtovima nižih težina (AL, BL, CL i DL) i viših težina (AH, BH, CH i DH). Uobičajeno je, mada ne obavezno, da se ovi registri koriste na sledeći način:

- AX kao akumulator,
- BX kao bazni registar,
- CX kao brojački registar,
- DX kao registar podataka.

Za pristup podacima u procesorskom steku (o čijoj će ulozi biti reči u odeljku o upravljanju memorijom) koriste se dva pokazivačka registra: SP (*stack pointer*) i BP (*base pointer*).

Za pristup elementima polja koriste se indeksni registri SI (*source index*) i DI (*destination index*).

Procesor sadrži i brojač naredbi IP (*instruction pointer*), statusni registar kao i 4 registra koji ukazuju na početke odgovarajućih segmenata u operativnoj memoriji:

- CS (*code segment*),
- DS (*data segment*),
- SS (*stack segment*) i
- ES (*extra segment*).

### 14.1.1 Osnovni skup instrukcija

U ovom odeljku biće prikazan samo deo instrukcija ovog asemblerskog jezika koje će biti korišćene pri generisanju koda. Sve instrukcije ovog jezika podelićemo u sledeće grupe:

- instrukcija za prenos podataka (tabela 14.1),
- aritmetičke instrukcije (tabela 14.2),
- instrukcije za rad sa procesorskim stekom (tabela 14.3) i
- instrukcije za promenu toka izvršenja programa (tabela 14.4).

TABELA 14.1: Instrukcije za prenos podataka

INSTRUKCIJA	ZNAČENJE
MOV <i>dst</i> , <i>src</i>	Podatak sa adrese <i>src</i> se prenosi na lokaciju <i>dst</i> : $[dst] \leftarrow [src]$

TABELA 14.2: Aritmetičke instrukcije

INSTRUKCIJA	ZNAČENJE
ADD <i>dst</i> , <i>src</i>	$[dst] \leftarrow [dst] + [src]$
SUB <i>dst</i> , <i>src</i>	$[dst] \leftarrow [dst] - [src]$
INC <i>src</i>	$[dst] \leftarrow [dst] + 1$
DEC <i>src</i>	$[dst] \leftarrow [dst] - 1$
MUL <i>src</i>	$[DX : AX] \leftarrow [DX : AX] \cdot [src]$
DIV <i>src</i>	$[AX] \leftarrow [DX : AX] / [src]$ $[DX] \leftarrow [DX : AX] \bmod [src]$

TABELA 14.3: Instrukcije za rad sa procesorskim stekom

INSTRUKCIJA	ZNAČENJE
PUSH <i>src</i>	Stavlja na stek podatak <i>src</i>
POP <i>dst</i>	Izbacuje podataka sa steka i upisuje ga na adresu <i>dst</i>

TABELA 14.4: Instrukcije za promenu toka izvršenja programa

INSTRUKCIJA	ZNAČENJE
JMP <i>adr</i>	Bezulsovni skok na naredbu sa zadatom adresom: $[IP] \leftarrow [adr]$
Jc <i>adr</i>	Uslovni skok na naredbu sa zadatom adresom. Uslov skoka <i>c</i> se obično vezuje za vrednost nekog flega u statusnom registru, pa se ova naredba uglavnom upotrebljava nakon naredbe CMP: <b>if <i>c</i> then</b> $[IP] \leftarrow [adr]$
CALL <i>adr</i>	Poziv potprograma čiji se kod nalazi na zadatoj adresi. U procesorski stek se upisuje tekući sadržaj brojača naredbi, a u brojač naredbi se upisuje data adresa: PUSH IP, $[IP] \leftarrow [adr]$
RET	Povratak na pozivajući modul, adresa koja se nalazi na vrhu procesorskog steka se upisuje u brojač naredbi: POP IP.

### 14.1.2 Načini adresiranja

Polja *src*, *dst* i *adr* u navedenim naredbama ponekad sadrže stvarni podatak sa kojim će se definisana operacija izvoditi, ali češće adresu podatka gde se on nalazi. S obzirom da se podatak može nalaziti u nekom od registara ili u različitim zonama operativne memorije, svaki procesor podržava isvestan skup načina adresiranja podataka. Skup načina adresiranja procesora 8086 koji će se koristiti pri generisanju koda u ovom poglavlju dat je u tabeli 14.5.

U ovom jeziku postoji i ograničenje da kod instrukcija koje imaju dva operanda bar jedan od njih mora da bude u nekom od registara procesora.



TABELA 14.5: Pregled zastupljenih načina adresiranja

NAČIN ADRESIRANJA	FORMAT	PRIMER	ZNAČENJE
neposredno	CONST	100	Vrednost operanda je navedena konstanta.
direktno memorijsko	NAME ili [CONST]	$X$ ili [0100]	Vrednost operanda je sadržaj memorijske lokacije čija je adresa data simboličkim imenom ili konstantom.
direktno registarsko	REG	AX	Vrednost operanda je sadržaj datog registra.
indirektno registarsko	[REG]	[BX]	Vrednost operanda je u memorijskoj lokaciji čija se adresa nalazi u datom registru.
bazno	[REG+CONST]	[BP + 4]	Bazni registar čuva početnu adresu nekog bloka, a stvarna adresa operanda se dobija tako što se na sadržaj datog baznog registra doda vrednost navedene konstante.
indeksno	NAME[REG] ili CONST[REG]	$X[SI]$ ili 25h[DI]	Konstanta ili simboličko ime predstavlja početnu adresu bloka, a u registru je zapamćen pomeraj u odnosu na tu početnu adresu. Adresa operanda se opet dobija sabiranjem početne adrese i pomeraja.

## 14.2 Upravljanje memorijom u toku izvršenja programa

Zadaci ove komponente kompilatora su sledeći.

- Organizacija memorije u toku izvršenja programa – što podrazumeva određivanje gde će se u memoriji smestiti kod programa, kako će se memorisati podaci različitih tipova, gde će se u memoriji smestiti podaci različitih klasa memorije...
- Pristup podacima smeštenim u različitim zonama memorije,
- Pristup podacima definisanim u nekom spoljašnjem opsegu (npr. kod ugnježenih potprograma obezbeđivanje pristupa podacima definisanim u spoljašnjim potprogramima),
- Promena sadržaja memorije prilikom poziva potprograma i povratka na pozivajući modul, tj. prevod poziva potprograma i povratka na pozivajući modul.

### 14.2.1 Organizacije memorije

Operativni sistem dodeljuje neki deo memorije programu pri pokretanju njegovog izvršenja. Zavisno od toga da li se sadržaj memorije dodeljen programu menja u toku izvršenja programa ili ne, viši programski jezici se dele na:

- statičke (eng. *static*),
- dinamičke (eng. *dynamic*) i
- polustatičke (eng. *semistatic*).

Kod statičkih jezika se pre početka izvršenja programa u memoriju učitava kompletan kod programa i rezerviša prostor za sve podatke koji se u programu koriste i u toku izvršenja programa se organizacija memorije ne menja. U ovu grupu jezika spadaju jezici Fortran 77, Cobol i dr. Koncept organizacije memorije koji se koristi kod statičkih jezika je poznat kao statička alokacija memorije.

Kod dinamičkih jezika se u toku samog izvršenja programa mogu dodavati novi delovi koda, rezervisati prostor za nove podatke i izbacivati iz memorije podaci koji se više ne koriste. Ovoj grupi pripadaju uglavnom skript jezici kao što su: PHP, Perl, Prolog, Python, Ruby, JavaScript ali i jezici opšte namene poput Java i C#-a (kod kojih se dodavanje novih delova koda vrši korišćenjem refleksije). Kod ovih programskih jezika se koristi takozvana dinamička alokacija memorije.

Koncept polustatičkih jezika je nešto između statičkih i dinamičkih. Kod njih se kod programa u toku izvršenja programa ne menja i postoje i takozvani statički podaci koji se nalaze u memoriji za sve vreme izvršenja programa, ali postoje i takozvani dinamički podaci za koje se rezerviša prostor u toku izvršenja

programa i koji se uklanjaju iz memorije kada se više ne koriste. Kod ovakvih jezika se memorija dodeljena programu deli na statičku zonu (u koju se smešta kod programa i statički podaci) i dinamičku zonu u koju se smeštaju dinamički podaci. U ovu grupu jezika spadaju Pascal, C, C++...

Kod polustatičkih jezika se i statička i dinamička zona dalje dele na po dva segmenta. Jedan segment statičke zone se koristi za smeštaj koda programa, a drugi za smeštanje statičkih podataka. Jedan deo dinamičke zone memorije se organizuje u vidu steka i služi za smeštanje aktivacionih slogova potprograma (o čemu će biti reči kasnije) a drugi deo (*heap*) se koristi za smeštanje dinamičkih podataka (slika 14.2). Ovakav koncept organizacije memorije se naziva stek alokacija memorije.



SLIKA 14.2: Organizacija memorije kod polustatičkih jezika (stek alokacija)

Veličina generisanog koda je poznata u toku prevođenja programa, a takođe može da se izračuna i memorijski prostor potreban za smeštaj statičkih podataka. Ove informacije kompilator može da iskoristi i da za kod i podatke odredi memorijske lokacije u koje će biti smešteni, tj. odredi njihove određene apsolutne ili relativne adrese.

Veličina steka i dinamičkog dela memorije može da se menja u toku izvršavanja programa. Zbog toga se memorija organizuje tako da neiskorišćeni deo bude između ova dva dela (kao što je prikazano na slici 14.2) tako da u nekom trenutku može procesorski stek da zauzima veći deo dinamičke zone memorije, a u drugom dinamički podaci.

### 14.2.2 Aktivacioni slogovi

Za svaki poziv potprograma kompilator obično generiše jedan aktivacioni slog u kome čuva sve informacije potrebne tom pozivu potprograma. U opštem slu-

čaju aktivacioni slog ima strukturu prikazanu na slici 14.3. Međutim, struktura aktivacionog sloga nije ista kod svih jezika niti kod svih kompilatora za jedan jezik. Ona se obično prilagođava potrebama kompilatora.

rezultat programa
stvarni parametri
link prema aktivacionom slogu pozivajućeg modula
link prema podacima pozivajućeg modula
stanje procesora u trenutku poziva
lokalni podaci
privremeni podaci

SLIKA 14.3: Struktura aktivacionog sloga

Namena polja u aktivacionom slogu je sledeća:

1. Na početku sloga smešta se vrednost koju potprogram vraća glavnom programu. Ova vrednost se zbog efikasnosti može da vraća i kroz neki od registra procesora.
2. Polje namenjeno stvarnim parametrima se koristi od strane glavnog programa da prenese parametre potprogramu koji odgovaraju konkretnom pozivu potprograma.
3. Opciono polje sa upravljačkim linkovima se koristi da se obezbedi veza sa aktivacionim slogom modula iz kojeg je pozvan potprogram.
4. Opciono polje sa linkovima za pristup podacima postoji kod jezika koji dozvoljavaju da se koriste nelokalni podaci, podaci iz aktivacionih slogova drugih potprograma.
5. Deo aktivacionog sloga namenjen čuvanju statusa procesora služi da se u njemu memorišu vrednosti registara procesora koje su zatečene u trenutku poziva potprograma. Ovde se obično smešta vrednost brojača neredbi i registara procesora opšte namene.
6. Polje namenjeno lokalnim podacima služi za smeštaj podataka koji pripadaju samo potprogramu, definisani su u potprogramu.
7. Polje namenjeno privremenim podacima služi za memorisanje podataka koje generiše kompilator kao pomoćne lokacije u koje smešta međurezultate.

## 14.3 Statička alokacija memorije

Kod statičkih jezika se u toku kompiliranja programa generiše po jedan aktivacioni slog za svaki potprogram tako da se isti aktivacioni slog koristi kod svakog poziva potprograma. Ovaj aktivacioni slog se smešta u statički deo memorije. U ovom slučaju nema mogućnosti za rekurzivne pozive potprograma zato što bi svaki naredni poziv potprograma prebrisao sadržaj aktivacionog sloga koji odgovara prethodnom pozivu koji još nije završen.

### Primer 14.1

Uzmimo kao primer kod programa CONSUME i potprograma PRODUCE, napisanih u programskom jeziku FORTRAN.

```

PROGRAM CONSUME
  CHARACTER * 50 BAF
  INTEGER NEXT
  CHARACTER C, PRODUCE
  DATA NEXT /1/, BUF /'  '/
6  C = PRODUCE ( )
  BUF(NEXT:NEXT) = C
  NEXT = NEXT + 1
  IF (C .NE. ' ') GOTO 6
  WRITE (*, '(A)') BUF
END

CHARACTER FUNCTION PRODUCE ( )
  CHARACTER * 80 BUFFER
  INTEGER NEXT
  SAVE BUFFER, NEXT
  DATA NEXT /81/
  IF (NEXT .GT. 80 ) THEN
    READ (*, '(A)') BUFFER
    NEXT = 1
  END IF
  PRODUCE = BUFFER(NEXT:NEXT)
  NEXT = NEXT + 1
END

```

Kako su veličina memorijskog prostora potrebnog za smeštaj koda programa i potprograma, kao i aktivacionih slogova za ova dva modula poznati u fazi kompiliranja programa raspodela memorije u ovom slučaju će biti kao što je prikazano na slici 14.4.

aktivacioni slog za PRODUCE
aktivacioni slog za CONSUME
kod funkcije PRODUCE
kod programa CONSUME

SLIKA 14.4: Statička alokacija memorije

## 14.4 Alokacija memorije pomoću steka

Ako programski jezik dozvoljava rekurzivne pozive potprograma onda nije moguće koristiti statičku alokaciju memorije. Naime u ovom slučaju u jednom trenutku može da postoji više aktiviranih instanci potprograma i za svaki takav poziv potreban je poseban aktivacioni slog.

Jedna od tehnika koju je u ovom slučaju moguće koristiti je alokacija memorije pomoću steka u okviru koje se prilikom svakog poziva potprograma kreira novi aktivacioni slog i smešta na stek. Kada se izvršavanje određene instance potprograma završi odgovarajući aktivacioni slog se skida sa steka. U jednom trenutku u steku može da se nalazi više instanci aktivacionog sloga jednog potprograma, kao i više različitih aktivacionih slogova. U vreme kompiliranja programa zna se samo veličina aktivacionog sloga ali ne i dubina rekurzije (koliko će aktivacionih slogova jednog potprograma u nekom trenutku biti u steku).

### Primer 14.2

Uzmimo kao primer rekurzivnu proceduru `quicksort` kojom se uređuje vektor brojeva, čiji je kod dat u nastavku.

```
void quicksort(int m, int n)
{
    if (n > m)
    {
        i = partition(m,n);
        quicksort(m, i - 1);
        quicksort(i + 1, n);
    }
}

int a[11];
void main ()
{
    a[0] = -9999;
    a[10] = 9999;
    readarray();
}
```

```
quicksort(1,9);
}
```

Uočimo da se u datoj proceduri poziva potprogram `partition` kao i sam potprogram `quicksort` rekurzivno i to dvostrukom rekurzijom. U glavnom programu pre poziva potprograma `quicksort` poziva se i potprogram `readarray`. Redosled poziva potprograma najbolje se može prikazati aktivacionim stablom koje je za dati primer prikazano na slici 14.5.

Svaki čvor u aktivacionom stablu sa slike 14.5 predstavlja jedan poziv potprograma. Pozivi potprograma koji se izvrše iz jedne instance potprograma predstavljani su čvorovima do kojih vode potezi iz čvora koji odgovara toj instanci. Pri tome redosled poziva odgovara redosledu čvorova posmatrano sleva udesno.

Na slici 14.6 je prikazan redosled ubacivanja aktivacionih slogova u stek koji odgovara prikazanom delu aktivacionog stabla za program iz primera 14.2.

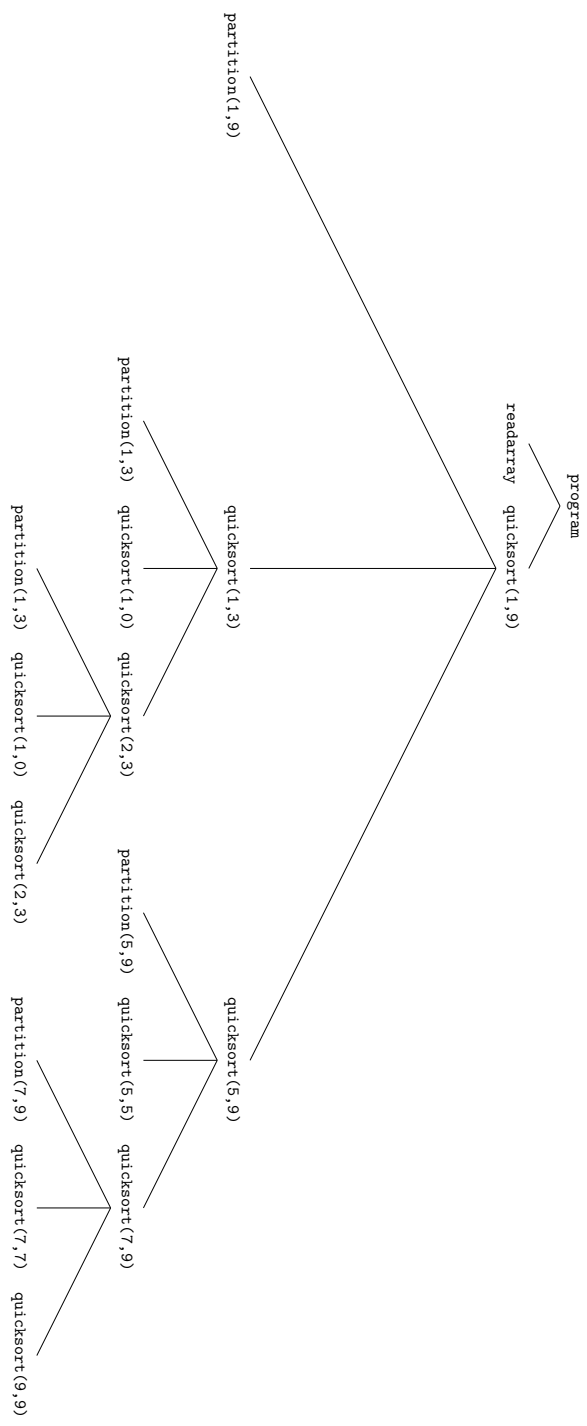
## 14.5 Dinamička alokacija memorije

Kod dinamičkih jezika se i kod i svi podaci (pa i aktivacioni slogovi) smeštaju u dinamičku memoriju (*heap*) i ne izbacuju se iz memorije odmah po završetku potprograma kao kod stek alokacije. Kod dinamičkih jezika, postoji posebna komponenta koja se povremeno aktivira i koja iz memorije briše sve podatke koji se više ne koriste (odnosno na koji se ne referencira ni jedna aktivna promenljiva u kodu) i koja se popularno zove *garbage collector*. Na primer, u slučaju izvršenja programa iz primera 14.2, u trenutku prvog poziva potprograma `quicksort`, u memoriji bi bio sačuvan i aktivacioni slog potprograma `readarray` sve dok se ne aktivira *garbage collector*. Dakle, u *heap*-u bi se našli istovremeno i aktivacioni slog potprograma čije je izvršenje već završeno i potprograma koji se trenutno izvršava, kao što je to prikazano na slici 14.7.

### 14.5.1 Generisanje koda za poziv potprograma

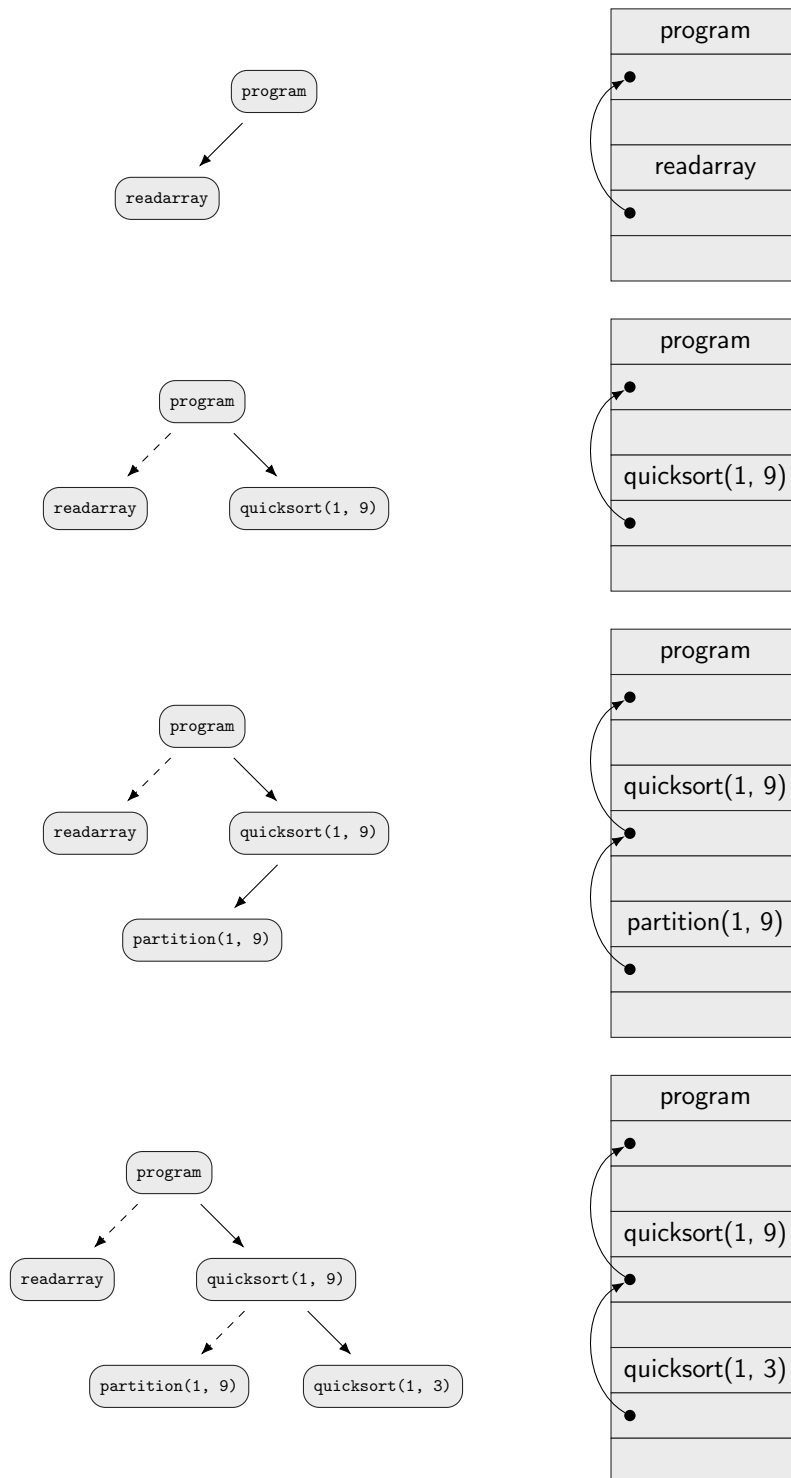
U ovom poglavlju biće prikazan način generisanja koda pri pozivu potprograma i pri povratku iz potprograma za asemblerski jezik procesora Intel 8086. Rečeno je već da ovaj procesor podržava alokaciju memorije pomoću steka pa će u ovom poglavlju biti pokazano prevođenje poziva potprograma kod polustatičkih jezika u ovaj asemblerski jezik.

Dakle, prilikom poziva potprograma se generiše aktivacioni slog i smešta na stek. Rečeno je već, da ovaj asemblerski jezik koristi bazni pokazivač za pristup podacima u steku. Dakle taj bazni pokazivač uvek ukazuje na neku lokaciju aktivacionog sloga poslednjeg pozvanog potprograma, tj. potprograma koji se trenutno izvršava. Kad se izvršenje aktuelnog potprograma završi, bazni pokazivač treba da se vrati da ukazuje na aktivacioni slog pozivajućeg modula.

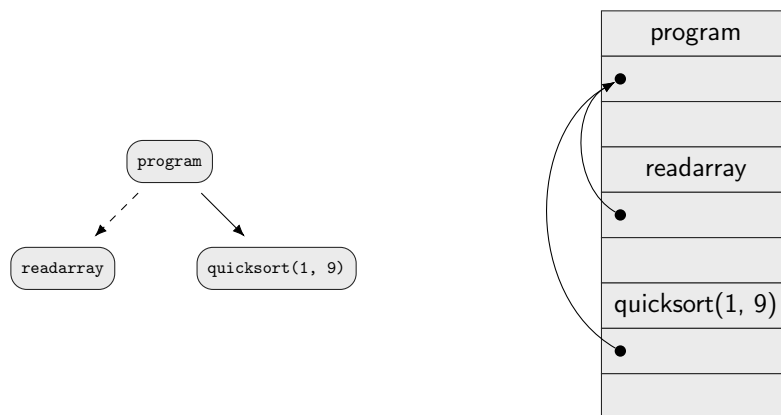


Slika 14.5: Aktivaciono stablo za program iz primera 14.2





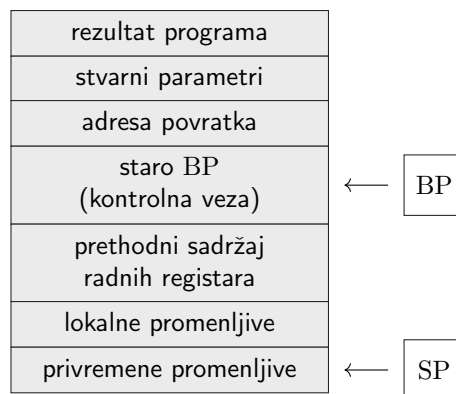
SLIKA 14.6: Dinamika ubacivanja aktivacionih slogova u stek za program iz primera 14.2

SLIKA 14.7: Dinamika ubacivanja aktivacionih slogova u *heap* za program iz primera 14.2

Dakle, stara vrednost baznog pokazivača mora biti, takođe, zapamćena u steku i lokacija gde se ona čuva se naziva kontrolnom vezom. Upravo je i to lokacija na koju ukazuje bazni pokazivač za sve vreme izvršenja potprograma. U skladu sa tim, struktura aktivacionog sloga koja će se koristiti u ovom poglavlju je prikazana na slici 14.8.

U trenutku poziva potprograma izvršava se skup instrukcija za generisanje aktivacionog sloga i njegovo smeštanje u stek. Ovaj skup instrukcija se naziva sekvencom poziva.

U trenutku povratka iz potprograma izvršava se skup instrukcija za izbacivanje aktivacionog sloga iz steka. Ovaj skup instrukcija se naziva sekvencom povratka.



SLIKA 14.8: Aktivacioni slog koji se koristi u jeziku 8086

### 14.5.2 Generisanje sekvence poziva

Iz strukture aktivacionog sloga se vidi da je izvestan skup podataka koji treba da se upišu u aktivacioni slog poznat pozivajućem potprogramu, dok je drugi skup podataka koji se nalazi u aktivacionom slogu poznat pozvanom modulu. Zato će sekvenca poziva biti podeljena na dva dela: prvi deo je onaj koji treba da se generiše u pozivajućem modulu, a drugi deo se generiše na početak pozvanog modula.

Deo sekvence poziva koji se generiše u pozivajućem potprogramu.

1. **Rezervacija prostora u steku za smeštanje rezultata.** Pošto stek raste od najviših memorijskih adresa, prema nižim (slika 14.2), rezervacija prostora se svodi na umanjeње vrednosti pokazivača steka za veličinu memorijskog prostora potrebnog da se smesti povratna vrednost, odnosno, rezervacija prostora za rezultat se vrši sledećom naredbom.

SUB SP, size

Pošto se u ovom asemblerskom jeziku adresiraju bajtovi, a celobrojni podaci zauzimaju jednu reč, rezervacija prostora za rezultat celobrojnog tipa bi bila sledeća.

SUB SP, 2

Inače, kod ovog asemblerskog jezika je uobičajeno da se rezultati celobrojnog tipa ili tipa pokazivača (koji su veličine jedne memorijske reči) vraćaju kroz registar CX.

2. **Smeštanje u stek stvarnih parametara potprograma.** Napisano je pravilo da se parametri u stek upisuju u obrnutom redosledu od onog kojim su oni navedeni u pozivu potprograma. Jedino o čemu treba voditi računa je da se naredbom PUSH u stek smešta jedna memorijska reč. To znači da ako je vrednost parametra zapamćena u više memorijskih reči, potrebno je generisati više PUSH naredbi za njegov upis u stek.
3. **Izvršenje CALL naredbe.** CALL naredba automatski stavlja na stek trenutni sadržaj brojača naredbi IP, odnosno adresu povratka.

Deo sekvence poziva koji se generiše u pozivajućem potprogramu je sledeći.

1. **Smeštanje u stek stare vrednosti baznog pokazivača.** Za ovu akciju treba generisati naredbu:

PUSH BP

2. **Definisanje nove vrednosti baznog pokazivača.** Pošto bazni pokazivač treba da ukazuje na lokaciju gde je zapamćena njegova prethodna vrednost, a nakon izvršenja prethodno generisane instrukcije ona će biti na vrhu steka, baznom pokazivaču treba dodeliti trenutnu vrednost pokazivača steka, odnosno generisati instrukciju:

MOV BP, SP

3. **Smeštanje u stek stanja procesora.** Ovaj korak nije obavezan, ali ukoliko se ne generiše skup instrukcija za pamćenje stanja procesora, u pozivajućem modulu se nakon poziva potprograma ne smeju koristiti vrednosti zapamćene u registrima pre poziva. Zato je preporučivo da se ove instrukcije generišu.

Stanje procesora pre poziva je definisano sadržajem registara opšte namene i indeksnih registara. Dakle, treba generisati skup instrukcija za njihovo smeštanje u stek:

PUSH AX  
 PUSH BX  
 PUSH CX  
 PUSH DX  
 PUSH SI  
 PUSH DI

4. **Rezervacija prostora za lokalne promenljive.** U fazi prevođenja je poznata veličina memorijskog prostora potrebna za smeštanje lokalnih promenljivih definisanih na početku potprograma, pa se rezervacija prostora vrši jednostavnim umanjavanjem vrednosti pokazivača steka za tu vrednost, odnosno za rezervisanje prostora za lokalne promenljive treba generisati naredbu:

SUB SP, size

### 14.5.3 Generisanje sekvence povratka

Prilikom povratka u pozivajući modul, deo aktivacionog sloga koji je kreirao pozvani modul, pozvani modul treba i da ukloni sa steka, a deo koji je kreirao pozivajući modul, pozivajući modul treba i da ukloni. Dakle i sekvenca povratka se opet deli na dva dela: onaj koji izvršava pozvani potprogram i onaj koji izvršava pozivajući modul.

Kako se podaci sa steka skidaju u obrnutom redosledu u odnosu na ono kako su smešteni, biće najpre prikazana sekvenca instrukcija koja treba da bude generisana u pozvanom potprogramu:

1. **Upis rezultata u predviđenu memorijsku lokaciju u steku.**
2. **Izbacivanje iz steka lokalnih promenljivih.** Jednostavnim povećanjem pokazivača steka za veličinu memorijskog prostora potrebnog za smeštanje lokalnih podataka, taj deo memorijskog prostora je oslobođen i spreman za upis novih podataka. Znači, naredba koju treba generisati je:

ADD SP, size

3. **Preuzimanje iz steka starog stanja procesora.** Za preuzimanje starih vrednosti radnih registara treba generisati skup instrukcija:

```
POP DI  
POP SI  
POP DX  
POP CX  
POP BX  
POP AX
```

4. **Preuzimanje iz steka stare vrednosti baznog pokazivača.** Za ovu akciju treba generisati naredbu:

```
POP BP
```

5. **Izvršenje return naredbe.** Return naredba automatski izbacuje iz magacina povratnu adresu i smešta je u brojač naredbi IP. Dakle, dovoljno je generisati samo naredbu:

```
RET
```

Pozivajući modul treba iz steka da izbaciti stvarne parametre i rezultat potprograma, tj. deo sekvence povratka koja se generiše u pozivajućem modulu sadržaće sledeće akcije.

1. **Izbacivanje iz steka stvarnih parametara.** Kao i kod izbacivanja lokalnih promenljivih i ovde je dovoljno samo povećati vrednost pokazivača steka za vrednost koja predstavlja veličinu memorijskog prostora koji je korišćen za smeštanje stvarnih parametara. Generisana instrukcija biće:

```
ADD SP, size
```

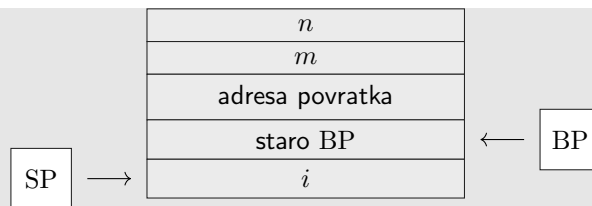
2. **Preuzimanje rezultata** Instrukcija koju treba generisati je:

```
POP destination
```

destination je adresa na koju treba smestiti rezultat potprograma.

### Primer 14.3

Pokazaćemo kako će izgledati generisana sekvenca poziva i sekvenca povratka za proceduru **quicksort** iz primera 14.2. Da bismo ispravno generisali te sekvence, najpre treba definisati kako će izgledati aktivacioni slog ove procedure. Zbog jednostavnosti rešenja u aktivacionom slogu se neće pamtit stari stanje procesora. Aktivacioni slog procedure je dat na sledećoj slici.



Pretpostavimo da je procedura pozvana na sledeći način:

```
quicksort(1, 9);
```

U pozivajućem modulu, u tački poziva ove funkcije generisaće se sledeće instrukcije.

```
; DEO SEKVENCE POZIVA U POZIVAJUĆEM MODULU
; potprogram je opšteg tipa pa se ne rezerviše prostor
; za rezultat

; smeštanje stvarnih parametara na stek:
MOV AX, 9
PUSH AX
MOV AX, 1
PUSH AX

; prelaz na potprogram
CALL QUICKSORT

; DEO SEKVENCE POVRATKA U POZIVAJUĆEM MODULU
; izbacivanje stvarnih parametara iz steka
ADD SP, 4

; PROCEDURA QUICKSORT
; DEO SEKVENCE POZIVA
; smeštanje u stek stare vrednosti baznog pokazivača
PUSH BP
; definisanje nove vrednosti baznog pokazivača
MOV BP, SP
; rezervacija prostora za celobrojnu lokalnu promenljivu i
SUB SP, 2

; TELO METODE
;...

; DEO SEKVENCE POVRATKA U POZVANOM POTPROGRAMU
; izbacivanje lokalnih promenljivih iz steka
ADD SP, 2
```

```
; preuzimanje stare vrednosti baznog pokazivača  
POP BP  
; vraćanje u pozivajući modul  
RET
```

#### 14.5.4 Smeštanje promenljivih i pristup promenljivama u različitim delovima memorije

Kod većine polustatičkih programskih jezika promenljive se u memoriji smeštaju na sledeći način:

- globalne promenljive se smeštaju u statičkoj zoni memorije;
- parametri programa i lokalne promenljive se smeštaju u procesorskom steku;
- dinamičke promenljive (promenljive za koje programer rezerviše prostor onda kada su mu potrebne) se smeštaju u *heap*-u.

Programski jezik C, na primer, ima dodatnu mogućnost da programer eksplicitnim definisanjem klase memorije za promenljive definiše gde će ona biti smeštena. Lokalne promenljive u ovom programskom jeziku mogu imati klasu memorije:

- **static** – takve promenljive se smeštaju u statičku zonu memorije jer se njihova vrednost definisana u jednom pozivu potprograma može koristiti prilikom narednog poziva;
- **auto** – podrazumevana klasa memorije, takve promenljive se smeštaju u procesorski stek;
- **register** – takve promenljive se smeštaju u neki registar procesora (ako ima slobodnih registara, ako ne, tretiraju se na isti način kao i promenljive sa klasom memorije *auto*).

Prevodilac je taj koji će predvideti memorijske lokacije za smeštanje promenljivih i pri pojavi određene promenljive u kodu znati kako da im pristupi, odnosno koje će adresiranje koristiti za pristup promenljivama. U slučaju asemblerskog jezika 8086, pristup podacima se vrši na sledeći način:

- promenljivama smeštenim u statičkoj zoni memorije se pristupa korišćenjem direktnog memorijskog adresiranja,
- promenljivama smeštenim u procesorskom steku se pristupa korišćenjem baznog adresiranja,
- promenljivama smeštenim u registrima se pristupa korišćenjem direktnog registarskog adresiranja,
- pristup promenljivama zapamćenim u *heap*-u se obično pristupa u dva koraka:

- najpre se adresa podatka smesti u neki registar procesora (obično BX, ali ne obavezno),
- zatim se indirektnim registarskim adresiranjem pristupa samom podatku.

#### Primer 14.4

Neka se u nekoj C funkciji nalazi sledeći skup naredbi:

```
register int i;
static int j;
int a;
static int *b;
i = j;
a = *b;
```

Zadatak kompilatora je da rezerviše prostor za promenljive definisane u ovoj funkciji, a zatim i da prevede navedene naredbe dodele.

- Promenljiva *i* treba da bude smeštana u nekom registru (neka je to registar AI).
- Promenljiva *j* je u delu za statičke podatke čija je adresa određena u fazi prevođenja i zapamćena u tabeli simbola. Neka je to adresa 100.
- Promenljiva *a* je zapamćena u steku (pošto je to prva lokalna promenljiva sa klasom meorije *auto*; njen pomeraj u odnosu na registar BP je  $-2$ ).
- Pokazivačka promenljiva *b* je takođe u statičkoj zoni memorije i neka je ona zapisana na adresi 102.

U skladu sa navedenim načinom smeštanja promenljivih, navedene naredbe dodele će biti prevedene na sledeći način.

1. Naredbi *i = j*; odgovara kod

```
MOV AI, [0100]
```

pri čemu je direktno registarsko adresiranje iskorišćeno za pristup za pristup promenljivoj *i*, a direktno memorijsko adresiranje za pristup promenljivoj *j*.

2. Naredbi *a = \*b*; odgovara kod

```
MOV BX, [0102] ; direktno memorijsko adresiranje za
                ; pristup pokazivaču b
MOV AX, [BX]   ; indirektno registarsko adresiranje
                ; za pristup neimenovanoj promenljivoj
                ; na koju ukazuje b
```



```
MOV [BP-02], AX ; bazno adresiranje za pristup  
                ; promenljivoj a
```

## 14.6 Realizacija generatora koda

Pre nego definišemo sam generator koda, uvešćemo ograničenje da se rezultat izvršavanja neke operacije zadržava u registru sve dok je potreban. Iz registra ćemo ga prebacivati u memorijsku lokaciju samo u slučaju kad je taj registar potreban za neku drugu operaciju kao i neposredno pre poziva potprograma, pre naredbe skoka kao i pre naredbe sa labelom.

### 14.6.1 Algoritam generisanja koda

Generator koda preuzima naredbe iz ulazne sekvence koja predstavlja troadresni međukod. U okviru ovih sekvenci se obično identifikuju delovi (blokovi) koji sami za sebe predstavljaju celine, tako da se može učitavati blok po blok. Takođe, za svaki od raspoloživih registara koristimo deskriptor registra koji ukazuje na to vrednost koje promenljive je memorisana u registru. Takođe, za svaku od promenljivih imamo deskriptor adrese koji ukazuje na to gde je memorisana vrednost te promenljive.

Za svaku troadresnu naredbu oblika  $x := y \text{ op } z$  izvršavaju se sledeći koraci.

1. **Određivanje lokacije  $L$  u kojoj će biti zapamćen rezultat.** U ove svrhe koristi se funkcija `getreg` koja je data neposredno iza ovog algoritma. Nastoji se da odredište bude neki od registara u procesoru, ali isto tako to može da bude i neka od memorijskih lokacija.
2. **Određivanje podatka nad kojim se izvršava operacija.** Određuje se gde se nalazi podatak  $y$  nad kojim se izvršava operacija. On može da bude u memoriji ili u nekom od registara procesora. Ukoliko se nalazi na oba mesta prednost se daje registrima. Ako podatak već nije u  $L$ , generiše se naredba `MOV L, y'`, kojom se podatak  $y$  prebacuje u lokaciju  $L$  (sa  $y'$  je označena adresa lokacije u kojoj se nalazi podatak  $y$ ).
3. **Generisanje naredbe.** Generiše se naredba oblika `OP L, z'`, gde je sa  $z'$  označena lokacija u kojoj se nalazi podatak simbolički označen sa  $z$ . Deskriptor adrese za  $x$  se koriguje tako da pokazuje na lokaciju  $L$ . Ako je  $L$  registar, treba korigovati njegov deskriptor tako da pokazuje da on sadrži podatak  $x$ , i izbrisati  $x$  iz sadržaja svih drugih deskriptora registara.
4. **Oslobađanje registara.** Ako se podaci  $y$  i  $z$  ne koriste neposredno iza razmatrane naredbe, nisu živi po izlasku iz bloka onda, ako se nalaze u registrima treba osloboditi registre i naznačiti da se posle bloka ti sadržaji ne koriste.

### 14.6.2 Funkcija getreg

Funkciju getreg koristimo da odredimo lokaciju gde će biti smešten rezultat generisane naredbe. Ako je cilj generisanje optimalnog koda, ovaj zadatak može da bude jako složen. Ovde će biti razmatrano jedno jednostavno rešenje koje daje prednost registrima procesora u odnosu na memorijske lokacije ako ta mogućnost postoji.

1. Ako deskriptor nekog registra pokazuje da je  $y$  u tom registru i ako se  $y$  više ne koristi posle naredbe  $x := y \text{ op } z$ , onda se taj registar vraća kao  $L$ . Kako će posle izvršenja naredbe u ovom registru biti rezultat operacije, a ne više  $y$  treba korigovati deskriptor adrese za  $y$  da  $y$  više nije u tom registru.
2. Ako prva solucija nije moguća, za  $L$  uzimamo prazan registar, ako takav postoji.
3. Ako ni druga solucija nije moguća i ako se  $x$  koristi dalje u istom bloku, ili je  $op$  operator koji zahteva upotrebu registara, onda biramo neki od zauzetih registara  $R$ . Vrednost koja je trenutno zapisana u registru prebacujemo u memoriju (generišemo naredbu `MOV M, R`) ako ta vrednost već nije zapamćena u memoriji, korigujemo deskriptor adrese za  $M$  i vraćamo  $R$  kao  $L$ . Ako je u registru  $R$  zapisana vrednost većeg broja promenljivih (što može da bude posledica naredbi za kopiranje) treba generisati odgovarajuću `MOV` naredbu za svaku od tih promenljivih.
4. Ako se  $x$  ne koristi dalje u bloku ili ne može da se nađe odgovarajući prazan registar onda se za  $L$  uzima memorijska lokacija u kojoj je zapamćena vrednost za  $x$ .

#### Primer 14.5

Razmotrimo kao primer naredbu dodeljivanja

$$d = (a - b) + (a - c) + (a - c)$$

na osnovu koje je generisan sledeći međukod:

$$t_1 := a - b$$

$$t_2 := a - c$$

$$t_3 := t_1 + t_2$$

$$d := t_3 + t_2$$

Genarator koda opisan gore datom procedurom generisaće sekvencu assemblyskih naredbi prikazanu u tabeli 14.6.

TABELA 14.6: Generisani kod za blok naredbi iz primera 14.5

NAREDBA MEĐUKODA	GENERISANI KOD	DESKRIPTOR REGISTRA	DESKRIPTOR ADRESE
		registri prazni	$a, b, c$ i $d$ u memoriji
$t_1 := a - b$	MOV AX, a SUB AX, b'	AX sadrži $t_1$	$t_1$ u AX $a, b$ i $c$ u memoriji
$t_2 := a - c$	MOV BX, a SUB BX, c	AX sadrži $t_1$ BX sadrži $t_2$	$t_1$ u AX $a, b$ i $c$ u memoriji $t_2$ u BX
$t_3 := t_1 + t_2$	ADD AX, BX	AX sadrži $t_3$ BX sadrži $t_2$	$t_2$ u BX $a, b$ i $c$ u memoriji $t_3$ u AX
$d := t_3 - t_2$	ADD AX, BX MOV d, AX	AX sadrži $d$ registri prazni	$d$ u AX $a, b$ i $c$ u memoriji $a, b, c$ i $d$ u memoriji

### 14.6.3 Generisanje koda za neke specijalne naredbe

Složenije naredbe troadresnog koda, kao što su naredbe u kojima se javljaju indeksirane promenljive i pointeri, se transformišu u asemblerski kod na sličan način kao i naredbe sa binarnim operatorom.

### 14.6.4 Generisanje koda za naredbe sa indeksnim promenljivama

Kod koji se generiše za slučaj naredbi dodeljivanja u kojima se javljaju promenljive sa indeksima prikazan je u tabeli 9.5.

U tabeli 14.7 su razmatrani slučajevi naredbi  $a := b[i]$  i  $a[i] := b$ , a uzeta su u obzir tri slučaja: kada je indeks zapamćen u registru procesora, u memorijskoj lokaciji ili se čuva na steku.

### 14.6.5 Generisanje koda za naredbe sa pokazivačima

Kod koji se generiše u slučaju naredbi u kojima se koriste pokazivači prikazan je u tabeli 14.8. I ovde su uzeta u obzir dva oblika međunaredbi:  $a := *p$  i  $*p := a$ . Takođe su razmatrane slučajevi kada se vrednost pokazivača nalazi u registru procesora, u memorijskoj lokaciji ili na steku.

TABELA 14.7: Generisani kod za naredbe sa indeksima

NAREDBA MEĐUKODA	INDEKS $i$ JE U REGISTRU	INDEKS $i$ JE U MEMORIJI $M_i$	INDEKS $i$ JE NA STEKU SA POMERAJEM $d_i$ U ODNOSU NA BP
$a := b[i]$	MOV AX, b[SI] MOV a, AX	MOV SI, Mi MOV AX, b[SI] MOV a, AX	MOV SI, [BP + di] MOV AX, b[SI] MOV a, AX
$a[i] := b$	MOV AX, b MOV a[DI], AX	MOV AX, b MOV DI, Mi MOV a[DI], AX	MOV AX, b MOV DI, [BP + di] MOV a[DI], AX

TABELA 14.8: Generisani kod za naredbe sa pokazivačima

NAREDBA MEĐUKODA	POKAZIVAČ $p$ JE U REGISTRU BX	POKAZIVAČ $p$ JE U MEMORIJI $M_p$	POKAZIVAČ $p$ JE NA STEKU SA POMERAJEM $d_p$ U ODNOSU NA BP
$a := *p$	MOV AX, [BX] MOV a, AX	MOV BX, Mp MOV AX, [BX] MOV a, AX	MOV BX, [BP + dp] MOV AX, [BX] MOV a, AX
$*p := a$	MOV AX, a MOV [BX], AX	MOV BX, Mp MOV AX, a MOV [BX], AX	MOV BX, [BP + dp] MOV AX, a MOV [BX], AX

#### 14.6.6 Generisanje koda za naredbe uslovnog skoka

Naredbe uslovnog skoka troadresnog koda oblika `if x rel op y goto z` mogu da budu transformisane u asemblerski, u principu, na dva načina.

Jedan način je da se najpre  $x$  oduzme od  $y$  i da se rezultat zapamti u nekom registru  $R$ , nakon čega se generiše naredba skoka na  $z$  ako je vrednost registra  $R$  negativna.

Drugi način se primenjuje kod asemblerskih jezika koji koriste statusni registar (kakav je i jezik za procesor 8086) – registar koji sadrži skup bitova kojima se registruje kakva je rezultujuća vrednost poslednje izvršene operacije bilo da je to bila aritmetička operacija ili operacija prenosa vrednosti iz memorije u registar ili obrnuto. Ovi jezici obično sadrže naredbe skoka koje se vezuju za te bitove i omogućavaju prenos upravljanja na neku naredbu ako je neki od uslova ispunjen. U ovom slučaju, naredba međukoda oblika `if x < y goto z` će biti preslikana u sledeću sekvencu naredbi:

```
CMP    'x, 'y
JNGE   z
```

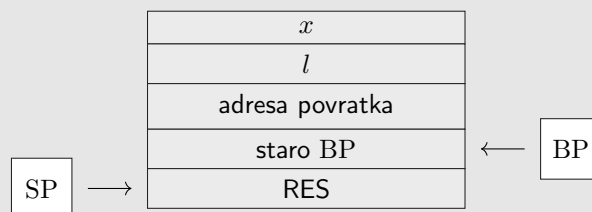
Najpre se poredi vrednost podataka  $x$  i  $y$ , nakon čega se postavljaju vrednosti bitova kojima se registruje sadržaj rezultata. Druga naredba je skok na labelu  $z$  ako je u bitu kojim se registruje uslov  $<$  upisana vrednost 1.

### Primer 14.6

Definisati 8086 kod i izgled aktivacionog sloga funkcije `find` koja proverava da li zadati podatak postoji u lančanoj liti. Zadatak rešiti uz pretpostavku da se rezultat funkcije smešta u registar CX i uz pretpostavku da se rezultat funkcije prenosi kroz listu parametara.

```
struct list
{
    int info;
    struct list* next;
};
int find(struct list* l, int x)
{
    int res;
    if ( l == 0 )
        res = 0;
    else if ( l->info == x )
        res = 1;
    else
        res = find(l->next, x);
    return res;
}
```

Aktivacioni slog funkcije `find` u slučaju kad se rezultat upisuje u registar CX prikazan je na sledećoj slici.



Generisani asemblerski kod u tom slučaju bi bio:

```
; upis BP na stek:
    PUSH BP

; promena vrednosti pokazivaca BP:
    MOV BP, SP
```

```
; rezervacija prostora za lokalnu promenljivu:
    ADD SP, 2

; telo funkcije:
    MOV AX, [BP + 4]
    CMP AX, 0
    JNE lab1
    MOV [BP - 2], 0
    JMP lab2
lab1: MOV BX, [BP + 4]
    MOV AX, [BX]
    CMP AX, [BP + 6]
    JNE lab3
    MOV [BP - 2], 1
    JMP lab2

; upis stvarnih parametara u stek:
lab3: MOV AX, [BP + 6]
    PUSH AX
    MOV BX, [BP + 4]
    MOV AX, [BX + 2]
    PUSH AX

; poziv funkcije:
    CALL find

; izbacivanje stvarnih parametara iz steka:
    ADD SP, 4

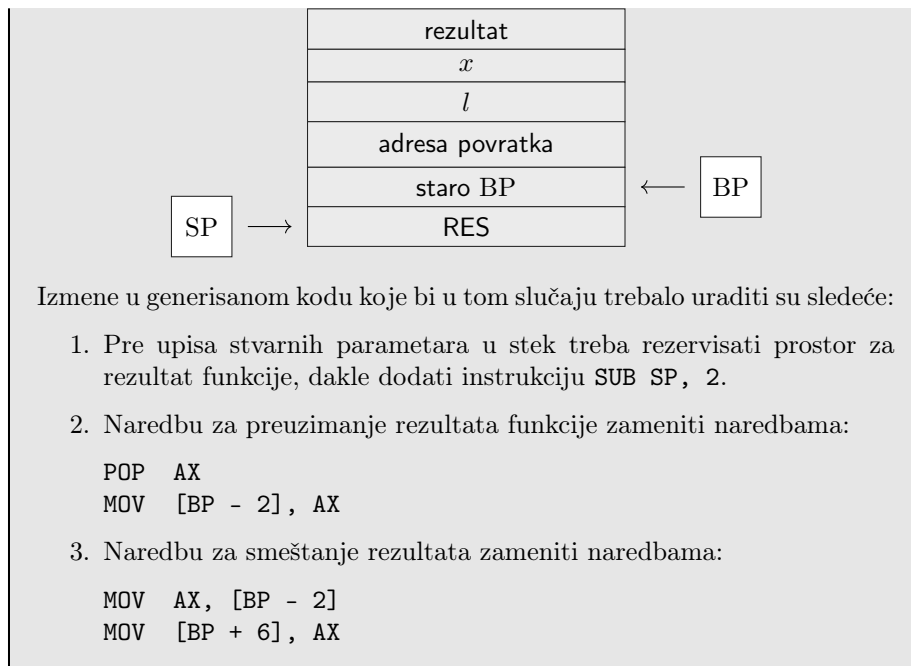
; preuzimanje rezultata funkcije:
    MOV [BP - 2], CX

; smestanje rezultata u CX:
lab2: MOV CX, [BP - 2]

; izbacivanje lokalnih promenljivih iz steka:
    MOV SP, BP

; uzimanje stare vrednosti BP-a:
    POP BP
    RET
```

Aktivacioni slog funkcije `find` u slučaju kad se rezultat prenosi kroz listu parametara prikazan je na sledećoj slici.



## 14.7 Pitanja

1. Šta su zadaci komponente za upravljanje memorijom.
2. Kako se viši programski jezici dele na osnovu toga kako je organizovana memorija koja im se dodeljuje u toku izvršenja.
3. Šta je to aktivacioni slog i kako izgleda njegov uobičajeni sadržaj?
4. Kako je organizovana memorija pri izvršavanju programa kod polustatičkih programskih jezika.
5. Šta se podrazumeva pod sekvencom poziva, a šta pod sekvencom povratka?
6. Objasniti skup instrukcija koje treba generisati pri prevođenju poziva potprograma.
7. Objasniti skup instrukcija koje treba generisati pri prevođenju povratka iz potprograma.
8. Pokazati na primeru kako treba da izgledaju generisane instrukcije u assemblerском jeziku 8086 za pristup podacima smeštenim u različitim zonama memorije.

## 14.8 Zadaci

1. Definisati aktivacioni slog i odgovarajući 8086 kod za funkciju `sum`. Pretpostaviti da se rezultat funkcije prenosi kroz registar CX.

```
struct node
{
    int info;
    struct node* next;
};

int sum( struct node* head )
{
    int s=0;
    if (head != 0)
        s = head->info + sum(head->next);
    return s;
}
```

2. Data je funkcija za nalaženje maksimuma u uređenom binarnom stablu:

```
struct node
{
    int info;
    struct node* left;
    struct node* right;
};

int max(struct node* head)
{
    int m;
    if ( head->right == 0 )
        m = head->info;
    else
        m = max(head->right);
    return m;
}
```

- (a) Definisati sadržaj aktivacionog sloga funkcije `max`. Pretpostaviti da se rezultat funkcije prenosi kroz listu parametara.
- (b) Definisati 8086 asemblerski kod za datu funkciju.



## Glava 15

# Realizacija kompilatora

Iz svega što je u prethodnim poglavljima rečeno može se zaključiti da je kompilator sam po sebi jedan nešto složeniji program koji se može napisati na nekom višem programskom jeziku koji omogućava rad sa strukturama kakve su liste stabla i slično. U suštini, najveći problem je bio da se napravi prvi kompilator koji je morao da bude napisan na nekom asemblerskom ili mašinskom jeziku. Sve naredne generacije kompilatora su pisane na višim programskim jezicima, a kako smo već videli, danas se za realizaciju pojedinih modula koriste i generatori koda.

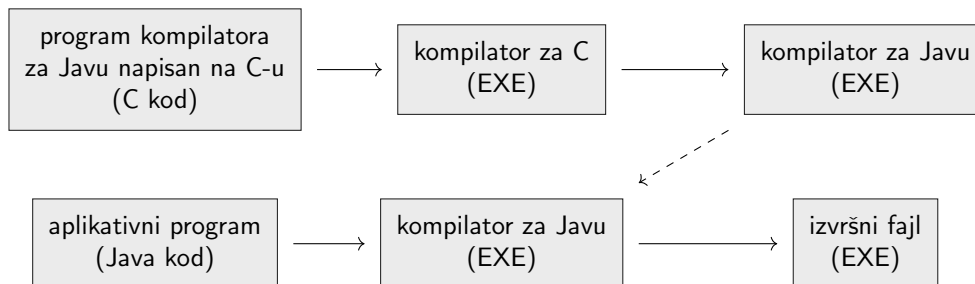
U ovom poglavlju razmotrićemo pristupe koji su korišćeni u realizaciji nekih komercijalnih rešenja kompilatora za programske jezike koji imaju široku primenu.

### 15.1 Tehnika butstrepovanja

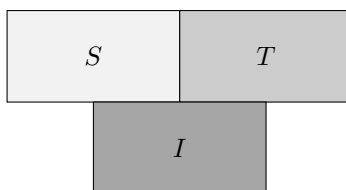
Za razvoj novih verzija kompilatora za određeni programski jezik se mogu koristiti postojeći (stari) kompilatori za isti programski jezik ili za novi programski jezik, tako što se program novog kompilatora piše u jeziku za koji postoji kompilator i prevodi u mašinski kod pomoću postojećeg kompilatora. Ova tehnika u kojoj se postojeći kompilatori koriste za razvoj novih poznata je pod imenom butstrepovanje. Na primer, na slici 15.1 je pokazano kako se postojeći C kompilator može iskoristiti za razvoj Java kompilatora.

U okviru ove tehnike za predstavljanje rešenja kompilatora koriste se dijagrami u kojima je svaki kompilator prikazan jednim  $T$ -blokom kao na slici 15.2.

$T$ -dijagramom obuhvaćene su tri komponente koje karakterišu jedan kompilator. Jednim blokom ( $S$ ) predstavljen je izvorni jezik, jezik sa kojeg se vrši prevođenje (C, Pascal, Java i sl); drugi blok ( $T$ ) se odnosi na ciljni jezik, jezik na koji se vrši prevođenje (assemblerski jezik, mašinski jezik i sl); dok se treći



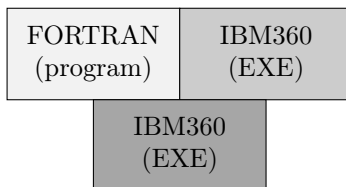
SLIKA 15.1: Razvoj novog kompilatora

SLIKA 15.2: Kompilator predstavljen  $T$ -dijagramom.  $S$  je izvorni jezik (*source language*),  $T$  je ciljni jezik (*target language*) i  $I$  je jezik na kome je napisan kompilator (*implementation language*).

blok ( $I$ ) odnosi na jezik u kom je predstavljen sam kompilator. Dok je kompilator na nivou programa, taj treći blok će biti neki viši programski jezik (C, Pascal, Java i sl), a kada se prevede i koristi za prevođenje drugih programa taj blok će se odnositi na mašinski jezik u koji je program preveden. Koristićemo oznaku  $SIT$  za kompilator predstavljen dijagramom na slici 15.2.

### Primer 15.1 Kompilator za FORTRAN

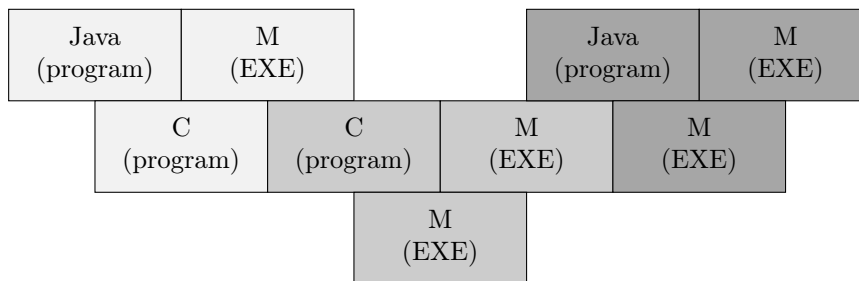
Na slici 15.3,  $T$ -dijagramom je predstavljen kompilator za programski jezik FORTRAN koji generiše programe za mašinu IBM360 i sam predstavlja izvršni program koji može da se koristi na istoj toj mašini, odnosno preveden je u mašinski jezik računara IBM360.



SLIKA 15.3: Kompilator iz primera 15.1

### 15.1.1 Kompilator za novi programski jezik

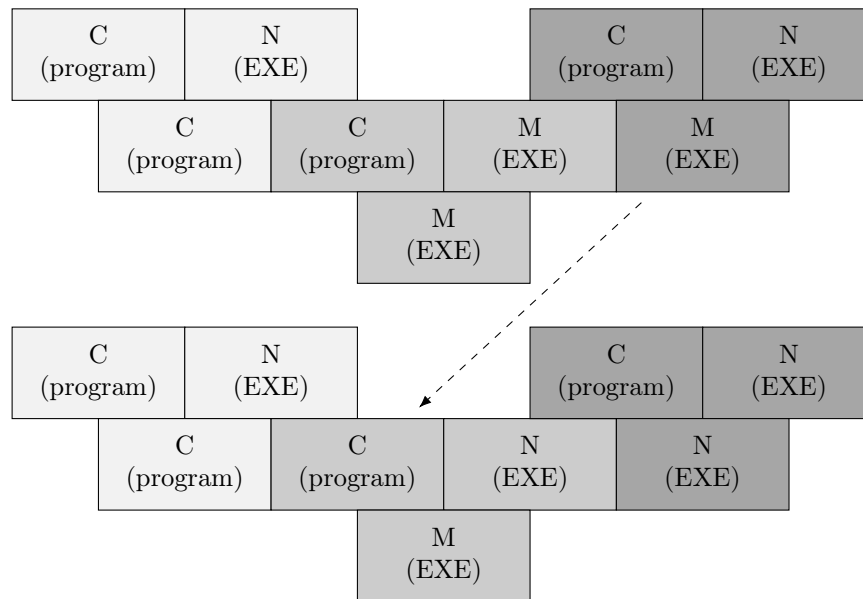
Primenom tehnike butstrepovanja, postojeći kompilatori se mogu iskoristiti za realizaciju kompilatora za novi programski jezik. Na slici 15.1 predstavljeno je kako se postojeći kompilator za programski jezik C može iskoristiti za realizaciju kompilatora za programski jezik Java. Preko  $T$ -dijagrama ovaj proces je predstavljen na slici 15.4. Raspolažemo kompilatorom za programski jezik C na mašini M ( $C_M M$ ), a cilj nam je da realizujemo kompilator za programski jezik Java koji će se koristiti na istoj mašini ( $Java_M M$ ). Program novog kompilatora pišemo na programskom jeziku C i prevodimo ga pomoću postojećeg kompilatora u mašinski kod.



SLIKA 15.4: Generisanje kompilatora za  $Java_M M$

### 15.1.2 Prenosjenje kompilatora sa jedne na drugu mašinu

Tehnikom Butstrepovanja može da se reši i zadatak generisanja kompilatora za novu mašinu. Raspolaže se kompilatorom za neki programski jezik na jednoj mašini i zadatak je da se realizuje kompilator za isti taj jezik ali za novu mašinu. Na primer hoćemo da kompilator za programski jezik C za mašinu M ( $C_M M$ ) iskoristimo za realizaciju kompilatora za C ali koji će moći da se primenjuje na mašini N ( $C_N N$ ). Za pisanje novog kompilatora koristimo programski jezik C. Pišemo program kompilatora koji za programe napisane na jeziku C generiše EXE fajl za novu mašinu. Ovaj program prevodimo kroz postojeći kompilator pri čemu se dobija kompilator ( $C_M N$ ) koji može da se koristi za generisanje EXE fajlova za novu mašinu, ali to mora da se radi na staroj mašini. Taj kompilator je samo privremeno rešenje. Program kompilatora koji je napisan na jeziku C treba prevesti ovim privremenim kompilatorom kako bi se dobilo konačno rešenje, kompilator koji može da se koristi za prevođenje C programa na novoj mašini, slika 15.5.

SLIKA 15.5: Realizacija kompilatora  $C_{NN}$  pomoću kompilatora  $C_{MM}$ 

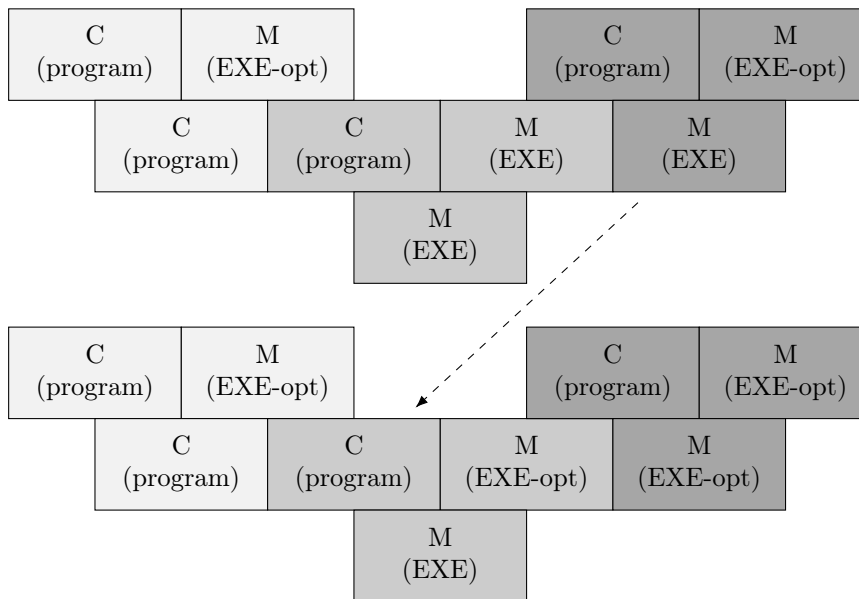
### 15.1.3 Optimizacija postojećeg kompilatora

Butstrepanjem se rešava i problem optimizacije postojećih kompilatora. Krećemo opet od kompilatora za jezik C na mašini M ( $C_{MM}$ ). Cilj je da se optimizuje ovaj kompilator tako da generiše efikasniji kod i da sam (kao program) bude efikasniji. U ovom slučaju, na jeziku ponovo pišemo program kompilatora za programski jezik C ali sa ugrađenim algoritmima za optimizaciju. Taj program se prevodi pomoću postojećeg kompilatora ( $C_{MM}$ ) pri čemu se dobija kompilator koji generiše optimizovani EXE fajl programa koji se prevodi ali sam nije optimizovan. Ovaj kompilator je samo privremeno rešenje. Kada se pomoću ovog kompilatora prevede polazni C program dobija se konačno rešenje, kompilator koji generiše optimizovane EXE fajlove i sam je optimizovan ( $C_{M-opt}M-opt$ ).

U praksi se tehnika butstrepanja veoma često koristi i u više koraka pri čemu se rešavaju različiti problemi. U nastavku ćemo detaljnije razmotriti dva slučaja kada je ova tehnika višestruko korišćena u komercijalne svrhe.

### 15.1.4 Razvoj kompilatora za programski jezik Pascal

Prvi primer se odnosi na proces koji je primenjen kod generisanja kompilatora za programski jezik Pascal 1981 kada je zadatak bio da se realizuje kompilator za novu, nešto izmenjenu verziju jezika Pascal kao i da se pri tome ugrade i neki

SLIKA 15.6: Optimizacija kompilatora  $C_M M$ 

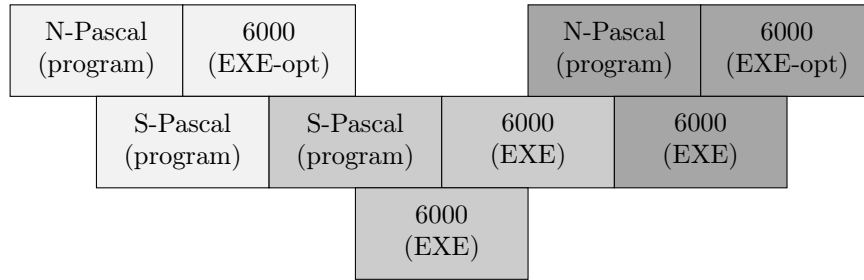
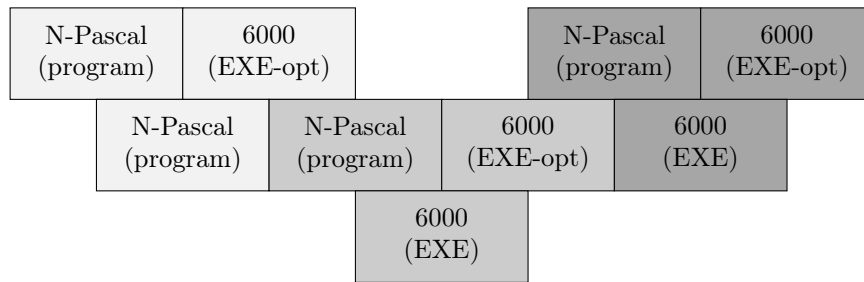
postupci za optimizaciju kompilatora koji su tada bili poznati. Za razvoj novog kompilatora korišćen je kompilator za staru verziju Pascal-a koji je na mašini CDC 6000 napisan 1972. godine. U ovom slučaju butstrepanje je izvršeno u nekoliko koraka.

1. Na starom Pascal-u napisan je program kompilatora  $NP_{SP6000-opt}$  koji treba da prevodi sa novog Pascal-a i da generiše optimizovan kod. Ovaj program je propušten kroz postojeći kompilator  $SP_{6000}6000$ . Dobijen je kompilator  $NP_{6000}6000-opt$  za novi Pascal koji generiše oprimizirani kod, ali sam nije optimiziran (slika 15.7).
2. U sledećem koraku, cilj je bio da se optimizuje i sam kod dobijenog kompilatora. Isti program kompilatora prekodiran je sa starog Pascal-a u novi Pascal i propušten kroz dobijeni kompilator. Znači, napisan je kompilator  $NP_{NP6000-opt}$ , propušten kroz kompilator  $NP_{6000}6000-opt$  i dobijen  $NP_{6000-opt}6000-opt$  (slika 15.8).

Prethodna dva koraka objedinjeno su predstavljena na slici 15.9.

### 15.1.5 Razvoj kompilatora za programski jezik FORTRAN

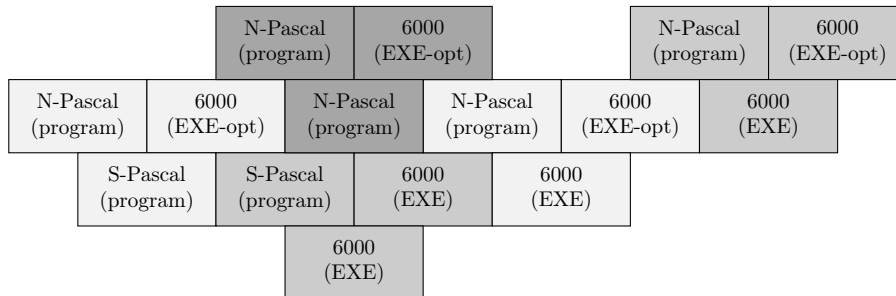
Drugi slučaj koji ćemo razmotriti odnosi se na proces generisanja kompilatora za programski jezik FORTRAN za mašinu IBM 360, kada je za razvoj korišćen postojeći kompilator za FORTRAN za mašinu IBM 7094. Cilj je bio da se

SLIKA 15.7: Generisanje kompilatora  $NP_{6000}6000\text{-opt}$ SLIKA 15.8: Generisanje kompilatora  $NP_{6000\#}6000\text{-opt}$ 

kompilator prenese na novu mašinu i da se pri tome dobije bolje, optimizovano rešenje. Znači, raspolagalo se kompilatorom  $F_{7094}7094$ , a cilj je bio da se dobije kompilator  $F_{360\#}360\#$  (znak  $\#$  i ovog puta označava optimizovanu verziju kompilatora).

U prvoj fazi rešavan je samo problem prenošenja kompilatora sa stare na novu mašinu kroz sledeća dva koraka.

1. Korišćena je mašina IBM 7094. Na Fortranu je napisan kompilator  $F_F360$  koji prevodi programe sa Fortrana u mašinski kod mašine IBM 360. Ovaj program je propušten kroz postojeći kompilator  $F_{7094}7094$  na mašini IBM



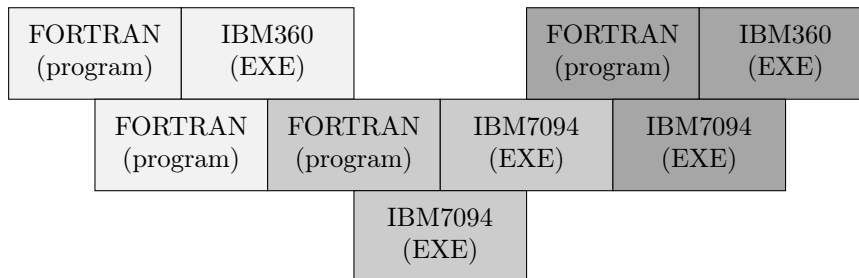
SLIKA 15.9: Objedinjeni koraci butstrepovanja sa slika 15.7 i 15.8

7094. Dobijen je kompilator  $F_{7094}360$  koji može na staroj mašini IBM 7094 da generiše programe za mašinu IBM 360 (slika 15.10).

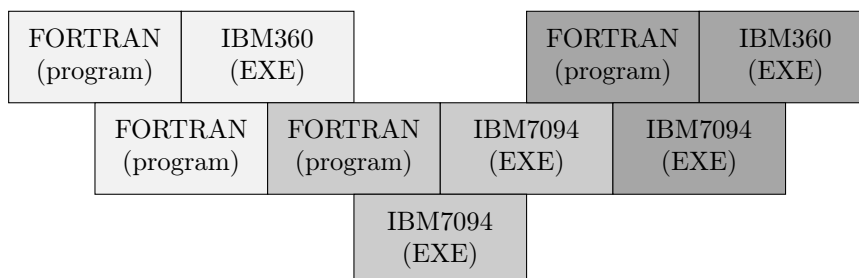
2. Korišćena je mašina IBM 7094. Napisani program kompilatora  $F_F360$  propušten je kroz novodobijeni kompilator  $F_{7094}360$ . Dobijen je kompilator  $F_{360}360$  koji može da se koristi na mašini IBM 360 za generisanje programa za istu tu mašinu (slika 15.11).

Prethodnim koracima, koji su objedinjeno predstavljeni na slici 15.12, rešen je problem prenosa kompilatora na novu mašinu. U nastavku je rešavan problem optimizacije kompilatora.

3. Korišćena je mašina IBM 360. Napisan je optimizovani kod kompilatora  $F_F360\#$  i propušten kroz kompilator  $F_{360}360$ , dobijen u prethodnoj fazi. Kao rezultat je dobijen je kompilator  $F_{360}360\#$  koji generiše optimizovani kod ali sam nije optimizovan (slika 15.13).
4. Korišćena je mašina IBM 360. Vršena je optimizacija samog kompilatora. Program kopilatora  $F_F360\#$  propušten je kroz novodobijeni kompilator  $F_{360}360\#$  koji generiše optimizovani kod. Dobijen je kompilator  $F_{360\#}360\#$ , koji generiše optimizovani kod i sam je optimizovan (slika 15.14).

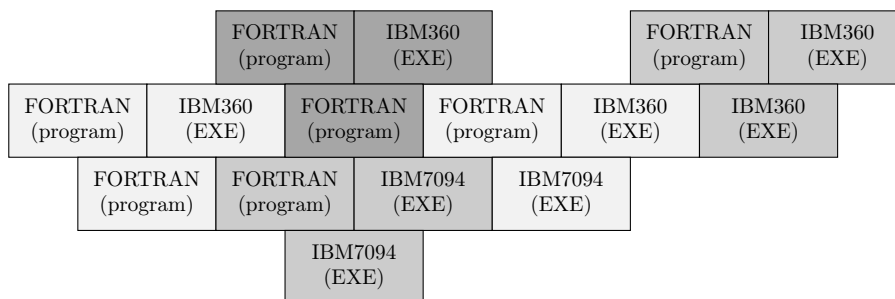


SLIKA 15.10: Prva faza butstrepanja

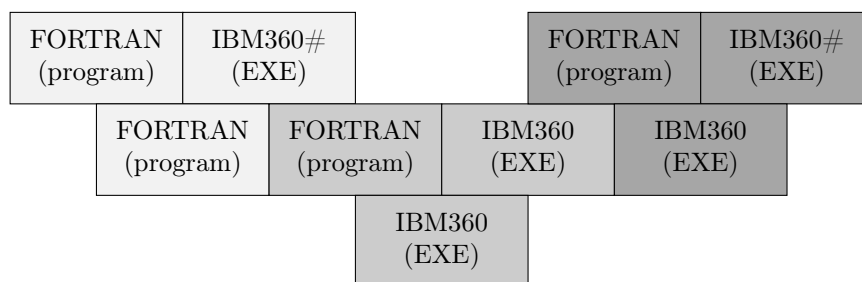


SLIKA 15.11: Druga faza butstrepanja

Sva četiri koraka butstrepanja koje je izvršeno u slučaju generisanja optimizovanog kompilatora za FORTRAN za mašinu IBM 360 prikazana su objedinjeno

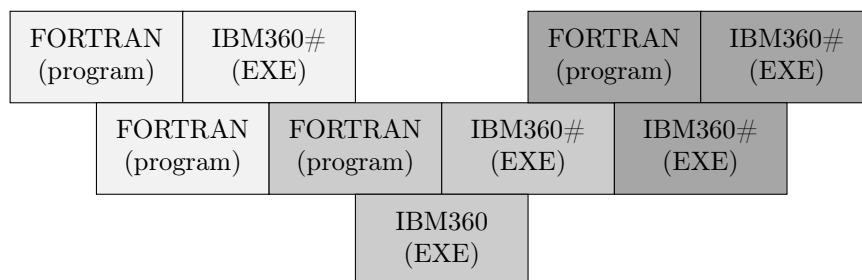


SLIKA 15.12: Objedinjene prve dve faze butstrepanja sa slika 15.10 i 15.11



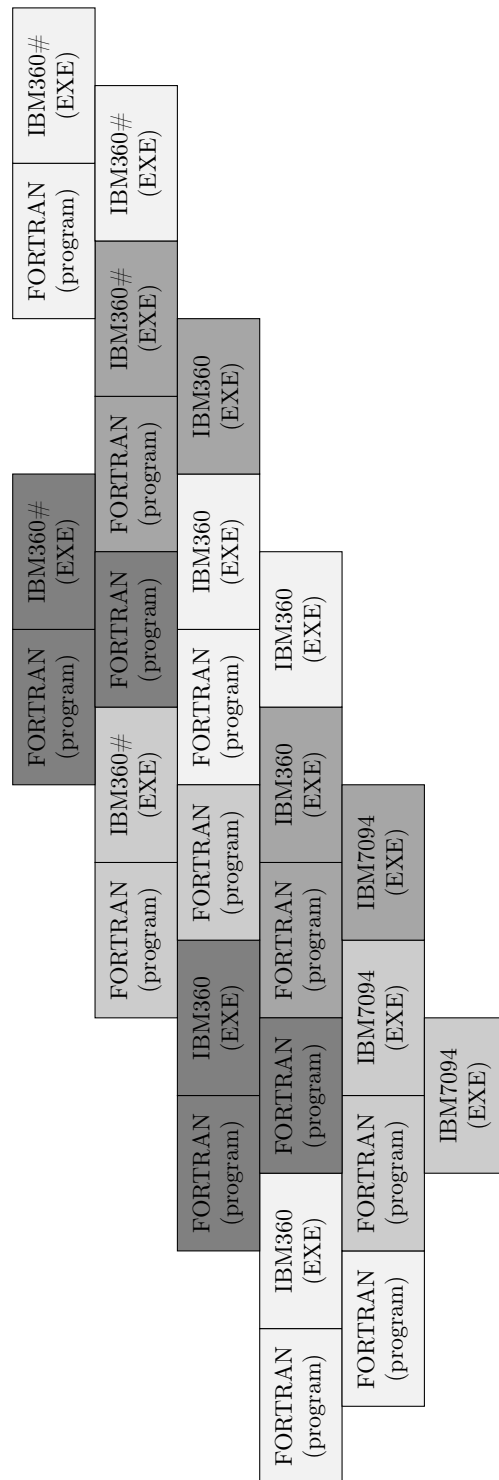
SLIKA 15.13: Treći korak butstrepanja

na slici 15.15. Inače, ovaj kompilator je bio poznat po tome što su kod njega prvi put primenjene neke tehnike optimizacije koje se danas smatraju standardnim.



SLIKA 15.14: Četvrti korak butstrepanja



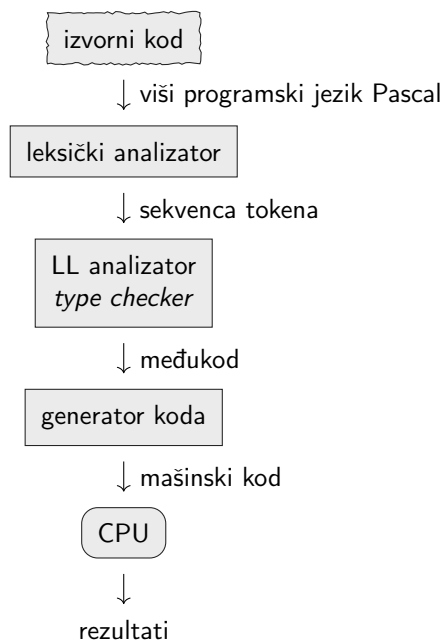


SLIKA 15.15: Sve faze butstrepovanja primenjenog kod generisanja kompilatora za FORTRAN

## 15.2 Struktura kompilatora

Kod praktičnih rešenja kompilator se obično realizuje kao višeprolazni sistem u kome se jedna na drugu nadovezuje više faza kompilatora. Kako će biti organizovane faze kompilatora i šta će one obuhvatati zavisi od mnogih faktora. Ovde ćemo dati strukturu nekih poznatih praktičnih rešenja kompilatora.

Na slici 9.5 prikazana je struktura kompilatora za programski jezik Pascal koji je 1971. godine realizovao prof. Wirt sa svojim saradnicima. Poznata je u literaturi kao P verzija kompilatora za Pascal zato što je tada prvi put upotrebljen poseban P međukod.



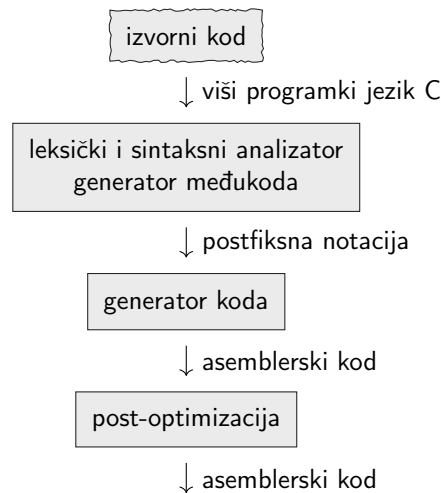
SLIKA 15.16: P kompilator za Pascal

### 15.2.1 C kompilator za mašinu PDP 11

Na slici 15.17 prikazan je kompilator za programski jezik C, za mašinu PDP 11.

### 15.2.2 FORTRAN H kompilator

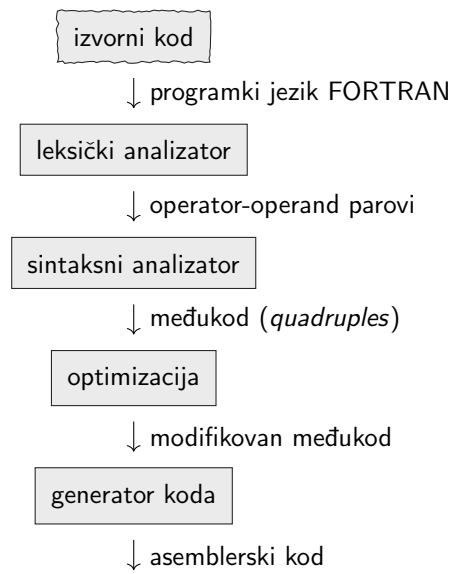
Na slici 15.18 prikazan je kompilator za programski jezik FORTRAN poznat u literaturi kao FORTRAN H kompilator.



SLIKA 15.17: Kompilator za programki jezik C za mašinu PDP 11

## 15.3 Pitanja

1. Šta su to T-dijagrami i u koje svrhe se koriste?
2. Objasniti tehniku butstrepanja.
3. Prikazati kako se tehnikom butstrepanja postojeći kompilator može iskoristiti za realizaciju kompilatora za novi programski jezik. Prikazati taj proces preko T-dijagrama.
4. Prikazati kako se tehnikom butstrepanja postojeći kompilator može iskoristiti za realizaciju naprednije (optimizovane) verzije kompilatora za isti programski jezik. Prikazati taj proces preko T-dijagrama.
5. Prikazati kako se tehnikom butstrepanja postojeći kompilator može iskoristiti za prenošenje kompilatora na novu platformu (realizaciju kompilatora za isti programski jezik ali za novu platformu). Prikazati taj proces preko T-dijagrama.
6. Opisati kako je tehnika butstrepanja iskorišćena u procesu realizacije kompilatora za programski jezik Pascal kada su u staru verziju jezika unete i neke izmene. Prikazati taj proces jednim T-dijagramom.
7. Opisati kako je tehnika butstrepanja iskorišćena u procesu realizacije kompilatora za programski jezik Fortran za platformu IBM 360, kada je ujedno izvršena i optimizacija tog kompilatora. Prikazati taj proces jednim T-dijagramom.



SLIKA 15.18: FORTRAN H kompilator

## Glava 16

# Dodaci

### 16.1 Java bajtkod

OZNAKA	KôD	DRUGI BAJTOVI	MAGACIN	OPIS
aaload	32		arrayref, index $\rightarrow$ value	učitava u magacin referencu na polje
aastore	53		arrayref, index, value $\rightarrow$	pamti referencu na polje
aconst_null	01		$\rightarrow$ null	ubacuje <i>null</i> referencu u magacin